

Hands-On Cisco Automation with **Python**

Streamline Network Tasks Using Netmiko,
NAPALM, and Nornir for Beginners





Hands-On Cisco Automation with **Python**

Streamline Network Tasks Using Netmiko,
NAPALM, and Nornir for Beginners



Hands-On Cisco Automation with Python: Streamline Network Tasks Using Netmiko, NAPALM, and Nornir for Beginners

**Rick Graziani Adrian Iliesiu, CCIE No.
43909**

Cisco Press

Hands-On Cisco Automation with Python: Streamline Network Tasks Using Netmiko, NAPALM, and Nornir for Beginners

Rick Graziani and Adrian Iliesiu

Copyright © 2026 Cisco Systems, Inc.

Published by:

Cisco Press

Hoboken, New Jersey

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit <https://www.pearson.com/global-permission-granting.html>.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

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

\$PrintCode

Library of Congress Control Number: 2026932840

ISBN-13: 978-0-13546-319-2

ISBN-10: 0-1-35-46319-X

Warning and Disclaimer

This book is designed to provide information about using Python with Netmiko, NAPALM and Nornir to configure and manage Cisco devices. 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 author and are not necessarily those of Cisco Systems, Inc.

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Cisco Press or Cisco Systems, Inc., cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

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>.

Head of IT & Professional Learning, Enterprise Learning and Skills

Julie Phifer

Alliances Manager, Cisco Press

Caroline Antonio

Executive Editor

James Manly

Managing Editor

Sandra Schroeder

Development Editor

Ellie C. Bru

Senior Project Editor

Mandie Frank

Copy Editor

Kitty Wilson

Technical Editors

Allan Johnson, Jozef Janitor

Designer

Chuti Prasertsith

Composition

codeMantra

Indexer

Proofreader

About the Authors

Rick Graziani has been teaching and working in the computer networking industry since 1980. He is a full-time Computer Science/Computer Information Systems faculty member at Cabrillo College and an adjunct Computer Science and Engineering faculty member at the University of California, Santa Cruz. Rick also works for Cisco Systems on the Curriculum Engineering team as part of Cisco Networking Academy. He has written several books for Cisco Press, including *Fundamentals of IPv6*, and has presented at technical conferences for Cisco Networking Academy. Prior to teaching, he worked in the information technology field for Santa Cruz Operation, Tandem Computers, and Lockheed Martin and served in the U.S. Coast Guard. When he is not working, he is most likely surfing at one of his favorite Santa Cruz surf breaks.

Adrian Iliesiu, CCIE No. 43909 (EI), is a network engineer with more than 20 years of professional experience in the field. He currently works as a Principal Engineer within the Cisco DevNet organization. Throughout his career, Adrian has held a wide range of roles, including team leader and network, systems, and QA engineer, across multiple industries and international organizations. In his current role at Cisco, Adrian focuses on advancing network programmability and automation, with particular emphasis on enterprise infrastructure. He is an established author, a distinguished speaker at Cisco Live!, and a recipient of the

prestigious Cisco Pioneer Award. Adrian has also appeared on Cisco TechWise, Cisco Champion podcasts, and DevNet webinars. He is the host of the Simplifying Network Automation with NerGru series of livestreams. He holds a bachelor's degree in electronics and telecommunications from the Technical University of Cluj-Napoca and a master's degree in telecommunication networks from Politehnica University of Bucharest.

About the Technical Reviewers

Allan Johnson entered the academic world in 1999, after 10 years as a business owner/operator to dedicate his efforts to his passion for teaching. He holds both an MBA and an MEd in training and development. He taught CCNA courses at the high school level for seven years and has taught both CCNA and CCNP courses at Del Mar College in Corpus Christi, Texas. In 2003, Allan began to commit much of his time and energy to the CCNA Instructional Support Team, providing services to Cisco Networking Academy instructors worldwide and creating training materials. He now splits his time between working as a curriculum lead for Cisco Networking Academy and as an account lead for Unicon (unicon.net), supporting Cisco's educational efforts.

Jozef Janitor is a Product Manager at Cisco Networking Academy, where he oversees the Networking, Cybersecurity, and Automation educational portfolios, which are focused on equipping learners with digital skills to become the next generation of IT professionals. Before joining Cisco a decade ago, he cofounded a startup, architected a university campus network, and helped small businesses manage their IT infrastructure. He is passionate about new and emerging technologies and enjoys good food, music, hiking, and bicycling in his free time.

Dedications

This book is dedicated to my beautiful and amazing wife, Alice Chialastri. This book would not have been possible without your continuous love, encouragement, and support. I am forever grateful for your understanding and patience as I wrote this book during many long nights and weekends.

—Rick Graziani

This book is dedicated to my family. Thank you for your endless love, patience, and support. This book exists because of you.

—Adrian Iliesiu

Acknowledgments

Rick Graziani:

I would like to begin by thanking my co-author and friend, Adrian Iliesiu. Your knowledge and passion, along with your dedication to help others obtain this knowledge, has made this book possible.

I would like to also express my sincere appreciation and gratitude to the technical editors for this book, Allan Johnson and Jozef Janitor. Their dedication and meticulous work ensured that this book is both technically accurate and clear. This book would not be what it is without their contributions.

I would also like to thank James Manly, the executive editor. We have worked on many projects together over the years, and I am grateful for his continued support and guidance throughout this project.

Thank you to Ellie Bru, the development editor for this book, for shepherding me through multiple development and editing cycles with patience and care, and for her thoughtful editing and patience in answering my many questions along the way.

Thank you to Mandie Frank, the production editor for this book, for your careful attention to detail and your thoughtful questions, which consistently improved clarity, flow, and precision throughout the manuscript. Your work helped

ensure that the final text communicates complex ideas clearly and effectively for the reader.

Finally, I would like to thank all of my students over the last 32 years, especially the Cisco Networking Academy students for the past 29 years. You have always been the catalyst for my passion for teaching and writing. This book—like all of my other books—was written for you and with you in mind.

Adrian Iliesiu:

First, I would like to thank my co-author, Rick Graziani. Rick, I'm amazed at how you can take complex topics and break them down into simple explanations. That is the true mark of an expert. From coming up with the idea for the book to correcting and improving my scribblings, I am truly grateful for the opportunity to collaborate on this project.

I would also like to thank the technical editors of this book, Allan and Jozef. Your timely feedback was invaluable in making this book the best version it could be.

Thank you to the folks at Cisco Press, James Manly and Ellie Bru, for your patience and understanding and for keeping me on track.

Big thank you to the Cisco DevNet community and all the people I've interacted with over the years about network automation topics. This book was written with you in mind, and I hope it helps you on your automation journey no matter at what stage you are.

Contents at a Glance

1 Introducing Netmiko, NAPALM, and Nornir

Part 1: Netmiko

2 Getting Started with Netmiko

3 Configuring Devices with Netmiko

4 Accessing Multiple Devices with Netmiko

Part 2: NAPALM

5 Introducing NAPALM and Structured Data

6 Understanding Python Dictionaries with NAPALM

7 Iterating Through NAPALM Dictionaries

8 Configuring Devices with NAPALM

Part 3: Nornir

9 Introducing Nornir: A Pythonic Framework for Network Orchestration

10 Using Nornir with Netmiko

11 Using Nornir with NAPALM

12 Inventory Management with Nornir

Part 4: What's Next

13 What's Next

Appendixes

A Python Virtual Environments

B Understanding `expect_string` with
`send_command()`

C Using Python Dictionaries as NAPALM Outputs

D The Relationship Between Python Dictionaries
and JSON

E Understanding Objects and Variables in Python

F Using a Recursive Function to Handle Nested
Dictionaries of Any Depth

G Using Tabular Output

H Using Public and Private Keys

I Netmiko-Supported Network Operating Systems

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 9780135463192 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

Chapter 1. Introducing Netmiko, NAPALM, and Nornir

- The 3 *Ns*: Netmiko, NAPALM, and Nornir
- Automation and Programmability for All Levels
- How Much Cisco IOS and Python Do I Need to Know?
- What Do I Need to Get Started?
- Using AI as an Alternative to Physical Equipment
- Summary

Part 1: Netmiko

Chapter 2. Getting Started with Netmiko

- Why Start with Netmiko?
- A First Look at a Netmiko Program
- The Basic Netmiko Framework
- Optional: Exploring Python Classes, Objects, Instances, and Methods
- Your Turn
- Sending Command Output to a Variable
- The `send_command_expect()` Method
- The `save_config()` Method

Debugging Netmiko by Using the Session Log Summary

Chapter 3. Configuring Devices with Netmiko

The send_command() Method

Summary

Chapter 4. Accessing Multiple Devices with Netmiko

Using a Variable to Store an IP Address

Understanding Python for Loops

Iterating Through Multiple Devices

Using Dictionaries to Store Device Connection Parameters

Using Python's getpass() for Secure Password Input

Using Netmiko Exceptions for Troubleshooting

Maintaining Multiple SSH Connections

Simultaneously

Summary

Part 2: NAPALM

Chapter 5. Introducing NAPALM and Structured Data

What Is NAPALM, and How Is It Different from Netmiko?

Installing the NAPALM Library

Basic NAPALM Framework

How NAPALM Retrieves Data Without an API on Cisco IOS

Good News, Bad News: Understanding Structured Data

- get_facts(): Our First NAPALM Method and Dictionary
- Creating a NAPALM Dictionary Without a Device
- NAPALM Methods
- Using Python Interactive Mode to Experiment with NAPALM
- Summary

Chapter 6. Understanding Python Dictionaries with NAPALM

- How NAPALM Organizes Data
- A Single Dictionary
- A Dictionary of Dictionaries
- A List of Dictionaries
- Sample Program Using All Three Types of Dictionaries
- Comparing the Three Types of Dictionaries
- Methods by Type of Data Structure
- Summary

Chapter 7. Iterating Through NAPALM Dictionaries

- Live or Simulated Data
- Working with .keys(), .values(), and .items() in NAPALM Dictionaries
- Using a for Loop with Dictionary Methods
- Determining the Type of Value
- Looping Through Key/Value Pairs with .items() and Processing Values with isinstance()
- Iterating Through a Dictionary of Dictionaries
- Iterating Through a List of Dictionaries
- Summary

Chapter 8. Configuring Devices with NAPALM

- A Quick Overview of NAPALM Methods

- Introducing Our Example Scenario

- Configuring a Device with
`load_merge_candidate()`

- Using the `cli()` method

- Using `load_replace_candidate()` to Replace the Configuration

- Displaying Running and Startup Configuration Files

- Next Step: Nornir

- Summary

Part 3: Nornir

Chapter 9. Introducing Nornir: A Pythonic Framework for Network Orchestration

- What Is Orchestration?

- How Does Nornir Compare to Netmiko and NAPALM?

- How Nornir Uses Netmiko and NAPALM

- Installing Nornir

- Basic Nornir Framework: Python and YAML

- Your First Nornir Program and YAML Files

- What's Next?

- Summary

Chapter 10. Using Nornir with Netmiko

- Installing Support for Netmiko: `nornir_netmiko`

- Using Nornir and `netmiko_send_command()`

Sending Configuration Commands with
`netmiko_send_config`
Summary

Chapter 11. Using Nornir with NAPALM

A Quick Review

Installing Support for NAPALM: `nornir_napalm`

Using Nornir and the `napalm_cli` Task

Using Nornir and the `napalm_get` Task

Using Nornir and the `napalm_configure` Task

Summary

Chapter 12. Inventory Management with Nornir

A Quick Overview with a Focus on Inventory

Where Inventory Data Comes From

A Note on Filtering

Inventory Management Core Architecture

Hosts

Groups

Effective Values After Inheritance

Using Group Data for Filtering

Defaults

Accessing Inventory Data

Inventory Plugins

Inventory Plugin Options

Summary

Part 4: What's Next

Chapter 13. What's Next

Why Network Automation Became Necessary

Netmiko, NAPALM, and Nornir in Context
Ansible, NETCONF, and RESTCONF: Expanding
the Automation Toolkit
Comparisons
Understanding YANG Models
Understanding APIs
Artificial Intelligence and Network Automation
Closing Thoughts

Appendixes

Appendix A. Python Virtual Environments

Appendix B. Understanding `expect_string` with
`send_command()`

Appendix C. Using Python Dictionaries as NAPALM
Outputs

Creating a Dictionary to Simulate Output

Appendix D. The Relationship Between Python
Dictionaries and JSON

Appendix E. Understanding Objects and Variables in
Python

Appendix F. Using a Recursive Function to Handle
Nested Dictionaries of Any Depth

Appendix G. Using Tabular Output

Appendix H. Using Public and Private Keys

Public Versus Private Keys: A Brief Overview
Basic Steps to Use SSH Keys with Netmiko

Why Use SSH Keys?

Public Key Configuration Example (Cisco IOS)

Appendix I. Netmiko-Supported Network Operating Systems

Icons Used in This Book



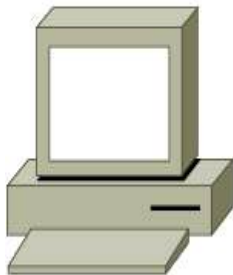
Router



Workgroup
Switch



Web
Browser



PC



Web
Server



File Server

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:

- *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.

Endorsements from Industry Leaders

Learning network automation can feel daunting, especially for engineers coming from a CLI-first background. These industry experts explain how this book removes those barriers, turning Python-based automation into something approachable, practical, and immediately useful for real networks.

"The deployment and operation of a complex environment has never been more difficult than now. Understanding and using valuable tools such as Netmiko, NAPALM, and Nornir will accelerate your journey to a fully automated environment."

This book, written by two world-class engineers, Rick Graziani and Adrian Iliesiu, will take you from beginner to a Python-driven automation practitioner in no time. I learned a lot from this book, and I know you will too."

**Shannon McFarland, Vice President Engineering,
Cisco Systems**

"Network automation is no longer optional—it's a foundational skill for operating modern networks at scale. Hands-On Cisco Automation with Python is a thoughtfully written, highly approachable guide that demystifies automation and makes it genuinely accessible for learners at every stage. Rick's learner-first approach, combined with

Adrian's deep, real-world automation industry experience, creates a rare balance of clarity and credibility, giving readers confidence that what they're learning is both practical and proven."

Kareem Iskander, Principal Engineer

"Network automation used to sound intimidating, but this book refactors that perception one script at a time. Rick and Adrian's step-by-step approach is so clear, even your smart toaster will want to change careers and start automating networks. With these skills, you won't just keep up, you'll be the one bringing ideas and innovation to your team."

Hank Preston, Distinguished Architect, CCIE, Cisco Systems

"An outstanding introduction to network automation. The authors present network programmability in a clear, methodical, and accessible manner, guiding the reader step by step from fundamentals to practical application. For those seeking a rigorous and reliable starting point, this book sets the standard."

Giuseppe Cinque, Principal Architect, Cisco Systems

"As a network engineer, learning and understanding network automation (especially Python) has transitioned from a nice-to-have to a requirement. Being able to write, test, and debug network automation solutions is key to continued growth and success of your professional career. Adrian and Rick have put together a top-notch book, outlining various frameworks and tools that will assist you in understanding and creating your own automations; from concept to idea, and I highly recommend this title to anyone looking to get started and grow their career."

Quinn Snyder, Engineering Technical Leader, Cisco Systems

"As soon as I saw an early draft, I wanted the guys to be done so I could have a finished copy!"

Comprehensive, organized, and useful for every networker—because every networker needs to have network automation skills.

You know, I would probably have bought this book sight unseen just because of how much I've enjoyed learning from Rick and Adrian over the years. Once I saw the idea, then the outline, and then the first pages, I was hooked. I'm looking forward to its release—I think it will be a popular book for most every network's bookcase."

Wendell Odom, Cisco Press Author, CCIE

"A lack of programming skills is a barrier to adopting network automation. Hands-On Cisco Automation with Python knocks that barrier down by translating the familiar CLI into easy-to-understand Python code. An engineer becoming a network automation expert can start their journey with this title."

Ethan Banks, Packet Pushers Founder

Introduction

Network automation can feel intimidating, especially when it is presented as a sudden shift away from the skills network engineers already have. This book was written to challenge that perception. The motivation behind this work is simple: Modern networks demand automation, but automation does not require abandoning Cisco IOS, operational experience, or practical intuition. The goal of this book is to help you take confident, manageable steps into automation by building directly on familiar workflows using Python, Netmiko, NAPALM, and Nornir. Through a progressive, hands-on approach—moving from CLI-based automation to structured data and scalable orchestration—this book aims to demystify automation, reduce the barrier to entry, and equip you with skills that matter in real networks today and into the future.

Who Should Read This Book?

This book is written for anyone who works with networks and wants a practical, approachable path into network automation. It is especially well suited for Cisco Networking Academy students and networking students who are just beginning their journey and want to understand how Python and automation connect to the Cisco IOS skills they are already learning. By starting with familiar CLI-based workflows, the book helps new learners build confidence

without requiring prior experience with APIs, data models, or advanced programming concepts.

The book is equally valuable for working network engineers and administrators who manage real production networks and recognize the growing need for automation, consistency, and scale. For experienced professionals who may feel pressed for time or unsure where to begin, this book provides a focused, hands-on approach that respects existing operational knowledge while introducing modern automation techniques in a clear and incremental way.

Whether you are a student preparing for your first networking role or a seasoned engineer adapting to the realities of modern networks, this book is designed to meet you where you are and help you move forward with confidence.

How This Book Is Organized

Chapter 1, “Introducing Netmiko, NAPALM, and Nornir,” introduces Netmiko, NAPALM, and Nornir and explains how these three Python libraries provide an accessible entry point into network automation and programmability. It outlines what each tool does, how these tools fit together, and the minimal Cisco IOS and Python knowledge needed to get started. This chapter sets the stage for the hands-on automation work that follows.

Chapter 2, “Getting Started with Netmiko,” introduces Netmiko as a practical first step into network automation, showing how familiar Cisco IOS commands can be executed programmatically over SSH using Python. The chapter walks through the basic structure of a Netmiko program, demonstrates key methods for sending commands and handling output, and introduces essential troubleshooting

techniques that build confidence before moving on to configuration automation.

Chapter 3, “Configuring Devices with Netmiko,” focuses on using Netmiko to configure network devices, moving beyond information retrieval to making reliable configuration changes. It introduces the preferred Netmiko methods for configuration automation, including applying multiple commands as lists and loading configurations from external files, while reinforcing best practices for efficient and repeatable device management.

Chapter 4, “Accessing Multiple Devices with Netmiko,” expands Netmiko automation from a single device to managing multiple devices using Python **for** loops. It demonstrates how to scale command execution and configuration across routers, introduces structured ways to store device connection details, and covers best practices for handling credentials, errors, and multiple SSH sessions in real-world automation scenarios.

Chapter 5, “Introducing NAPALM and Structured Data,” introduces NAPALM and the concept of structured data, explaining how network information can be retrieved in a consistent, vendor-agnostic format. The chapter contrasts NAPALM with Netmiko, introduces Python dictionaries as the foundation of structured data, and demonstrates how NAPALM methods return information that is easier to analyze, automate, and extend to API-driven workflows.

Chapter 6, “Understanding Python Dictionaries with NAPALM,” deepens your understanding of structured data by examining the three dictionary formats NAPALM uses: a single dictionary, a dictionary of dictionaries, and a list of dictionaries. Through practical examples, the chapter shows how different types of network data are organized and how

recognizing these patterns makes it easier to extract, iterate over, and automate device information.

Chapter 7, “Iterating Through NAPALM Dictionaries,”

focuses on iterating through the structured dictionaries returned by NAPALM, showing how to navigate simple, nested, and list-based data structures using Python **for** loops. The chapter emphasizes practical techniques for processing keys and values, handling different data types, and systematically extracting meaningful information from real-world network data.

Chapter 8, “Configuring Devices with NAPALM,”

discusses how to safely configure network devices using NAPALM’s staged configuration workflow. It explains how configuration changes can be loaded, previewed, committed, or rolled back using NAPALM’s configuration methods, and demonstrates both merging and replacing configurations while minimizing operational risk through comparison and backup mechanisms.

Chapter 9, “Introducing NORNIR: A Pythonic Framework for Network Orchestration,” introduces Nornir as a Python-based orchestration framework that brings together Netmiko and NAPALM to automate tasks across multiple devices at scale. It explains the concept of orchestration, shows how Nornir organizes device inventory using YAML files, and demonstrates how configuration and inventory data are combined to prepare for parallel, large-scale automation.

Chapter 10, “Using NORNIR with Netmiko,”

demonstrates how Nornir integrates with Netmiko to execute operational and configuration commands across multiple devices in parallel. It introduces the `nornir_netmiko` plugin, explains how tasks are run and how results are aggregated, and shows how shared and per-device

configurations can be applied efficiently using inventory data and custom task functions.

Chapter 11, “Using NORNIR with NAPALM,” introduces how Nornir integrates with NAPALM to provide vendor-neutral, structured network automation at scale. It demonstrates using Nornir with NAPALM tasks to run CLI commands, retrieve normalized operational data, and safely apply configuration changes, highlighting how inventory, parallel execution, and transactional workflows work together in real-world automation scenarios.

Chapter 12, “Inventory Management with NORNIR,” explores Nornir’s inventory system and explains how structured device data drives scalable automation. It introduces the hosts, groups, and defaults model; demonstrates how inheritance and filtering work; and surveys multiple inventory sources—including YAML files, Python dictionaries, NetBox, Ansible, spreadsheets, and modern sources of truth—to show how inventory remains consistent regardless of where the data originates.

Chapter 13, “What's Next,” looks beyond the tools covered in this book and places network automation in a broader operational and architectural context. It explains why automation became an operational necessity, clarifies the roles of SDN and intent-based networking, and introduces technologies such as Ansible, RESTCONF, NETCONF, APIs, YANG models, and AI, showing how the skills you’ve developed form a strong foundation for what comes next.

Chapter 1. Introducing Netmiko, NAPALM, and Nornir

The 3 *Ns*: Netmiko, NAPALM, and Nornir

Whether you're just starting to learn Cisco IOS and Python or have years of experience in networking, this book provides a straightforward introduction to the emerging world of network automation and programmability. This book is designed for *everyone*—from Cisco Networking Academy students beginning their networking journey to the seasoned professionals managing complex infrastructures. This book opens the door to a fun and fascinating realm of possibilities.

This book will introduce you to the fundamentals of network automation using three popular and powerful Python libraries: Netmiko, NAPALM, and Nornir. These tools provide a practical, accessible entry point for automating tasks, managing configurations, and streamlining network operations.

As you can see in [Figure 1-1](#), these three Python libraries enable you to immediately begin writing simple and beneficial Python code without needing prior knowledge of APIs (application programming interfaces), data models, or

structured data. Another key advantage is that they do not rely on any specific versions of Cisco IOS. All you need to get started with practical automation is SSH access to your Cisco devices.

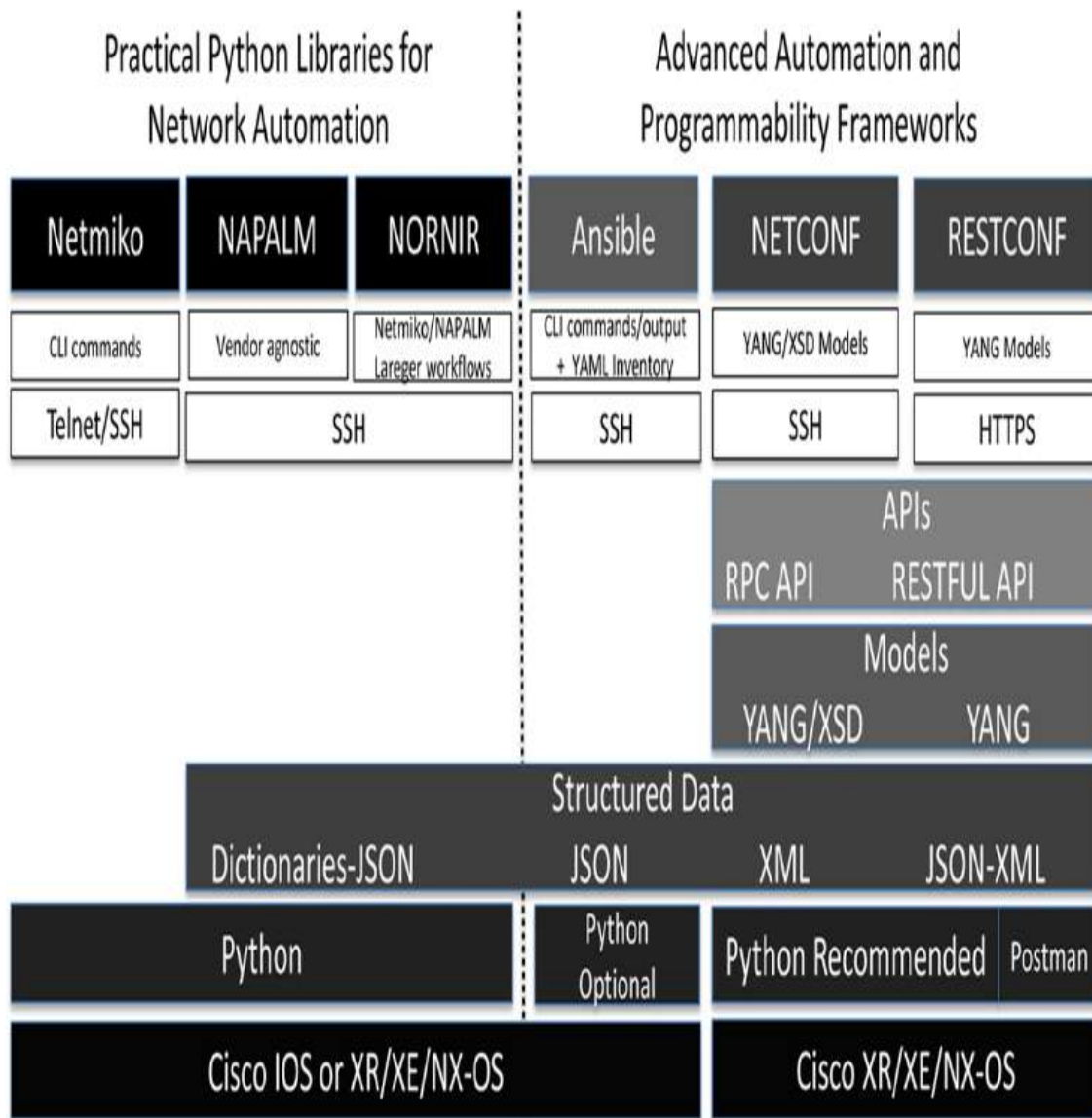


Figure 1-1 Network automation tools: From the CLI to API-driven frameworks

Part 1: Netmiko

In [Part 1](#) of this book, you will use your existing knowledge of the Cisco IOS CLI to quickly learn how to use Python for automation, starting with Netmiko. Netmiko simplifies the process of running commands across one or multiple devices, allowing you to retrieve IOS output and perform configuration tasks efficiently. Using it is a great way to build your confidence in network automation.

Netmiko is an open-source Python library designed to simplify SSH connections to network devices. It is licensed under the permissive MIT License, which means it can be freely used, modified, and distributed—even in commercial projects. Netmiko was originally developed by Kirk Byers and is now widely adopted and actively maintained by the network automation community. The source code is publicly available on GitHub, at github.com/ktbyers/netmiko. Built on top of the Paramiko SSH library, Netmiko provides a more network engineer-friendly interface, supporting a wide range of network device platforms across various vendors.

Note

In Python, library commands are technically referred to as *methods*. However, we will at times use the term *command* instead of *method* for those who are new to this concept. The terminology is less important than helping you get started quickly and effectively.

Part 2: NAPALM

In [Part 2](#) of the book, you will learn about NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support), a library that uses vendor-agnostic

commands to retrieve device output as structured data. *Vendor agnostic* means that the same commands can be used on a variety of vendor devices. This is especially beneficial in multivendor environments.

In case you are new to structured data, [Part 2](#) introduces Python dictionaries, demonstrating their flexibility and advantages. You will see how the organized structure of the data allows the information to be analyzed and displayed in a way that best suits your needs. You will also learn how NAPALM simplifies configuration tasks and is a versatile tool for both retrieving and managing network data. So, if you are new to Python dictionaries, don't worry: It will all be explained to you.

NAPALM is an open-source Python library used for network device configuration and state retrieval across multiple vendor platforms. It is licensed under the Apache 2.0 License, which allows for free use, modification, and distribution, even in commercial applications. NAPALM, which was originally developed by a community of engineers at napalm-automation.net, is now part of the broader network automation ecosystem maintained by the community, and organizations like Network to Code have contributed to it. The source code is available on GitHub at github.com/napalm-automation/napalm.

Part 3: Nornir

Finally, in [Part 3](#) we will dive into Nornir, a powerful library that integrates seamlessly with both Netmiko and NAPALM. Nornir takes automation to the next level by enabling advanced functions such as inventory management and the use of plugins to extend its capabilities. You will learn how Nornir simplifies management of large-scale networks and coordination of complex tasks across multiple devices,

providing a robust foundation for scalable and efficient network automation.

Nornir is an open-source Python automation framework designed specifically for network automation at scale. It is licensed under the Apache 2.0 License, which permits free use, modification, and distribution, including for commercial use. The project is community driven and maintained on GitHub at github.com/nornir-automation/nornir.

Part 4: What's Next?

Part 4 of this book looks ahead to what's next, including an introduction to APIs in networking and an exploration of Cisco DevNet and network automation ecosystems. This section provides a glimpse into advanced tools and concepts, helping you prepare for the broader possibilities in the world of network programmability.

The three libraries discussed in this book—Netmiko, NAPALM, and Nornir—are all widely used and well established in the industry. In **Part 4** you will see how these tools can also serve as a bridge to other technologies, such as RESTCONF and NETCONF. Both RESTCONF and NETCONF use APIs much as NAPALM and Nornir do, and rely on structured data for network automation and programmability.

By being introduced to structured data through using NAPALM and Nornir libraries, you will become more comfortable using other technologies, like RESTCONF and NETCONF, as you'll already have experience working with structured information.

Automation and Programmability for All Levels

One of the key advantages of the Netmiko, NAPALM, or Nornir libraries is their accessibility. They are not just designed for experienced network professionals but are also perfect for someone just beginning their journey with Cisco IOS and Python. As your knowledge of Cisco IOS and Python grows, you will be able to take better advantage of these libraries for network automation and programmability. But as you will see, you do not need to be an expert in Cisco IOS or Python to get started.

Which one of these libraries—Netmiko, NAPALM, or Nornir—is the best? Is Ansible, RESTCONF, NETCONF, or something else better? Well, which is better—Windows, Linux, or macOS? iPhone or Android? The best automation tool is the one that you are most comfortable using to accomplish your task. There is always something new and shiny, and it is important to continue to stay informed on what else is available and other ways to improve what you are doing. As you increase your learning, you may find that automation can help you with something you hadn't thought of before.

How Much Cisco IOS and Python Do I Need to Know?

This book assumes only a basic understanding of Cisco IOS and Python, making it accessible to a wide audience. The Cisco IOS commands used throughout are fundamental commands that should be familiar to those preparing for the CCNA exam and even to students taking their first Cisco Networking Academy introduction to networking course. This book also assumes familiarity with using SSH to access Cisco IOS devices.

This book also assumes that you know how to create and run a Python program and have a basic understanding of programming fundamentals, such as *variables*, the *print*

function, if-elif, for loops, and lists. While some familiarity with dictionaries is helpful, this book provides a thorough introduction to working with them to ensure that you have the understanding you need. If you already have more advanced Python knowledge, you will quickly understand the potential of these libraries and be able to take even more advantage of their capabilities.

This book focuses on providing just the essentials you need to quickly understand and start using the Netmiko, NAPALM, and Nornir Python libraries. Some Python features that are not specific to these libraries are valuable in production environments (such as using public/private keys for faster authentication), and we've included them in appendixes.

At times, we might say "for those more familiar with Python" or something similar. That text is designed to provide a more precise and technical explanation of the code, using terms and concepts that may be new to you. If you're unfamiliar with these ideas, don't worry: They are not essential for understanding or using the code effectively. Feel free to skip them and focus on the main explanations.

Also don't worry if Python is new to you: This book keeps the code simple and straightforward. No matter your level of Python knowledge, you can use this book to enter the world of automation and programmability.

What Do I Need to Get Started?

All you need to get started with this book is a computer that has Python installed and that can SSH into at least one Cisco IOS device. ***Any router or switch model, no matter how old, and any relatively current version of Cisco IOS will work!*** The output shown in this book is from Cisco IOS XE 16. Your output may differ if you are using a different IOS version.

All you need is SSH access to one device, though several devices would be even better. Having multiple IOS devices will allow you to see how a Python program can be used to access and perform the same tasks on different devices. However, if you have just a single Cisco IOS device, you will be able still do everything in this book.

Note

If you do not have access to physical Cisco IOS routers or switches, see [Appendix A, “Python Virtual Environments,”](#) for other possible options or using generative artificial intelligence (Gen AI) tools like ChatGPT, which is discussed later in this chapter.

In case you are unsure how to configure SSH on Cisco IOS, [Example 1-1](#) shows an example.

Example 1-1 *Configuring SSH*

```
Router(config)# username admin password cisco
Router(config)# ip domain-name SSH-KEY.com

Router(config)# crypto key generate rsa general-keys modulus 1024

Router(config)# line vty 0 15
Router(config-line)# login local
Router(config-line)# transport input ssh
Router(config-line)# exit
```

Here is a brief explanation of each command in [Example 1-1](#):

- Router(config)# **username admin password cisco**: This command creates a user named **admin** with the plaintext password **cisco** to allow SSH and console login

access. Although this works, it's not very secure because the password is stored in plaintext in the running configuration. A better approach would be to use **username admin secret cisco**, which hashes the password for improved security, but for simplicity, we are using the **password** parameter in this example.

- Router(config)# **ip domain-name SSH-KEY.com**: This command sets the domain name of the router to **SSH-KEY.com**. This is required for generating RSA keys, and any domain name can be used.
- Router(config)# **crypto key generate rsa general-keys modulus 1024**: This command generates a pair of RSA keys with a modulus of 1024 bits for secure SSH communication.
- Router(config)# **line vty 0 15**: This command selects the virtual terminal lines (VTY) 0 through 15 for remote access configuration.
- Router(config-line)# **login local**: This command configures the VTY lines to use local authentication, relying on the username and password defined on the router.
- Router(config-line)# **transport input ssh**: This command limits remote access to the router through VTY lines to only use SSH for secure communication. Another option is to use **transport input all**, which permits both SSH and Telnet.

Verify that you can SSH remotely into the Cisco IOS device before you go any further.

Using AI as an Alternative to Physical Equipment

While having access to physical Cisco devices is ideal for hands-on practice, it's not always possible for every learner. Fortunately, generative artificial intelligence (Gen AI) tools like ChatGPT can simulate router and switch behavior in a conversational environment. By asking ChatGPT to “act like a Cisco router,” you can practice entering commands, troubleshoot configurations, and receive contextual tips—all mimicking the experience of working on real hardware. While not a substitute for actual IOS behavior, this approach provides a valuable learning experience, and it is especially handy when equipment or simulation tools aren't available.

Try entering the following prompt into ChatGPT:

Act like a Cisco router. I will connect to you using SSH. Your job is to respond exactly as a router would in the terminal. If I enter incorrect commands, show appropriate IOS error messages. Outside the terminal window, you may give me tips or hints to help me succeed.

Here's the setup:

- I'm using a Linux PC running Python 3.
- My IP address is 192.168.1.10/25.
- I'm directly connected to you via your GigabitEthernet0/0/0 interface.
- Your hostname is **R1**.

This is the current output of your **show ip interface brief** command:

```
R1(config)# username admin password cisco
R1(config)# ip domain-name SSH-KEY.com
R1(config)# crypto key generate rsa general-keys modulus
R1(config)# line vty 0 15
```

```
R1(config-line)# login local
R1(config-line)# transport input ssh
R1(config-line)# exit
```

Please open a Linux-style terminal window so I can test connectivity. I should be able to:

- ping 192.168.1.1 (your G0/0/0 interface)
- ssh admin@192.168.1.1

When interacting as a terminal, I will preface each command with 'Terminal:', so you know that I am entering a command at the terminal prompt.

For example:

- Terminal: ping 192.168.1.1
- Terminal: ssh admin@192.168.1.1
- Terminal: python3 1_netmiko_show.py

Once this prompt is entered, ChatGPT takes on the role of both the terminal and the router, responding to your inputs as if you were connected to real hardware. You can then practice issuing commands, observing output, and experimenting safely—reinforcing concepts before applying them to actual devices.

As generative AI tools evolve, the way we interact with them will likely become even more intuitive and interactive. The method you use may vary slightly from what we've shown here, depending on the tool you're using, but the core idea remains the same: You can leverage AI to reinforce your skills and gain confidence in network automation.

Summary

In this chapter, we have discussed how this book serves as a gateway to network automation and programmability, introducing three powerful Python libraries: Netmiko, NAPALM, and Nornir. These libraries provide practical, accessible tools for automating tasks and managing Cisco IOS and other vendor networking devices. You will be able to use them regardless of your prior experience. Some foundational knowledge of Cisco IOS and Python is required to get started, but this book is designed for learners at all levels. Now, let's dive into the exciting world of network automation, starting with Netmiko!

Part 1: Netmiko

Chapter 2. Getting Started with Netmiko

Do not underestimate the utility of the SSH Netmiko library, as it's proven useful in providing a smooth transition from traditional network CLI management to network automation, and it's heavily used and developed.

*From the book *Network Programmability & Automation*, 2nd edition, O'Reilly*

As networks grow in size and complexity, traditional methods of managing devices through the CLI are proving insufficient. Netmiko is a Python library that bridges the gap between manual network device configuration and full-scale automation. Designed to work seamlessly with networking devices, Netmiko simplifies routine tasks by enabling engineers to programmatically interact with devices over SSH, using familiar CLI commands wrapped in the flexibility and power of Python.

This chapter introduces network automation with Netmiko. Whether you're a seasoned network engineer or just getting started, you'll discover how easy it is to transition from manual CLI tasks to automated solutions. By the end of this chapter, you'll have the foundational knowledge you need to

build scripts that connect to devices, execute commands, and retrieve output.

Why Start with Netmiko?

Why start with Netmiko? Because it's easy! Netmiko is a Python library that offers an easy introduction to network programmability, using the same Cisco IOS commands you would use from a command line prompt. If you know a little IOS and you know a little Python, you can start using Netmiko quickly and easily.

And Netmiko isn't just for Cisco IOS. Netmiko can also be used with just about any network operating system, as long as you have SSH access to the device.

Netmiko is a Python SSH library that was developed from another library, Paramiko, which allows users to access and manage remote devices, servers, and other systems over SSH. However, Netmiko was specifically designed for networking devices in order to simplify the automation of common networking tasks.

Kirk Byers is the creator of the Netmiko Python library. He is also one of the core maintainers of the NAPALM library.

A First Look at a Netmiko Program

Before we discuss how to download the Netmiko library or the basic framework of a Netmiko program, take a look at the Netmiko code in [Example 2-1](#). Just by looking at the parts in **bold**, you can probably understand what this program does.

Example 2-1 *ex2-1_my_first_netmiko_program.py*

```
# My first Netmiko program
import netmiko

connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
                                    secret='spot')

print(connection.send_command('show ip interface brief'))

connection.disconnect()
```

Simply put, this program connects to the Cisco IOS device with IP address 192.168.1.1 and sends the **show ip interface brief** command. [Example 2-2](#) shows the output. Yes, it is that easy!

Example 2-2 Output from [Example 2-1](#)

```
MyPrompt% python3 ex2-1_my_first_netmiko_program.py

Interface                IP-Address      OK? Method Status
GigabitEthernet0/0/0     192.168.1.1     YES NVRAM  up
GigabitEthernet0/0/1     192.168.2.1     YES NVRAM  up
GigabitEthernet0/0/2     unassigned      YES NVRAM  administrative
GigabitEthernet0         unassigned      YES NVRAM  administrative
MyPrompt%
```

Note

In this book, we show the **python3** command to ensure that you are running Python version 3. While some systems may allow you to use the **python**

command instead, especially as Python 2 is phased out, using **python3** helps avoid confusion. If you're unsure which version is mapped to Python on your system, run **python --version** or **python3 --version**.

Notice in the output in [Example 2-2](#) that the prompt and Cisco IOS command are not displayed before the output of **send_command()**. Later in this chapter, in the section *"Sending the Output to a Variable,"* you will see how to include one or both of these.

Note

Remember that we are assuming that you have Python installed and SSH connectivity to the device.

Now that you have seen how easy this is going to be, you're ready to download the Python Netmiko library.

Installing the Netmiko Library

To install Netmiko, use either the **pip** or **pip3** command. **pip** is the Python package installer, and you use it to download and manage Python libraries and dependencies. It simplifies the process of installing, upgrading, and uninstalling packages from the Python Package Index (PyPI) and other sources.

To install Netmiko, from your Windows Command Prompt or PowerShell or from a macOS or Linux terminal, type this command:

```
MyPrompt% pip3 install netmiko
```

If you encounter errors during installation, ensure that Python and pip are updated and run the command as an

administrator (for example, use **sudo pip3 install netmiko** on macOS/Linux or run the Windows terminal as an administrator).

The Basic Netmiko Framework

Let's look more closely at what is happening in the program shown in [Example 2-1](#) and the results shown in [Example 2-2](#). Every Netmiko program has the same basic framework, which includes these steps:

1. Import the Netmiko library.
2. Create the SSH connection.
3. Implement Cisco IOS commands.
4. Close the connection.

This framework is part of each Netmiko program and gives you the ability to leverage Python to access and manage one or more devices automatically. [Example 2-3](#) shows this basic framework for every Netmiko program. It is the same code shown in [Example 2-1](#) but with additional comments.

Note

At times we refer to parts of the Python code as classes, objects, and methods. However, you can understand the explanations and the code in these programs without being familiar with classes, objects, and methods. If you want to learn more about classes, objects, and methods, you can read the optional section later in this chapter.

Example 2-3 *ex2-3_netmiko_framework.py*

```
# Step 1: Import the Netmiko Library
# The Netmiko library provides SSH connectivity to network devices
```

```

automation.
import netmiko

# Step 2: Establish an SSH connection to the device
# The ConnectHandler method creates a connection using the provided
parameters.

connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
                                    secret='spot')

# Step 3: Execute an IOS command and display the output
# The send_command() method sends the 'show ip interface brief'
device.
print ( connection.send_command ( 'show ip interface brief' ) )

# Step 4: Close the SSH connection
# The disconnect() method gracefully terminates the connection.
connection.disconnect()

```

The **print** statement in [Example 2-3](#) includes additional spaces for readability. These spaces make the different parts of the code in this Netmiko command easier to distinguish:

With spaces: `print ( connection.send_command ( 'show ip interface brief' ) )`

Without spaces:

`print(connection.send_command('show ip interface brief'))`

Example 2-4 shows the output from running to code in Example 2-3. The **show ip interface brief** command works just like it would directly from the CLI.

Example 2-4 Output from Example 2-3

```
MyPrompt% python3 ex2-3_netmiko_framework.py
Interface          IP-Address      OK? Method Status
GigabitEthernet0/0/0 192.168.1.1    YES NVRAM  up
GigabitEthernet0/0/1 192.168.2.1    YES NVRAM  up
GigabitEthernet0/0/2 unassigned      YES NVRAM  administrative
GigabitEthernet0     unassigned      YES NVRAM  administrative
MyPrompt%
```

Let's look at the steps in a little more detail.

Step 1: Import the Netmiko Library

The **import netmiko** statement imports the Netmiko library, which provides tools to simplify the process of managing network devices over SSH. This is required to use the Netmiko commands, also known as Python *methods*, in your program.

Step 2: Establish an SSH Connection to the Device

The **connection** variable (also known as a Python *object*) contains the result of the **netmiko.ConnectHandler** command. The **netmiko.ConnectHandler()** command creates an SSH connection to the network device. This connection allows you to send commands to the device and receive responses.

ConnectHandler creates an SSH connection to the network device using the following parameters:

- **ip:** The device's IP address
- **device_type:** The type of network device (for example, **cisco_ios** for Cisco IOS devices or **cisco_ios_telnet** for Telnet access)
- **username** and **password:** The SSH credentials used for authentication
- **secret:** The privileged EXEC password

The **secret** parameter is necessary only if the program needs to execute commands that require privileged EXEC mode access.

Step 3: Execute an IOS Command and Display the Output

Figure 2-1 shows a typical Netmiko combination of the **connection** variable (or *object*) and Netmiko command (or *method*) used to implement a Cisco IOS command.

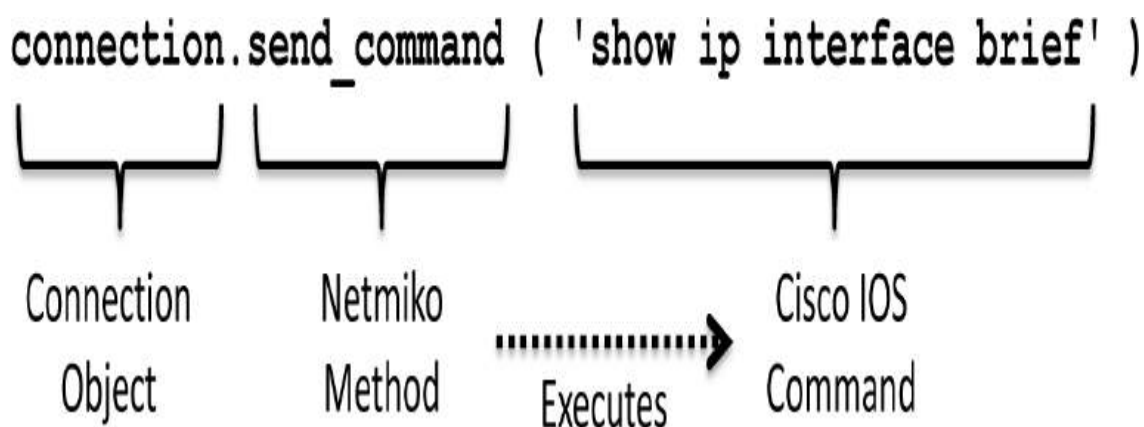


Figure 2-1 Example of a Netmiko connection and method

In the example shown in [Figure 2-1](#), **connection.send_command ('show ip interface brief')** is the command that is executed on the device and retrieves the output as a string. The output includes the same information you would see if you typed the command directly on the device's CLI. These are the parts of the command:

- **connection:** This is the variable created earlier (using the **netmiko.ConnectHandler**) that holds the SSH connection to your network device. It will be used before each Netmiko method. **connection** is simply the name of the variable used with ConnectHandler and can be replaced with any valid Python variable name that you choose.
- **.send_command():** Netmiko provides this method to send a single Cisco IOS command to the connected device. You can think of methods as a command that you can use because you imported the Netmiko library. In this case, the IOS command being sent is '**show ip interface brief**'.
- '**show ip interface brief**': This is a common Cisco IOS command that is used to display a summary of the device's interfaces. In this case, it is used as a Python string that the **send_command()** method refers to.

[Figure 2-2](#) shows how the Python **print** function is prepended to print the results of the **show ip interface brief** command—the string output—to the screen. This function displays the output of the **show ip interface brief** command exactly as it would appear using the CLI.

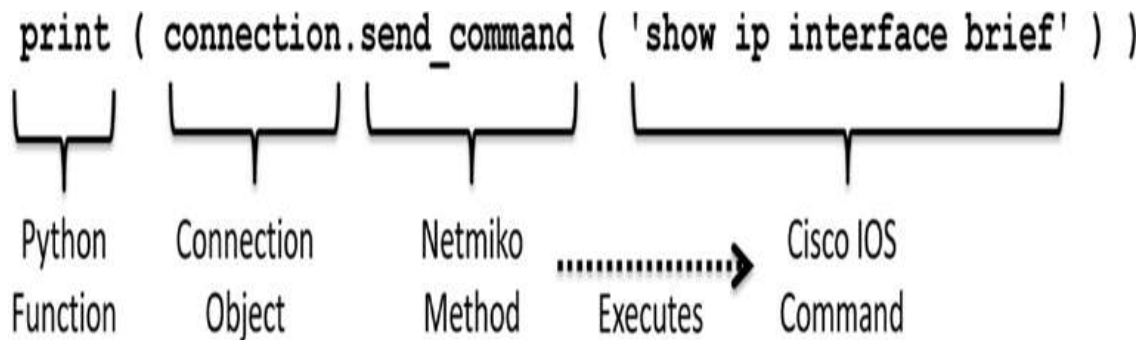


Figure 2-2 Prepending the Netmiko **`send_command()`** method with the Python **`print`** function

The **`send_command()`** Method

In [Example 2-3](#), the Netmiko method **`send_command()`** sends a command to the connected device. The **`send_command()`** method can both retrieve information from devices and also configure devices. ([Chapter 3](#), “[Configuring Devices with Netmiko](#),” discusses device configuration.)

If you are new to Python, you might not be familiar with methods. You can think of a *method* as being similar to a command or function that was imported from a library—in this case, the Netmiko library. A *method* is like a command that you can give to an object to make it do something, such as sending a command to a network device. For example, the **`send_command()`** method tells the **`connection`** object to run a specific command on the device and return the result. The **`connection`** object is the device you are remotely connected to with SSH.

Don't worry. All of this will become more familiar to you as you see the Netmiko connection object and methods in more examples. Also, we've provided an optional section on classes, objects, instances, and methods in this chapter—and you'll get to it very soon.

Step 4: Close the SSH Connection

The **disconnect** method prefaced with the **connection** variable terminates the SSH session gracefully, releasing resources and preventing potential issues such as connection timeouts or orphaned sessions. Using **disconnect()** to close connections when a program completes is a best practice for resource management.

Optional: Exploring Python Classes, Objects, Instances, and Methods

You do not have to understand object-oriented programming (OOP) to use Netmiko or to understand the code in this book. But if you are new to Python or programming in general, you might be interested in what people are talking about when they use terms like *class*, *object*, and *method*. In that case, you might find this section useful. Because you can understand the material in this book without understanding the information in this section, you can consider this section optional.

ConnectHandler is a Netmiko *class*. This simply means it's a template that allows you to create variables (such as **connection**) that are set up to interact with a network device. When you create a variable or an object (such as **connection**) using the Netmiko **ConnectHandler** class, you can access special Netmiko commands, called *methods*, that are built into the class. For example, one of these methods is **send_command()**, which lets you send commands to a device and get the output back. A method is like a command provided by a class to perform specific actions—for example, **send_command()** and **send_command_expect()**.

You access a method by appending the method name to the variable or object using dot notation. For example, **connection.send_command('command')** allows the **connection** object to execute commands and interact with the network device seamlessly.

Figure 2-3 shows the relationship between the Netmiko **ConnectHandler** and methods such as **send_command()**. Here's an explanation of what the figure shows:

- **ConnectHandler** is a Netmiko class with embedded commands called *methods* that you can use.
- The **ConnectHandler** class requires certain *parameters*, such as the IP address of the device, the network operating system, and SSH credentials.
- **connection** is the name of a variable or object, chosen by the programmer, that is an object of this class, which means it has access to the methods, or commands, associated with the **ConnectHandler** class.
- **send_command()** is a Netmiko method that belongs to the **ConnectHandler** class and is accessed through the **connection** object (a Python variable). It requires a Cisco IOS command, passed as a string, which Netmiko sends to the device and returns the output.

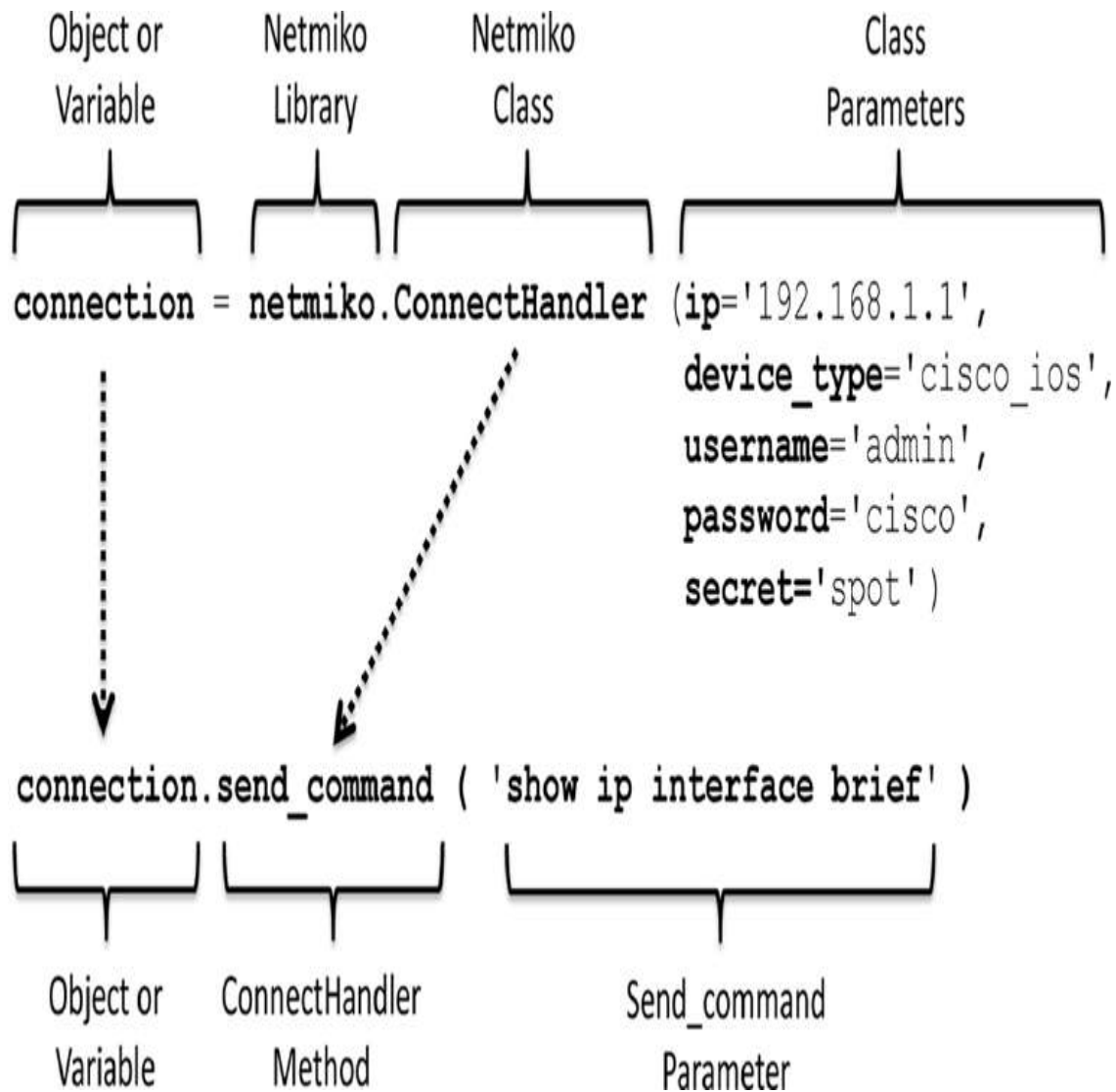


Figure 2-3 Relationship between classes, objects, and methods

If you are familiar with OOP, you can think of it this way: Everything in Python is an object, including all variables, no matter their data type (string, integer, Boolean, and so on). Functions, including built-in functions like **print()**, are also objects.

The **ConnectHandler** class creates an SSH connection to the network device by using the parameters provided. The **connection** variable is an object—specifically an *instance* of the **ConnectHandler** class. This means it is an object

created using the **ConnectHandler** class, which is part of the Netmiko library. As an instance, **connection** has access to all the methods and attributes defined in **ConnectHandler**, such as **send_command()** and **disconnect()**.

Your Turn

Now that you've seen how easy it is to use Netmiko to interact with a Cisco device, take some time to experiment with commands on your own. Try using other common Cisco IOS commands, such as **show version** to display device information or **show arp** to retrieve the current configuration, as shown in [Example 2-5](#).

Before each **send_command()** statement in [Example 2-5](#), we've added a **print()** statement to indicate which command generated the output.

Example 2-5 *ex2-5_send_command.py*

```
# Import the Netmiko Library
import netmiko

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco',
                                     secret='spot')

# Using the connection to send IOS commands
print('\nResults from: show ip interface brief')
print(connection.send_command('show ip interface brief'))
```

```

print('\nResults from: show version')
print(connection.send_command('show version'))

print('\nResults from: show arp')
print(connection.send_command('show arp'))

# Close the SSH connection
connection.disconnect()

```

These commands work just as they would in the CLI, but now you can execute them through Python, as shown in [Example 2-6](#). Trying out different commands on your own is a great way to explore Netmiko and become more comfortable with both Python scripting and network automation. Don't be afraid to test different commands. If you are on a production network, just ensure that they are not disruptive to your network devices. The more you experiment, the more confident you'll become!

Example 2-6 Output from [Example 2-5](#)

```
MyPrompt% python3 ex2-5_send_command.py
```

Results from: show ip interface brief

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.1.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.1	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

Results from: show version

```

Cisco IOS XE Software, Version 16.06.03
Cisco IOS Software [Everest], ISR Software (X86_64_LINUX_IOSD-UI

```

```
Version 16.6.3, RELEASE SOFTWARE (fc8)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2018 by Cisco Systems, Inc.
Compiled Wed 28-Feb-18 23:54 by mcpre
```

```
<output omitted for brevity>
```

```
cisco ISR4331/K9 (1RU) processor with 1796073K/6147K bytes of me
Processor board ID FLM2229W1R6
3 Gigabit Ethernet interfaces
32768K bytes of non-volatile configuration memory.
4194304K bytes of physical memory.
3125247K bytes of flash memory at bootflash:.
0K bytes of WebUI ODM Files at webui:.
```

```
Configuration register is 0x2102
```

Results from: show arp

Protocol	Address	Age (min)	Hardware Addr	Type	Int
Internet	192.168.1.1	-	dcf7.1994.a830	ARPA	Gi0
Internet	192.168.1.10	0	00e0.4c68.05b6	ARPA	Gi0
Internet	192.168.2.1	-	dcf7.1994.a831	ARPA	Gi0

MyPrompt%

Notice that all these commands only require user EXEC mode access. You will see commands that require privileged EXEC mode access later in this chapter.

Sending Command Output to a Variable

So far, we've used the Python **print()** function directly with the **connection** object to display the output of a command. We have also used the **print()** function to indicate the command that was used to create the output.

Now, let's try something different and store the results of the **send_command()** method in a variable. This allows you to use the output later or at different points in the program, making it more versatile.

*Example 2-7 shows how we used **send_command()** in the Example 2-5 to display the results directly from the **print()** function. It also shows how to store the results from **send_command()** to the variable **cmd_output** and then use the **print(cmd_output)** function to display the contents of the variable.*

Example 2-7 ex2-7_send_command_variable.py

```
import netmiko

connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
                                    secret='spot')

# Example of directly printing the output of the send_command()
print('\nDirectly from the print function:')
print(connection.send_command('show ip interface brief'))

# Example of storing the output of the send_command() method in
# The variable cmd_output stores the command result, which is then
print('\nPrinting from contents from the variable:')
cmd_output = connection.send_command('show ip interface brief')
```

```
print(cmd_output)

connection.disconnect()
```

Both approaches—printing directly from the **connection** object and printing the stored variable **cmd_output**—give you the same output, as shown in [Example 2-8](#).

Example 2-8 Output from [Example 2-7](#)

```
MyPrompt% python3 ex2-7_send_command_variable.py

Directly from the print function:
Interface                IP-Address      OK? Method Status
GigabitEthernet0/0/0    192.168.1.1     YES NVRAM  up
GigabitEthernet0/0/1    192.168.2.1     YES NVRAM  up
GigabitEthernet0/0/2    unassigned      YES NVRAM  administrative
GigabitEthernet0        unassigned      YES NVRAM  administrative

Printing from contents from the variable:
Interface                IP-Address      OK? Method Status
GigabitEthernet0/0/0    192.168.1.1     YES NVRAM  up
GigabitEthernet0/0/1    192.168.2.1     YES NVRAM  up
GigabitEthernet0/0/2    unassigned      YES NVRAM  administrative
GigabitEthernet0        unassigned      YES NVRAM  administrative
MyPrompt%
```

The **send_command()** method executes the specified command on the device and retrieves the output, but if it is not paired with the **print()** function or assigned to a variable, the output is discarded. While this can be useful for configuration commands where output is not typically

needed (for example, **connection.send_command('username alice password cisco')**), it's important to note that **send_command()** is generally intended for retrieving output. For configuration commands, the **send_config_set()** method is often preferred, as it is specifically designed for executing multiple configuration commands and ensuring proper handling of the device's configuration mode. In [Chapter 3](#), we discuss how to configure devices with Netmiko, including by using **send_config_set()**.

Including the IOS Command in the Output

In the previous section, we looked at how to store command output in the variable **cmd_output**. However, the output of **send_command()** does not include the actual IOS command that generated it. By default, Netmiko removes the command from the output to streamline the results. While this is often helpful and we can include a separate **print()** command to display the command ourselves, there are times when we would like Netmiko to do this for us.

To include the IOS command in the output, Netmiko provides the **strip_command** parameter. We can use **strip_command=False** to have the method prepend the executed command to the output. [Example 2-9](#) shows how it works.

Example 2-9 *ex2-9_send_command_showCommand.py*

```
import netmiko

connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
```

```

secret='spot')

# Using the send_command() method with strip_command=False
# Setting strip_command=False ensures that the executed command
# ('show ip interface brief') is included in the output,
# making it clear what command generated the results.
print('\nPrinting from contents from the variable (including the
cmd_output = connection.send_command('show ip interface brief',
                                     strip_command=False)

print(cmd_output)

connection.disconnect()

```

This example uses **strip_command=False** in the **send_command()** method. As a result, the command **show ip interface brief** is included at the very beginning of the output, as shown in [Example 2-10](#). This provides better context for someone who is reviewing the output.

Example 2-10 Output from [Example 2-9](#)

```

MyPrompt% python3 ex2-9_send_command_showCommand.py

Printing from contents from the variable:
show ip interface brief
Interface                IP-Address      OK? Method Status
GigabitEthernet0/0/0    192.168.1.1     YES NVRAM  up
GigabitEthernet0/0/1    192.168.2.1     YES NVRAM  up
GigabitEthernet0/0/2    unassigned      YES NVRAM  administrative
GigabitEthernet0        unassigned      YES NVRAM  administrative
MyPrompt%

```



```
# Using find_prompt() to display the device's current prompt
# (e.g., "Router-R1>")
# The value of cmd_output is displayed on a separate line
print(connection.find_prompt())
print(cmd_output)
print('\n')

# Using find_prompt() and send_command() together
# This demonstrates how to display the device prompt followed by
# command output, making the result look more like the native CLI
print(connection.find_prompt(), cmd_output)

connection.disconnect()
```

The first part of [Example 2-11](#) displays the device prompt returned by **find_prompt()** on one line and the command output stored in the **cmd_output** variable on the next line. This approach separates the two visually, making it easy to distinguish between the prompt and the command output.

As shown in the second part of [Example 2-11](#), to re-create how a command might appear in the CLI, you can print both the prompt and the command output on the same line. You do this by combining the results of **find_prompt()** and **cmd_output** into a single **print** statement.

[Example 2-12](#) shows the results of both approaches.

Example 2-12 *Output from [Example 2-11](#)*

```
MyPrompt% python3 ex2-11_send_command_showPromptCommand.py

Router-R1>
show ip interface brief
```

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.1.1	YES	manual	up
GigabitEthernet0/0/1	192.168.2.1	YES	manual	up
GigabitEthernet0/0/2	unassigned	YES	unset	administrative
GigabitEthernet0	unassigned	YES	unset	administrative

Router-R1> show ip interface brief

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.1.1	YES	manual	up
GigabitEthernet0/0/1	192.168.2.1	YES	manual	up
GigabitEthernet0/0/2	unassigned	YES	unset	administrative
GigabitEthernet0	unassigned	YES	unset	administrative

MyPrompt%

Storing the output of the **send_command()** method in a variable like **cmd_output** and using the **find_prompt()** method provides flexibility in how and when you present the results, allowing you to customize the display format to best suit your needs.

The **send_command_expect()** Method

The Netmiko **send_command_expect()** method, shown in [Example 2-13](#), allows you to use commands that require user confirmation or interaction, such as **copy running-config startup-config**, which requires the user to confirm the destination filename by pressing Enter. Unlike **send_command()**, the **send_command_expect()** method waits for a specific string, defined by the **expect_string** parameter, to appear in the command's output before proceeding.

Example 2-13 *ex2-13_send_command_expect.py*

```
import netmiko

# The secret parameter is required for privileged EXEC mode access
# It provides the password that will be used with the enable command
# to enter privileged mode.
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
                                    secret='spot')

# Ensure privileged EXEC mode
# The enable() method is necessary to enter privileged EXEC mode
# Commands like 'copy running-config startup-config' require this
# to execute successfully.
connection.enable()

# Using send_command_expect() to handle interactive prompts
# The expect_string parameter specifies the prompt Netmiko
# should wait for before proceeding.
print(connection.send_command_expect('copy running-config startup-config',
                                     expect_string='Destination filename [startup-config]?'))

print('Success!')

connection.disconnect()
```

[Example 2-14](#) shows an example of using the command **copy running-config startup-config** at the CLI. The user must press Enter after the '**Destination filename [startup-config]?**' prompt.

Example 2-14 *copy running-config startup-config*


```
Router-R1>enable
Password: spot
Router-R1#copy running-config startup-config
Destination filename [startup-config]? <user presses Enter>
Building configuration...

[OK]
Router-R1#
```

The **copy running-config startup-config** command requires privileged EXEC mode access. This means you have to use the **secret** parameter with the enable secret password **spot** in **ConnectHandler**. The **connection.enable()** command ensures that the session enters privileged EXEC mode (the Router# prompt), which is required for commands like **copy running-config startup-config**.

If the **secret** parameter is provided in **ConnectHandler**, Netmiko attempts to elevate to privileged EXEC mode automatically. However, this might fail due to device-specific configurations or an unusual prompt format. To avoid inconsistencies across different devices, explicitly include **connection.enable()** in your scripts whenever you need privileged EXEC mode.

If you are using the **expect_string** parameter in automation, it is important to note that although **expect_string** tells Netmiko to wait for a specific prompt before continuing, it does *not* automatically send input, like pressing Enter. In the case of **copy running-config startup-config**, IOS expects the user to press Enter to confirm the default filename. If this step is skipped, the configuration may not be saved. Netmiko does not send that newline by default.

To have Netmiko automatically press the Enter key to confirm the filename, you can use **send_command()**, as shown here:

```
# Send the copy command and wait for the prompt asking for confirm
connection.send_command_expect('copy running-config startup-config
                                expect_string='Destination filename
# Send a newline to simulate pressing Enter and accept the default
connection.send_command("\n", expect_string="#")
```

This command simulates pressing the Enter key and waits until the router returns to privileged EXEC mode to ensure that the configuration is saved properly.

If you prefer to allow the user to manually interact with the prompt (such as in extended ping scenarios), you can have the program pause at that point, or you can script out more complex input sequences step by step.

[Example 2-15](#) shows the program from [Example 2-13](#) executing the **copy running-config startup-config** command. With the **send_command_expect()** method, **expect_string** is set to '**Destination filename**', which matches the prompt Netmiko will encounter during the execution of the **copy running-config startup-config** command. Once this string is detected, the method considers the command complete and takes the appropriate action.

Example 2-15 *Output from [Example 2-13](#)*

```
MyPrompt% python3 ex2-13_send_command_expect.py

Destination filename [startup-config]?
```

```
Success!  
MyPrompt%
```

The **expect_string** argument should be specific enough to uniquely identify the point at which the script needs to act. For instance, when prompted with **Destination filename [startup-config]?**, the phrase '**Destination filename**' serves as a clear, unique marker, allowing the program to automatically detect and handle the situation without user intervention. With the **send_command_expect()** method, Netmiko knows when to continue processing, making your program fully automated, without user input.

You may encounter other situations where you need to use the **expect_string** argument, such as those used for deleting files (**delete flash:test.txt**), reloading devices (**reload**), initiating software upgrades (**copy tftp://...**), or running interactive troubleshooting commands (extended **ping**).

The **save_config()** Method

The **save_config()** method in Netmiko is a convenient method for saving the running configuration to the startup configuration on a network device. The **save_config()** method is functionally similar to the **copy running-config startup-config** IOS command. The **save_config()** method bypasses any user intervention or, as we saw with the **send_command_expect()** method, a required **expect_string** setting. This makes **save_config()** an easier and less error-prone solution for saving configurations.

The code in [Example 2-16](#) shows the **save_config()** method being used to save the running configuration to the startup configuration. In this example, after calling **save_config()**,

we use the **send_command()** method with the **show startup-config** command to verify that the configuration has been successfully saved.

Example 2-16 *ex2-16_save_config.py*

```
import netmiko

# The secret parameter is included to allow privileged EXEC mode
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
                                    secret='spot')

# Save the running configuration to startup configuration
# The save_config() method saves the running configuration without
# requiring user intervention.
# Unlike send_command_expect(), it does not need an expect_string
# since it handles the entire process internally.
print('\nCopy running-config to startup-config, please wait...\n')
connection.save_config()

# Store results to variable including the IOS command
cmd_output = connection.send_command('show startup-config',
                                     strip_command=False)

# Print the current prompt and cmd_output which includes the command
print(connection.find_prompt(), cmd_output)

connection.disconnect()
```

Once again, this example uses the **strip_command** parameter in the **send_command()** method. By setting **strip_command=False**, the executed IOS command is included in the output returned by **send_command()**; in this case, it is stored in the variable **cmd_output**.

This example also uses the **find_prompt()** method to print the current device prompt ('Router-R1#'), prior to printing the variable **cmd_output**. The **cmd_output** variable includes the IOS command ('**show startup-config**'), and the output of that command, the **startup-config** file.

Combining these methods offers a clearer, more structured display of the operation and its results, as you can see in the output of this code shown in [Example 2-17](#). The output includes both the command and its results, providing clear confirmation of the operation.

Example 2-17 *Output from [Example 2-16](#)*

```
MyPrompt% python3 ex2-16_save_config.py

Copy running-config to startup-config, please wait...

Router-R1# show startup-config
Using 1773 out of 33554432 bytes
!
! Last configuration change at 16:25:20 UTC Fri Jan 24 2025
!
version 16.6
service timestamps debug datetime msec
service timestamps log datetime msec
platform qfp utilization monitor load 80
no platform punt-keepalive disable-kernel-core
platform hardware throughput level 100000
!
```

```
hostname Router-R1
!

<output omitted for brevity>

!
line con 0
  exec-timeout 0 0
  logging synchronous
  transport input none
  stopbits 1
line aux 0
  stopbits 1
line vty 0 4
  login local
  transport input ssh
line vty 5 15
  login local
  transport input ssh
!
wsma agent exec
!
wsma agent config
!
wsma agent filesys
!
wsma agent notify
!
!
end
```

Debugging Netmiko by Using the Session Log

The **session_log** parameter in **ConnectHandler** is an invaluable tool for debugging. It records all the interactions between your program and the device, including commands sent and responses received. By enabling **session_log** and specifying a filename, you can review the exact output from the device to identify errors, unexpected behaviors, or inconsistencies in your script. This is especially helpful when troubleshooting issues such as misconfigured commands, incorrect prompts, or privilege level problems.

Missing the Secret Parameter

In the code in [Example 2-18](#), we have removed the **secret** parameter from **ConnectHandler** and replaced it with a blank line. (the **secret** parameter is typically required for privileged EXEC mode access.) We have also included the **session_log** parameter with the filename **my_session_log.txt** to capture all session interactions.

Example 2-18 *ex2-18_send_command_expect_Bug.py*

```
import netmiko

# The secret parameter is intentionally omitted in this example
# This will cause an issue when attempting to use enable() to
# enter privileged EXEC mode.
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco',
```

```

session_log='my_session_log'

# Attempt to enter privileged EXEC mode without providing a secret
# Since the secret parameter is missing, the enable() method will fail
# and privileged commands will not work.
connection.enable()

# The IOS command, copy running-config startup-config, requires
# privileged EXEC mode access.
print(connection.send_command_expect('copy running-config startup-config',
                                     expect_string='Destination')

print('Success!')

connection.disconnect()

```

Example 2-19 shows that we encounter errors when we run this code. Netmiko provides helpful suggestions for troubleshooting, including "You can also look at the Netmiko session_log or debug log for more information." This emphasizes the importance of enabling the **session_log** parameter to review the command interactions and outputs for debugging purposes.

Example 2-19 *Output from [Example 2-18](#)*

```

MyPrompt% python3 ex2-18_send_command_expect_Bug.py

Netmiko/Code/ex2-18_send_command_expect_Bug.py", line 10, in <module>
    connection.enable()
  File "/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/netmiko/cisco_base_connection.py", line 25, in enable
    return super().enable(

```



```
File "/Library/Frameworks/Python.framework/Versions/3.10/lib/
packages/netmiko/base_connection.py", line 1935, in enable
```

```
<output omitted for brevity>
```

```
netmiko.exceptions.ReadTimeout:
```

```
Pattern not detected: 'Router\\-R1' in output.
```

```
Things you might try to fix this:
```

1. Adjust the regex pattern to better identify the terminating :
many situations the pattern is automatically based on the netwo
prompt.
2. Increase the read_timeout to a larger value.

```
You can also look at the Netmiko session_log or debug log for m
```

```
MyPrompt%
```

[Example 2-20](#) shows the session log, **my_session_log.txt**, from [Example 2-18](#). It reveals repeated attempts by Netmiko to elevate to privileged EXEC mode by using the **enable** command. However, because the **secret** parameter was not provided in **ConnectHandler**, Netmiko does not have the required password to successfully complete this step. As a result, the program fails to execute commands that require privileged EXEC mode. This shows the importance of including the **secret** parameter when working with commands such as '**copy running-config startup-config**' that necessitate elevated privileges.

Example 2-20 Session Log from [Example 2-18](#)

```
Router-R1>
Router-R1>terminal width 511
Router-R1>terminal length 0
Router-R1>
Router-R1>
Router-R1>enable
Password:
Password:
```

In the session log, you'll see lines such as **terminal width 511** and **terminal length 0**. These commands are issued right after login to ensure that the output from the device is returned in a format suitable for programmatic parsing. The **terminal width 511** command prevents line wrapping by setting the terminal width to a large value, while **terminal length 0** disables output pagination, avoiding interruptions like --More-- prompts. These adjustments help ensure that full command outputs are returned cleanly and without breaks, making them easier for your Python code to handle.

The solution is to include the **secret** parameter and password in **ConnectHandler** as shown here:

[illegible]

Requiring Privileged EXEC Mode Access

Example 2-21 demonstrates another use of the session log to help debug potential issues. Here, we have added the **secret** parameter, which is typically required for privileged EXEC mode access, back to **ConnectHandler**. However, we have intentionally commented out the **connection.enable()** method to prevent Netmiko from automatically entering privileged EXEC mode. Once again, we have included the **session_log** parameter to capture the interaction between Netmiko and IOS, so we can troubleshoot any issues that may arise.

On some systems (especially Windows), the session log may appear blank unless the program is given time to flush output. To ensure the log is complete, insert a brief delay (e.g., **time.sleep(1)**) **after disconnect()**, and explicitly close the session log file to flush any remaining output, as shown in [Example 2-21](#).

Example 2-21 `ex2-21_send_command_expect_Bug2.py`

[illegible]

```

# The enable() method is required to enter privileged EXEC mode
# It has been commented out here to demonstrate the impact of not
# connection.enable()

# Attempt to save the running configuration to startup configuration
# The copy running-config startup-config command requires privileged
# mode to execute.
# Without successfully entering privileged mode, this command will fail
print(connection.send_command_expect('copy running-config startup-config',
                                     expect_string='Destination')

print('Success!')

connection.disconnect()

# Optional but recommended to ensure the session log is written to disk
exits
time.sleep(1)

# Explicitly close the session log file (flushes remaining output)
if connection.session_log_file:
    connection.session_log_file.close()

```

Once again, we encounter errors when we run this code, as shown in [Example 2-22](#).

Example 2-22 *Output from [Example 2-21](#)*

```

MyPrompt% python3 ex2-18_send_command_expect_Bug2.py

Netmiko/Code/ex2-21_send_command_expect_Bug2.py", line 15, in <module>
    print(connection.send_command_expect('copy running-config startup-config',

```

```
File "/Library/Frameworks/Python.framework/Versions/3.10/lib/
packages/netmiko/base_connection.py", line 1751, in send_command
    return self.send_command(*args, **kwargs)
File "/Library/Frameworks/Python.framework/Versions/3.10/lib/
packages/netmiko/utilities.py", line 592, in wrapper_decorator

<output omitted for brevity>
```

Things you might try to fix this:

1. Explicitly set your pattern using the `expect_string` argument
2. Increase the `read_timeout` to a larger value.

You can also look at the Netmiko `session_log` or `debug log` for more

MyPrompt%

Because we have continued to use the **session_log** parameter, we have captured the problem. The session log reveals that IOS did not recognize the **copy running-config startup-config** command because the program never entered privileged EXEC mode. This behavior matches the error you would receive if you attempted to execute the same command directly from the router's command line while in user EXEC mode. The session log in [Example 2-23](#) allows you to easily identify the issue and emphasizes the value of logging for debugging and troubleshooting.

Example 2-23 Session Log from [Example 2-21](#)

```
Router-R1>
Router-R1>terminal width 511
Router-R1>terminal length 0
Router-R1>
Router-R1>copy running-config startup-config
```

```
^  
% Invalid input detected at '^' marker.
```

```
Router-R1>
```

The solution, as you can see in [Example 2-21](#), is to uncomment the **connection.enable()** method so the session enters privileged EXEC mode prior to the **send_command_expect()** statement.

Note

Earlier in this chapter, we mentioned that if the **secret** parameter and password are provided in **ConnectHandler**, Netmiko attempts to elevate to privileged EXEC mode automatically. We mentioned that this might fail due to device-specific configurations or an unusual prompt format. We stated that, to avoid inconsistencies across different devices, it is a good idea to explicitly include **connection.enable()** in your programs whenever you need privileged EXEC mode. However, you will find on some devices that Netmiko can elevate to privileged EXEC mode automatically without **connection.enable()**, but sometimes it cannot. To play it safe, always use **connection.enable()** when using commands that require privileged EXEC mode access.

Using Python Interactive Mode to Experiment with Netmiko

While we focus on writing complete Netmiko programs in this book—so you can see the framework, structure, and sequence of statements clearly—it's important to know that

Python also offers an **interactive mode**, which is a great place to play, experiment, and learn.

Interactive mode lets you type and run individual Python commands one at a time. It's especially helpful when you're testing ideas, exploring a new method, or just want to see what happens when you run a command.

How to enter interactive mode:

1. **Enter interactive mode** by typing **python3** in your terminal, using Python IDLE or another method.
2. **Import Netmiko** (**import netmiko**) or copy the import from one of your programs.
3. **Create the connection** using **netmiko.ConnectHandler(...)** — The easiest way is to copy the entire command including parameters from a working example.
4. **Play, experiment, and learn!** Try different **send_command()** calls, test methods like **find_prompt()**, and see how Netmiko responds.
5. **Close the connection** with **connection.disconnect()** when you're done.

[Example 2-24](#) shows a sample interactive session using MacOS:

Example 2-24 *Sample interactive session*

```
MyPrompt% python3
Python 3.10.3 (v3.10.3:a342a49189
Type "help", "copyright", "credits" or "license" for more inform
>>> import netmiko
>>> connection = netmiko.ConnectHandler(ip='192.168.1.1',
```

```

...         device_type='cisco_ios'
...         username='admin',
...         password='cisco',
...         secret='spot')
>>>
>>> print(connection.send_command('show ip interface brief'))
Interface                IP-Address      OK? Method Status
GigabitEthernet0/0/0     192.168.1.1     YES NVRAM  up
GigabitEthernet0/0/1     192.168.2.1     YES NVRAM  up
GigabitEthernet0/0/2     unassigned      YES NVRAM  administrative
GigabitEthernet0         unassigned      YES NVRAM  administrative
>>> connection.disconnect()
>>>
>>> ^D
MyPrompt%

```

This interactive environment is perfect for building confidence and understanding. Use it freely alongside the structured programs in this book to deepen your learning. If you have any “what-if” questions, just “try it!”

Netmiko Supported Network Operating Systems

Netmiko is designed to support a wide range of network devices across many different vendors and platforms. The device type is specified in the **device_type** parameter

when creating a connection using **ConnectHandler**. Below is a list of commonly supported network operating systems, organized by vendor.

Popular platforms supported by Netmiko include:

Cisco:

- cisco_ios – Cisco IOS (routers/switches)
- cisco_xe – IOS XE (same as IOS, but newer codebase)
- cisco_xr – IOS XR (service provider routers)
- cisco_nxos – NX-OS (data center switches like Nexus)
- cisco_asa – ASA firewalls
- cisco_wlc – Wireless LAN Controllers (older AireOS)
- cisco_s300 – Small business switches (SG series)

Arista:

- arista_eos – Arista EOS (Extensible Operating System)

Juniper:

- juniper_junos – Junos OS (used on routers and switches)

HP / Aruba:

- hp_procurve – HP ProCurve switches
- aruba_os – Aruba Mobility Controllers

Dell:

- dell_os10 – OS10
- dell_force10 – Force10

Ubiquiti:

- ubiquiti_edge – EdgeOS (EdgeRouters)

F5:

- f5_ltm – BIG-IP Local Traffic Manager (via SSH)

Palo Alto:

- paloalto_panos – PAN-OS (firewalls)

Checkpoint:

- checkpoint_gaia – Gaia OS

Fortinet:

- fortinet – FortiOS (via SSH, not API)

Linux/Unix (for automation of generic devices):

- linux – Generic Linux server
- generic – When all else fails (basic SSH)

To get a complete and accurate list of network operating systems supported by the version of Netmiko you are using, [Example 2-25](#) shows how to obtain this information from the Python interactive mode.

Example 2-25 *Obtaining the current list of Netmiko supported Network Operating Systems*

```
MyPrompt% python3
Python 3.10.3 (v3.10.3:a342a49189, Mar 16 2022, 09:34:18) [Clang
darwin
Type "help", "copyright", "credits" or "license" for more inform
>>>
from netmiko import ConnectHandler
from netmiko.ssh_dispatcher import CLASS_MAPPER_BASE
```

```
print(CLASS_MAPPER_BASE.keys())
```

```
<List of supported network operating systems>
```

```
>>> ^D
```

```
MyPrompt%
```

Summary

In this chapter, we have introduced Netmiko, a proven and widely used Python library that simplifies network automation when connected to devices using SSH. We began by exploring the basic framework of a Netmiko program, including **ConnectHandler**, which creates the SSH connection to the network device. We introduced Netmiko commands, typically called methods, such as the **send_command()** method that is used to send IOS commands to a device, and the **disconnect()** method, which is used to properly close the connection. These foundational concepts set the stage for more advanced use cases.

As we progressed, we discussed the flexibility in presenting command outputs, showing how to use the **find_prompt()** method and the **strip_command** parameter to include the prompt and command with the output. Showing the prompt and the command with the output can help to contextualize command results, and it is especially helpful when working with multiple devices or documenting configurations.

We discussed troubleshooting techniques, focusing on the **session_log** parameter to capture all interactions with the

device. Using two examples, we showed how debugging logs can help identify common issues, such as missing or misconfigured methods or parameters.

In the next chapter, we will build on these concepts and introduce the methods for automating device configuration. This will allow you to apply configuration changes across devices efficiently, further unlocking the power of network automation with Python and Netmiko.

Chapter 3. Configuring Devices with Netmiko

In the previous chapter, we explored how to use the Netmiko **send_command()** method to establish SSH connections and retrieve information from Cisco devices. We focused on gathering information, such as interface statuses and routing information, using standard **show** commands. While retrieving information is important for monitoring and troubleshooting, real-world network automation often requires making configuration changes to devices efficiently and reliably.

This chapter shifts the focus from retrieving information to configuring network devices with Netmiko. We will begin by exploring how the **send_command()** can be used for configuration, but as you'll see, it is not always the most efficient method—especially when multiple configuration lines need to be applied. So, we will look at using **send_config_set()**, which simplifies the process by allowing us to send a list of configuration commands in a structured and efficient way.

Finally, we will cover **send_config_from_file()**, which enables us to apply configuration commands stored in an external text file. This method is

particularly useful when working with prewritten configurations or applying standardized settings across multiple devices.

By the end of this chapter, you will understand the different ways you can use Netmiko to configure Cisco devices.

The `send_command()` Method

In [Chapter 2](#), “[Getting Started with Netmiko](#),” you saw how to use the `send_command()` method to retrieve information. This versatile Netmiko method can be used for both retrieving information and configuring a device. When used with **show** commands, it captures and returns device output just as if the command had been entered manually at the CLI, making it useful for monitoring and troubleshooting.

The `send_command()` method can also be used to execute **configuration** commands, such as setting an interface description or enabling specific features. However, because it is designed primarily for single-line commands, it is not very efficient for making multiple configuration changes. For bulk configurations, the `send_config_set()` method is generally preferred, as it ensures proper command execution in configuration mode.

Using `send_command()` for Retrieving Information

[Example 3-1](#) is similar to programs you saw in [Chapter 2](#). It establishes an SSH connection to a Cisco device and retrieves the status of its interfaces. However, here we have removed the `secret='spot'` parameter and the

connection.enable() method because **show ip interface brief** only requires user EXEC mode. There is no need to provide a privileged EXEC password (secret parameter) when setting up the connection with **ConnectHandler**. The program in [Example 3-1](#) simply connects to the device, sends the command, prints the output, and then disconnects.

Example 3-1 *ex3-1_send_command_show.py*

```
# Import the Netmiko Library
import netmiko

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco')

# Using the connection to send an IOS command
print(connection.send_command('show ip interface brief'))

# Close the SSH connection
connection.disconnect()
```

[Example 3-2](#) shows that when the program is executed, the output is similar to what you would see in the CLI. It displays the status of each interface along with its assigned IP address.

Example 3-2 *Output from Example 3-1*

```
MyPrompt% python3 ex3-1_send_command_show.py
Interface                IP-Address    OK? Method Status
GigabitEthernet0/0/0    192.168.1.1  YES NVRAM  up
```

```
GigabitEthernet0/0/1    192.168.2.1    YES NVRAM    up
GigabitEthernet0/0/2    unassigned     YES NVRAM    administrative
GigabitEthernet0        unassigned     YES NVRAM    administrative
MyPrompt%
```

Using `send_command()` for Individual Configuration Commands

You can also use the **`send_command()`** method for configuration tasks where you want to send individual configuration commands directly to a device. While this is not the most efficient way to apply multiple changes, it is useful for making quick modifications, such as setting an interface description or enabling an interface.

[Example 3-3](#) demonstrates how to use the **`send_command()`** method to configure a static default route on a Cisco device. The **`ip route`** command is a global configuration command, which requires the session to be in privileged EXEC mode before entering global configuration mode.

Example 3-3 *ex3-3_send_command_configure.py*

```
import netmiko

connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco',
                                     secret='spot') # Enable pas:

# send_command() for configuration is similar to CLI
# secret password is required
```



```
connection.enable()

# Global config mode
connection.config_mode()

# Global configuration command
connection.send_command('ip route 0.0.0.0 0.0.0.0 192.168.2.2')

# Exit global config mode
connection.exit_config_mode()

# Verify running-config
print('\nIOS command: show running-config | include ip route')
print(connection.send_command('show running-config | include ip

# Verify routing table
print('\nIOS command: show ip route | begin Gateway')
print(connection.send_command('show ip route | begin Gateway'))

connection.disconnect()
```

The program first enables privileged mode by using the **enable()** method, which is necessary for making configuration changes. It then enters global configuration mode by using the **config_mode()** method, which allows the program to modify router settings.

The method **send_command('ip route 0.0.0.0 0.0.0.0 192.168.2.2')** configures a default static route and directs all unknown traffic to the next-hop address 192.168.2.2.

Once the route is applied, the **exit_config_mode()** method returns to privileged EXEC mode.

To verify that the configuration was successfully applied, the program prints the output of **print(connection.send_command('show running-config | include ip route'))**, which filters and displays only the lines containing the **ip route** command from the running configuration. This helps confirm that the static default route was added correctly.

In addition, **print(connection.send_command('show ip route | begin Gateway'))** examines the routing table, starting at the section that shows the gateway of last resort. This verifies that the default static route is not only configured but active in the routing table.

Finally, the script closes the SSH session by using the **disconnect()** method to cleanly terminate the connection.

Note

Each method is preceded by the **connection** variable, or object. The **connection** object represents the active SSH session established with the network device using **ConnectHandler**. This ensures that each method call—such as **enable()**, **config_mode()**, **send_command()**, and **disconnect()**—is executed using the established connection, allowing the program to seamlessly interact with the device.

[Example 3-4](#) shows the output from executing the code in [Example 3-3](#).

Example 3-4 Output from [Example 3-3](#)

```
MyPrompt% python3 ex3-3_send_command_configure.py
```

```
IOS command: show running-config | include ip route
```

```
ip route 0.0.0.0 0.0.0.0 192.168.2.2
```

```
IOS command: show ip route | begin Gateway
```

```
Gateway of last resort is 192.168.2.2 to network 0.0.0.0
```

```
S*    0.0.0.0/0 [1/0] via 192.168.2.2
```

```
    192.168.1.0/24 is variably subnetted, 2 subnets, 2 masks
```

```
C      192.168.1.0/24 is directly connected, GigabitEthernet0,
```

```
L      192.168.1.1/32 is directly connected, GigabitEthernet0,
```

```
    192.168.2.0/24 is variably subnetted, 2 subnets, 2 masks
```

```
C      192.168.2.0/24 is directly connected, GigabitEthernet0,
```

```
L      192.168.2.1/32 is directly connected, GigabitEthernet0,
```

```
MyPrompt%
```

The output in [Example 3-4](#) confirms that the static default route was successfully configured on the device. The first command, **show running-config | include ip route**, displays the running configuration, starting with **ip route** commands, verifying that the **ip route 0.0.0.0 0.0.0.0 192.168.2.2** command was applied.

The second command, **show ip route | begin Gateway**, provides a view of the device's routing table, where the newly configured default route (**S* 0.0.0.0/0 [1/0] via 192.168.2.2**) appears, confirming that the router will forward all unknown traffic to 192.168.2.2.

This output validates that the **send_command()** method successfully applied the configuration changes.

Note

You might want to use the **find_prompt()** and **cmd_output()** methods discussed in [Chapter 2](#) to re-create how a command might appear in the CLI.

Using `send_command()` for Multiple Configuration Commands

Although multiple `send_command()` global configuration mode statements can be used for configuration, the `send_config_set()` method is generally preferred for applying multiple configuration lines.

The `send_command()` method can be used to enter sub-modes, such as interface configuration mode (`interface GigabitEthernet0/0/0`), but using it this way so requires the use of the `expect_string` parameter to handle prompt changes. Because this adds complexity and is not the typical approach for multiple configuration commands, it is recommended to use the `send_config_set()` method, which automatically enters and exits configuration mode as needed. (The `send_config_set()` method is discussed in the next section.)

To keep things straightforward, this book focuses on the much more common `send_config_set()` method for making configuration changes. However, for those interested in how `send_command()` can be used with the `expect_string` parameter, we have included an example in [Appendix B](#), “[Understanding expect_string with send_command\(\)](#).”

Using `send_config_set()` for Configuration Commands

Using the `send_config_set()` method is a useful and easy way to apply multiple configuration commands to a network device. Unlike `send_command()`, which executes a single command at a time, `send_config_set()` automatically enters configuration mode, applies all commands from a Python list, and then exits configuration mode. This makes it

the preferred method for making multiple changes efficiently, such as configuring interfaces, setting descriptions, and adding static routes.

[Example 3-5](#) shows an example of how to streamline the configuration process with **send_config_set()**, which applies interface descriptions to two GigabitEthernet interfaces.

Example 3-5 *ex3-5_send_config_set.py*

```
# Import the Netmiko Library
import netmiko

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco',
                                     secret='spot') # Enable pas

# Define a list of interface configuration commands
interface_description_list = [
    'interface gig 0/0/0',
    'description LAN interface - used Netmiko',
    'exit',

    'interface gig 0/0/1',
    'description Connection to R2 - used Netmiko',
    'exit'
]

# Enter privileged EXEC mode (required for making configuration
connection.enable()
```

```
# Apply the list of configuration commands to the device
connection.send_config_set(interface_description_list)

# Display the IOS command before executing it for clarity
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0,

print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0,

# Close the SSH connection
connection.disconnect()
```

A *list* is a collection of items stored in a specific order. Lists are useful when you need to store multiple values and access them easily. [Example 3-5](#) uses a list called **interface_description_list** to store a series of Cisco IOS configuration commands. Each item in the list is enclosed in quotes, and the list items are separated by commas to ensure that the commands are processed in order. The list includes interface selection commands (such as **interface gig 0/0/0**), interface descriptions, and **exit** commands to return to global configuration mode.

interface_description_list is then passed to the **send_config_set()** method, which automatically executes all commands in sequence with **send_config_set(interface_description_list)**. This simplifies the automated configuration process.

[Example 3-6](#) shows the output from executing the code in [Example 3-5](#). This output confirms that the program successfully applied the interface configurations using **send_config_set()**.

Example 3-6 Output from [Example 3-5](#)

```
MyPrompt% python3 ex3-5_send_config_set.py
IOS command: show running-config | begin interface GigabitEthernet0/0
interface GigabitEthernet0/0/0
  description LAN interface - used Netmiko
  ip address 192.168.1.1 255.255.255.0
  negotiation auto
!
interface GigabitEthernet0/0/1
  description Connection to R2 - used Netmiko
  ip address 192.168.2.1 255.255.255.0
  negotiation auto
!
<output omitted for brevity>
!
end
MyPrompt%
```

Configuring and Displaying Commands with `send_config_set()`

In this section, we will look at storing the output of **`send_config_set()`** in a variable in order to display all the configuration commands along with their corresponding prompts. When executing configuration commands, Netmiko returns the full command output, including the device's responses. By storing the result of **`send_config_set()`** in a variable, you can capture both the executed commands and their prompts and the system's responses, which allows you to review the entire configuration process. This helps you

ensure that all commands in the list were executed correctly and provides a clear view of the configuration process.

The code in [Example 3-7](#) configures IPv6 on a Cisco device by applying a series of commands from a list using Netmiko. It defines a list of IPv6 configuration commands, **ipv6_configuration_list**, enables IPv6 routing, and assigns both global unicast and link-local IPv6 addresses to two interfaces.

The **send_config_set()** method is used to apply these configurations, and its output is stored in a variable (**cli_output**) in order to display the executed commands along with their prompts. Finally, the example verifies the configuration by using the **show ipv6 interface brief** command to confirm that the IPv6 addresses have been applied correctly.

Notice that the configuration list (**ipv6_configuration_list**) includes comments to help those who may not be very familiar with configuring IPv6. The comments provide context for each command as it is applied to the device.

Example 3-7 *ex3-7_send_config_set_variable.py*

```
# Import the Netmiko Library
import netmiko

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco',
                                     secret='spot') # Enable pas:

# Define a list of IPv6 configuration commands
```



```

ipv6_configuration_list = [
    'ipv6 unicast-routing',    # Enable IPv6 routing

    # Configure IPv6 addresses on GigabitEthernet0/0/0
    'interface gig 0/0/0',
    'ipv6 address 2001:db8:cafe:1::1/64', # Assign global IPv6 address
    'ipv6 address fe80::1:1 link-local',  # Assign link-local address
    'exit',

    # Configure IPv6 addresses on GigabitEthernet0/0/1
    'interface gig 0/0/1',
    'ipv6 address 2001:db8:cafe:2::1/64', # Assign global IPv6 address
    'ipv6 address fe80::2:1 link-local',  # Assign link-local address
    'exit'
]

# Enter privileged EXEC mode (required for making configuration changes)
connection.enable()

# Apply the list of configuration commands to the device
# Store the output (including prompts and commands) in cli_output
cli_output = connection.send_config_set(ipv6_configuration_list)

# Display the full command execution output, including prompts
print(cli_output)

# Display the IOS command before executing it for clarity
print('\nIOS command: show ipv6 interface brief')

# Verify IPv6 interface configuration commands
print(connection.send_command('show ipv6 interface brief'))

```

```
# Close the SSH connection
connection.disconnect()
```

The output in [Example 3-8](#) shows the complete sequence of configuration prompts and commands that were executed on the device. Because [Example 3-7](#) stores the result of **send_config_set()** in the variable **cli_output**, it is possible to print it and see exactly what was applied.

The output includes the automatic transition into global configuration mode (**configure terminal**). This is followed by the commands from **ipv6_configuration_list** that enable IPv6 unicast routing and the configuration of IPv6 addresses on GigabitEthernet0/0/0 and GigabitEthernet0/0/1.

Each interface is assigned both a global IPv6 address (2001:db8:cafe:x::1/64) and a link-local address (fe80::x:1 link-local). The **exit** commands indicate the return to global configuration mode, and the final **end** command goes back to privileged EXEC mode.

To verify that the configuration was successfully applied, you can use the **send_command('show ipv6 interface brief')** method, which displays the IPv6 address assigned to each interface. The output confirms that GigabitEthernet0/0/0 and GigabitEthernet0/0/1 are both up and have the correct IPv6 addresses, while other interfaces remain unassigned and administratively down.

Example 3-8 Output from [Example 3-7](#)

```
MyPrompt% python3 ex3-7_send_config_set_variable.py
configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router-R1(config)#ipv6 unicast-routing
```

```
Router-R1(config)#interface gig 0/0/0
Router-R1(config-if)#ipv6 address 2001:db8:cafe:1::1/64
Router-R1(config-if)#ipv6 address fe80::1:1 link-local
Router-R1(config-if)#exit
Router-R1(config)#interface gig 0/0/1
Router-R1(config-if)#ipv6 address 2001:db8:cafe:2::1/64
Router-R1(config-if)#ipv6 address fe80::2:1 link-local
Router-R1(config-if)#exit
Router-R1(config)#end
Router-R1#

IOS command: show ipv6 interface brief
GigabitEthernet0/0/0    [up/up]
    FE80::1:1
    2001:DB8:CAFE:1::1
GigabitEthernet0/0/1    [up/up]
    FE80::2:1
    2001:DB8:CAFE:2::1
GigabitEthernet0/0/2    [administratively down/down]
    unassigned
GigabitEthernet0        [administratively down/down]
    unassigned
MyPrompt%
```

When you use **send_config_set()**, Netmiko automatically enters configuration mode, executes the provided commands, and then exits configuration mode before returning control to privileged EXEC mode. As part of this process, Netmiko sends an implicit end command after completing all configuration changes. So, even though the program does not explicitly include **end**, Netmiko sends it as part of **send_config_set()** to ensure that the device exits configuration mode properly. This makes automation

smoother, preventing the program from accidentally leaving the session in configuration mode.

Note

Now that IPv6 is enabled on the device, it is operating in a *dual-stack* environment, meaning it supports both IPv4 and IPv6 simultaneously. Since Netmiko relies on SSH for connectivity, it can use either IPv4 or IPv6 to establish a connection, as long as the device is reachable via the chosen protocol. If an IPv6 address is used instead of an IPv4 address in the **ip** parameter of **ConnectHandler**, Netmiko will seamlessly connect using IPv6.

Note

An alternative to assigning a list of commands to a variable is to include the list directly as a parameter (positional argument) in the **send_config_set()** method:

```
send_config_set( [  
    'interface gig 0/0/0',  
    'ipv6 address 2001:db8:cafe:1::1/64',  
    'ipv6 address fe80::1:1 link-local',  
    'exit'] )
```

Sending Commands from a File with **send_config_from_file()**

The **send_config_from_file()** method functions similarly to **send_config_set()**, allowing you to apply multiple configuration commands to a network device. However, instead of using a Python list to store the commands, this method reads the commands from a *separate text file*. This approach is especially useful when working with prewritten

configurations, as it makes managing and reusing command sets easier. By storing configuration commands in a file, you can keep the program clean and organized and apply large or frequently used configurations when required.

[Example 3-9](#) shows the text file **r1_ipv6_update.txt**, which the program will use to update the IPv6 global unicast addresses and interface descriptions on two interfaces. The configuration file contains a set of commands that will be applied to GigabitEthernet0/0/0 and GigabitEthernet0/0/1, replacing the existing IPv6 addresses and updating the descriptions.

Specifically, this file changes the global unicast address prefix from **2001:db8:cafe::/64** to **2001:db8:c0de::/64**, with a zero instead of the letter o in code.

Note

Notice that this example uses the **no ipv6 address** command to remove previously configured addresses. With IPv6, an interface can have multiple IPv6 global unicast addresses on the same network or on different networks. So, when replacing an IPv6 address, it is important to remove any previous address that is no longer being used.

Example 3-9 *r1_ipv6_update.txt*

```
interface gig 0/0/0
description Updated LAN interface using Netmiko
no ipv6 address 2001:db8:cafe:1::1/64
ipv6 address 2001:db8:c0de:1::1/64
exit

interface gig 0/0/1
description Updated interface to R2 using Netmiko
```

```
no ipv6 address 2001:db8:cafe:2::1/64
ipv6 address 2001:db8:c0de:2::1/64
exit
```

[Example 3-10](#) uses the **send_config_from_file()** method to apply configuration changes stored in an external text file. This approach is similar to using **send_config_set()**, but instead of defining the commands within the program using a list, it references a separate file, **r1_ipv6_update.txt**, displayed in [Example 3-9](#).

The **send_config_from_file(config_file)** statement references the variable **config_file**, which contains the filename '**r1_ipv6_update.txt**'. This file holds the configuration commands that will be executed on the device. When you specify only the filename and not a full path, Python assumes that the file is located in the current working directory (CWD)—the same directory from which the script is being executed. If the file is not in this directory, the program will return an error because it won't be able to locate the file. This approach keeps things simple, but in cases where the program may run from different locations, specifying the full file path would be a more reliable solution.

Example 3-10 ex3-10_send_config_from_file.py

```
# Import the Netmiko Library
import netmiko

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
                                     password='cisco',
```

```
secret='spot') # Enable password

# Specify the configuration file
# (assumed to be in the current working directory)
config_file = ('r1_ipv6_update.txt')

# Enter privileged EXEC mode (required for making configuration
connection.enable()

# Display the current interface configuration before applying updates
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0,')
print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0,')

# Apply the configuration commands from the file to the device
# Store the output (including prompts and commands) in cli_output
cli_output = connection.send_config_from_file(config_file)

# Display the full command execution output, including prompts
print(cli_output)

# Display the updated interface configuration after applying the updates
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0,')
print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0,')

# Close the SSH connection
connection.disconnect()
```

[Example 3-11](#) shows the output from [Example 3-10](#). It confirms that the configuration commands from the file **r1_ipv6_update.txt** were successfully applied to the device. Much as with **send_config_set()**, assigning the result of **send_config_from_file()** to a variable allows you to capture the executed commands, their prompts, and the system's responses.

The first section of the output shows the initial interface configuration, before any changes are made. The GigabitEthernet0/0/0 and GigabitEthernet0/0/1 interfaces contain the existing IPv6 global unicast addresses under the 2001:DB8:CAFE::/64 prefix, along with their interface descriptions and other settings.

The second section of the output displays the actual configuration process, using **send_config_from_file(config_file)**. Netmiko enters global configuration mode, applies the new descriptions, removes the old IPv6 addresses (**no ipv6 address 2001:db8:cafe:x::1/64**), and assigns the updated IPv6 addresses with the new **2001:DB8:CODE::/64** prefix. After applying the changes, Netmiko automatically exits configuration mode and returns to privileged EXEC mode.

Finally, the program verifies the updates by running **show running-config | begin interface GigabitEthernet0/0/0** again. The output confirms that both interfaces have been updated, showing the new descriptions and IPv6 addresses reflecting the changes made in the configuration file. This demonstrates how **send_config_from_file()** can efficiently apply multiple configuration changes to help ensure both accuracy and consistency.

Example 3-11 *Output from [Example 3-10](#)*


```
MyPrompt% python3 ex3-10send_config_from_file.py
```

```
IOS command: show running-config | begin interface GigabitEthernet0/0/0
interface GigabitEthernet0/0/0
  description LAN interface - used Netmiko
  ip address 192.168.1.1 255.255.255.0
  negotiation auto
  ipv6 address FE80::1:1 link-local
  ipv6 address 2001:DB8:CAFE:1::1/64
!
interface GigabitEthernet0/0/1
  description Connection to R2 - used Netmiko
  ip address 192.168.2.1 255.255.255.0
  negotiation auto
  ipv6 address FE80::2:1 link-local
  ipv6 address 2001:DB8:CAFE:2::1/64
!
<output omitted for brevity
!
end
```

```
configure terminal
```

```
Enter configuration commands, one per line. End with CNTL/Z.
```

```
Router-R1(config)#interface gig 0/0/0
```

```
Router-R1(config-if)#description Updated LAN interface using Netmiko
```

```
Router-R1(config-if)#no ipv6 address 2001:db8:cafe:1::1/64
```

```
Router-R1(config-if)#ipv6 address 2001:db8:c0de:1::1/64
```

```
Router-R1(config-if)#exit
```

```
Router-R1(config)#
```

```
Router-R1(config)#interface gig 0/0/1
```

```
Router-R1(config-if)#description Updated interface to R2 using Netmiko
```

```
Router-R1(config-if)#no ipv6 address 2001:db8:cafe:2::1/64
```

```
Router-R1(config-if)#ipv6 address 2001:db8:c0de:2::1/64
```

```
Router-R1(config-if)#exit
Router-R1(config)#end
Router-R1#

IOS command: show running-config | begin interface GigabitEthernet
interface GigabitEthernet0/0/0
  description Updated LAN interface using Netmiko
  ip address 192.168.1.1 255.255.255.0
  negotiation auto
  ipv6 address FE80::1:1 link-local
  ipv6 address 2001:DB8:C0DE:1::1/64
!
interface GigabitEthernet0/0/1
  description Updated interface to R2 using Netmiko
  ip address 192.168.2.1 255.255.255.0
  negotiation auto
  ipv6 address FE80::2:1 link-local
  ipv6 address 2001:DB8:C0DE:2::1/64
!
<output omitted for brevity
!
end

MyPrompt%
```

Ensuring That Python Uses the Current Working Directory

When a Python program tries to open a config file using a relative path, it defaults to looking in the current working directory, which may not be the same directory where the program file is located. You might want to ensure that a

program always locates the configuration file in the same directory as the program, regardless of how or where the program is run. This is important when the program is executed from a different working directory. To ensure that the program always finds the configuration file in the same directory as the program itself, you can use **os.path.abspath(__file__)** along with **os.path.dirname()** to build the correct path—regardless of the path used to launch the program.

The code in [Example 3-12](#) ensures that the program looks for and locates the configuration file in the same directory where the program itself is located rather than relying on Python to use the CWD.

If you are new to Python, it is not necessary to understand the details of how the **os.path** module works. However, in case you are interested, here are the details: The **os** library helps build the correct path by using **os.path.abspath(__file__)** to determine the program's full path and **os.path.dirname()** to extract just the directory portion. Then, **os.path.join()** combines that directory with the filename **r1_ipv6_update.txt** to construct the full path. The full file path, including the filename, is stored in the variable **config_file**.

os.path.dirname(os.path.abspath(__file__)) returns the directory where the script file physically resides, which may not be the same as the CWD if the script is imported or run from another location.

Note

The **os.getcwd()** function can be used to return the CWD, which is the folder from which the Python program was launched or executed—not necessarily the directory where the program file is saved. For example, if you run a script from the terminal while

your working directory is /Users/rick/Documents, that path becomes the CWD, even if the script itself is stored in /Users/rick/Projects. This distinction is important when using relative file paths in your code.

By using this approach, the script can reliably locate the configuration file as long as it is in the same folder as the program, even if the script is executed from a different directory. If the file were stored elsewhere, you would need to manually specify its full path, such as **`'/Users/rick/r1_ipv6_update.txt'`**, or modify the script to reference another directory.

Example 3-12 demonstrates how to use the **os** library to reliably locate **r1_ipv6_update.txt** in the same directory as the program.

Example 3-12 *ex3-12_send_config_from_file_path.py*

```
# Import the Netmiko Library
import netmiko
import os

# Get the current directory path to ensure the program
# can locate the configuration file
my_directory = os.path.dirname(os.path.abspath(__file__))

# Construct the full path to the configuration file
config_file = os.path.join(my_directory, 'r1_ipv6_update.txt')

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                     device_type='cisco_ios',
                                     username='admin',
```

```
        password='cisco',
        secret='spot') # Enable password

# Enter privileged EXEC mode (required for making configuration
connection.enable()

# Display the current interface configuration before applying up
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0,
print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0,

# Apply the configuration commands from the file to the device
# Store the output (including prompts and commands) in cli_output
cli_output = connection.send_config_from_file(config_file)

# Display the full command execution output, including prompts
print(cli_output)

# Display the updated interface configuration after applying the
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0,
print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0,

# Close the SSH connection
connection.disconnect()
```

Once again, understanding the details of how the **os.path** functions is not necessary.

Summary

In this chapter, we have explored how to use Netmiko to configure network devices efficiently. We started by demonstrating how **send_command()** can be used for configuration, and we quickly saw its limitations—especially when working with multiple commands. To address this, we introduced **send_config_set()**, which makes it possible to send a list of configuration commands in a structured and automated way. This method ensures that Netmiko enters and exits configuration mode automatically, making it a more practical solution for most automation tasks.

We then covered **send_config_from_file()**, which makes it possible to apply configuration changes stored in an external text file. This approach is especially useful when working with predefined configurations or deploying standardized settings across multiple devices. In addition, we discussed how to manage file paths by using Python's **os** library to ensure that configuration files are reliably located.

You should now have a solid foundation for understanding how Netmiko can be used for device configuration. In the next chapter, we will take automation a step further by looking at how to configure multiple devices efficiently by using a **for** loop, which makes it possible to scale network automation workflows.

Chapter 4. Accessing Multiple Devices with Netmiko

So far in this book, you have seen how to retrieve information and configure IOS commands for a single device. In this chapter you'll learn how to configure multiple devices efficiently by using a **for** loop, which allows you to scale your network automation and programmability as needed.

Using a Variable to Store an IP Address

When working with network automation, it's common to refer to devices by their IP addresses. Instead of hardcoding an IP address directly into the **ConnectHandler** function, you can assign it to a variable. This approach makes the program more flexible and easier to maintain.

[Example 4-1](#) defines a variable named **device** and stores a single IP address, **192.168.1.1**, as a string, '**192.168.1.1**'. We then use this variable as the value for the **ip** parameter in **ConnectHandler**. While this may not seem particularly useful at first—after all, you could just type the IP address directly—it sets the foundation for scaling the program.

Example 4-1 *ex4-1_send_command_device.py*

```
# Import the Netmiko Library
import netmiko

# Assign IP address of the device to a variable
device = '192.168.1.1'

# Establish an SSH connection to the device
connection = netmiko.ConnectHandler(
    ip=device,      # Assign the IP address using
    device_type='cisco_ios',
    username='admin',
    password='cisco',
    secret='spot'   # Enable password if required
)

# Send a command to the device and print the output
print(connection.send_command('show ip interface brief'))

# Close the SSH connection
connection.disconnect()
```

[Example 4-2](#) shows that when the program is executed, it displays the status of each interface, along with its assigned IP address.

Example 4-2 *Output from [Example 4-1](#)*

```
MyPrompt% python3 ex4-1_send_command_device.py
Interface          IP-Address      OK? Method Status
GigabitEthernet0/0/0 192.168.1.1    YES NVRAM  up
GigabitEthernet0/0/1 192.168.2.1    YES NVRAM  up
```



```
GigabitEthernet0/0/2    unassigned      YES NVRAM    administrative
GigabitEthernet0       unassigned      YES NVRAM    administrative
MyPrompt%
```

By using a variable, you set the stage for handling multiple devices dynamically. Instead of modifying the **ConnectHandler** call for each new device, you can reuse the same variable name and iterate over a list of IP addresses. This allows you to efficiently connect to and configure multiple devices without duplicating code. We'll explore this advantage in the next section, where we introduce looping over a list of devices.

Understanding Python for Loops

If you are already familiar with **for** loops in Python, you might want to skim or skip this section.

A **for** loop in Python allows you to iterate over a sequence, such as a list of IP addresses, and perform an action for each item in the list. This is useful in network automation because it enables you to execute the same commands on multiple devices without writing repetitive code.

[Example 4-3](#) is a simple example that demonstrates how a **for** loop works using IP addresses.

Example 4-3 *ex4-3_sample_for_loop.py*

```
# Define a list of IP addresses
ip_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']

# Loop through each IP address in the list
for ip in ip_list:
    print("Connecting to device with IP:", ip)
```

```
# Program execution continues here after the for loop finishes
# iterating through all IP addresses
print('Loop has completed.')
```

[Example 4-4](#) shows the output of this program. Notice that it displays each of the IP addresses in the Python list **ip_list**.

Note

This script does not actually connect to these IP addresses. It simply shows how a **for** loop works.

Example 4-4 Output from [Example 4-3](#)

```
Connecting to device with IP: 192.168.1.1
Connecting to device with IP: 192.168.2.2
Connecting to device with IP: 192.168.3.2
Loop has completed.
```

How a for Loop Works, Step-by-Step

To help you better understand **ex4-3_sample_for_loop.py**, we have isolated the **for** loop in [Example 4-5](#) to more closely examine how the **for** loop works.

Example 4-5 Isolating the **for** Loop

```
ip_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']

for ip in ip_list:
    print("Connecting to device with IP:", ip)
```

This is what the for loop in this example does:

1. The **for** loop begins by assigning the first value in **ip_list** to the variable **ip** (that is, **ip = '192.168.1.1'**).
2. The indented command inside the loop, **print("Connecting to device with IP: ", ip)**, is executed using the value **'192.168.1.1'**.
3. Once the last indented command inside the loop is executed, the process goes back to the **for** statement.
4. The variable **ip** is now assigned the next value in **ip_list**, which is **'192.168.2.2'**.
5. The indented command runs again, using this new value of **ip**.
6. This process repeats until there are no more values left in **ip_list**.
7. When the loop reaches the end of the list, the **for** loop terminates, and control of the program moves to the next statement at the same indentation level as the **for** loop—in this case, **print("Loop has completed.")**.

This example shows how a **for** loop can be used to iterate through a list of IP addresses. In the next section, we'll extend this concept to establish SSH connections and send commands to multiple network devices by using Netmiko.

Python F-Strings

An *F-string* (formatted string literal) in Python provides another and simpler way to insert variables directly into a string. It is created by prefixing the string with the letter **f** and using curly braces (**{}**) to embed variables or expressions. The **{ip}** inside the string is replaced with the actual value of the variable **ip**. [Example 4-6](#) shows both the

traditional way of printing strings and variables and the use of an F-string.

Example 4-6 *Traditional Printing of Strings with Variables and Use of an F-String*

```
print("Connecting to device with IP:", ip)

# Is equivalent to:

print(f"Connecting to device with IP: {ip}")
```

You can use either variables or f-strings, and you should be familiar with both options because they are both commonly used.

Iterating Through Multiple Devices

Now we will look at how to efficiently connect to multiple network devices by using Netmiko and a Python **for** loop. [Figure 4-1](#) shows the topology we will be using in this example.

Let's say we want to examine the IP routing table for each router. Instead of manually establishing separate SSH connections to each router, we will automate this process by iterating through a list of IP addresses. This approach makes network management more scalable and efficient.

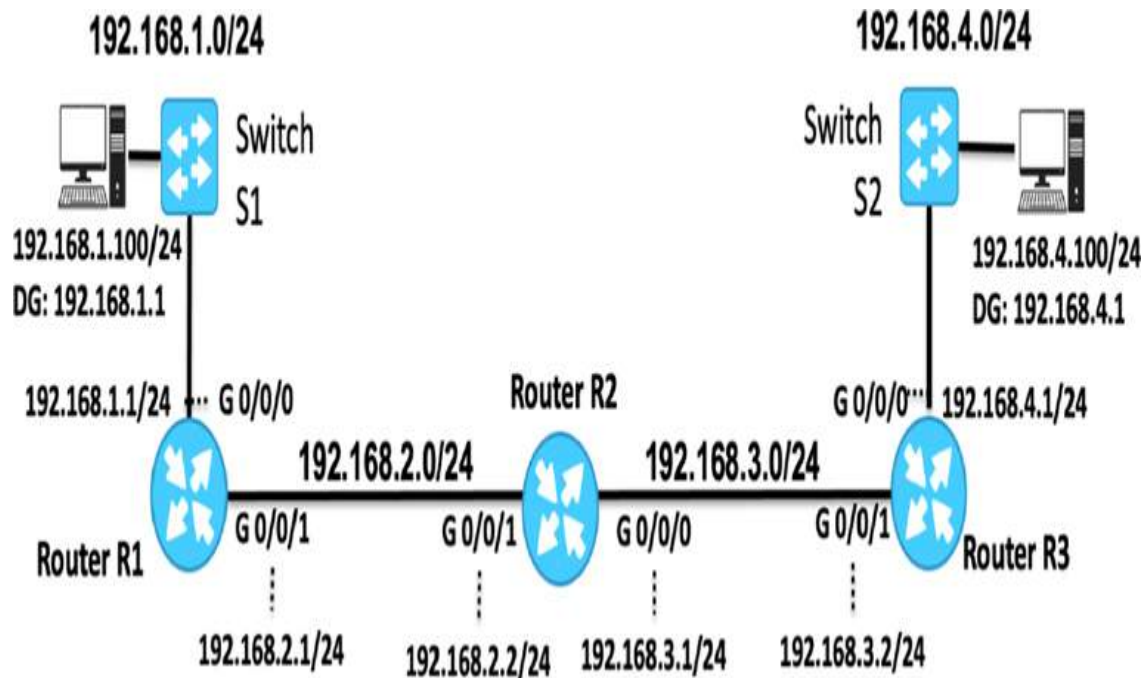


Figure 4-1 *Topology with IPv4 addressing*

The code in [Example 4-7](#) demonstrates the use of a **for** loop to efficiently retrieve routing table information from multiple routers using Python and Netmiko. This **for** loop also allows you to easily accommodate additional network devices simply by adding more IP addresses to the list.

Example 4-7 *ex4-7_send_command_loop.py*

```
# Import the Netmiko Library
import netmiko

# Assign IP address of the device to a variable
device_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']

# Loop through IP addresses
# Assign device to each item in the list of devices (device_list)
for device in device_list:

    # Establish an SSH connection to the device
```

```

connection = netmiko.ConnectHandler(
    ip=device,      # Assign the IP address using
    device_type='cisco_ios',
    username='admin',
    password='cisco',
    secret='spot'   # Enable password if required
)

# Display the IPv4 routing table
# send a command to the device and print the output
print("\nIPv4 routing table for router with the IP address: ")
print(connection.send_command('show ip route'))

# Close the SSH connection for this device
connection.disconnect()

print("All devices processed.")

```

[Example 4-8](#) shows the output of this program. Notice that this program displays each of the IP addresses in the Python list **ip_list**.

Example 4-8 *Output from [Example 4-7](#)*

```

IPv4 routing table for router with the IP address: 192.168.1.1

Codes: L - local, C - connected, S - static, R - RIP, M - mobile
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 -
       ia - IS-IS inter area, * - candidate default, U - per-user
       o - ODR, P - periodic downloaded static route, H - NHRP,

```

a - application route

+ - replicated route, % - next hop override, p - override

Gateway of last resort is not set

192.168.1.0/24 is variably subnetted, 2 subnets, 2 masks

C 192.168.1.0/24 is directly connected, GigabitEthernet0/

L 192.168.1.1/32 is directly connected, GigabitEthernet0/

192.168.2.0/24 is variably subnetted, 2 subnets, 2 masks

C 192.168.2.0/24 is directly connected, GigabitEthernet0/

L 192.168.2.1/32 is directly connected, GigabitEthernet0/

S 192.168.3.0/24 [1/0] via 192.168.2.2

S 192.168.4.0/24 [1/0] via 192.168.2.2

IPv4 routing table for router with the IP address: 192.168.2.2

Codes: L - local, C - connected, S - static, R - RIP, M - mobile
<output omitted>

Gateway of last resort is not set

S 192.168.1.0/24 [1/0] via 192.168.2.1

192.168.2.0/24 is variably subnetted, 2 subnets, 2 masks

C 192.168.2.0/24 is directly connected, GigabitEthernet0/

L 192.168.2.2/32 is directly connected, GigabitEthernet0/

192.168.3.0/24 is variably subnetted, 2 subnets, 2 masks

C 192.168.3.0/24 is directly connected, GigabitEthernet0/

L 192.168.3.1/32 is directly connected, GigabitEthernet0/

S 192.168.4.0/24 [1/0] via 192.168.3.2

IPv4 routing table for router with the IP address: 192.168.3.2

Codes: L - local, C - connected, S - static, R - RIP, M - mobile

```
<output omitted>
```

```
Gateway of last resort is not set
```

```
S      192.168.1.0/24 [1/0] via 192.168.3.1
```

```
S      192.168.2.0/24 [1/0] via 192.168.3.1
```

```
      192.168.3.0/24 is variably subnetted, 2 subnets, 2 masks
```

```
C      192.168.3.0/24 is directly connected, GigabitEthernet0,
```

```
L      192.168.3.2/32 is directly connected, GigabitEthernet0,
```

```
      192.168.4.0/24 is variably subnetted, 2 subnets, 2 masks
```

```
C      192.168.4.0/24 is directly connected, GigabitEthernet0,
```

```
L      192.168.4.1/32 is directly connected, GigabitEthernet0,
```

```
All devices processed.
```

Using a for Loop with Netmiko, Step-by-Step

If you are already familiar with Python **for** loops, the program in [Examples 4-7](#) and [4-8](#) will be self-explanatory. However, for those who are still be new to Python, here is an explanation of how this *for* loop works.

Step 1: Define the List of Device IP Addresses

First, you define a Python list named **device_list**, which contains the IP addresses of the routers you want to connect to:

```
device_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']
```

Each address represents a different router in the network topology. This allows you to iterate through the list and execute commands on multiple devices without duplicating code.

Step 2: Start the for Loop to Iterate over Devices

This **for** loop processes each IP address in **device_list**:

```
for device in device_list:
```

The variable **device** is assigned one IP address at a time, starting with **192.168.1.1**, then **192.168.2.2**, and finally **192.168.3.2**. Each iteration of the loop will establish an SSH connection to the current device, execute commands on that device, and then close the connection.

Step 3: Display the IPv4 Routing Table

Before executing a command, the program displays a message to indicate the IP address of the device—the router—that is being processed:

```
print("\nIPv4 routing table for router with the IP address:", device_ip)
print(connection.send_command('show ip route'))
```

The next statement uses the **send_command()** method to send the **show ip route** command to retrieve and display the router's IPv4 routing table.

This is where you can take advantage of your knowledge of IOS and Python. For example, in Python:

- If you know Python conditionals (**if-elif-else**), you can execute different commands on different devices.
- If you're familiar with file handling, you could save each router's output to a file for documentation or retrieve a file with configuration commands.

But this book is not a Python tutorial; it is a beginner-friendly introduction to Netmiko. If you are already

comfortable with Python, you no doubt already see how to move forward. If you're new to Python, focus on understanding Netmiko first before diving into advanced Python concepts.

Example: Using a for Loop to Configure IPv6 Addresses

Transitioning to IPv6 is a great use case for network automation because it allows you to configure IPv6 settings remotely while maintaining access over IPv4. Because these devices are already reachable via IPv4 SSH connections, you can automate the implementation of IPv6 addressing and routing and other IPv6 configurations over IPv4 to make the transition faster and more efficient.

Figure 4-2 shows the same topology we used earlier, now with the addition of IPv6 addressing.

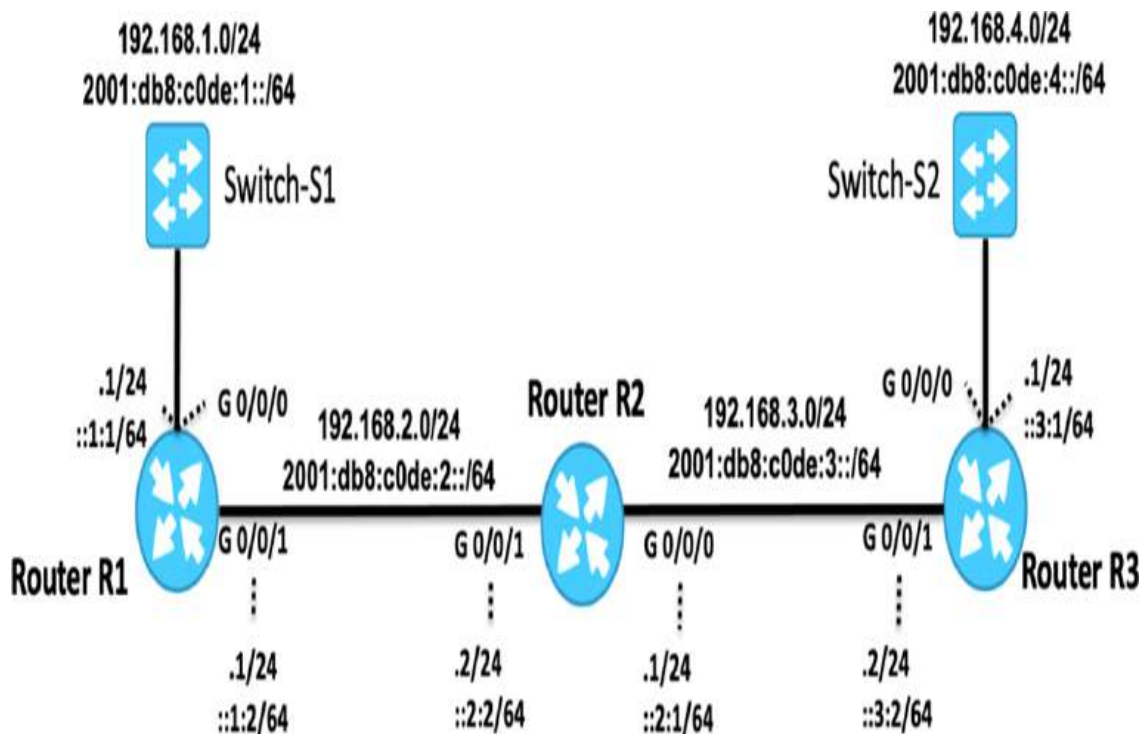


Figure 4-2 Topology with IPv4 and IPv6 Addressing

[Example 4-9](#) uses a **for** loop to configure IPv6 addresses on three routers. Each router has a predefined set of IPv6 configuration commands, and the program applies them dynamically, based on the device's IP address.

Note

In programming, there are often multiple ways to achieve the same result; some may be more efficient, others more elegant. However, the goal of this book is to keep the code simple and straightforward to ensure that readers of all experience levels, including those who are new to Python, can follow along and apply these concepts effectively. Our focus is on clarity and practicality rather than on optimizing for the most advanced or complex solutions.

Example 4-9 *ex4-9_for_loop_ipv6.py*

```
# Import the Netmiko Library
import netmiko

# List of interface configuration commands for each router
r1_ipv6_addressing = [
    'ipv6 unicast-routing',    # Enable IPv6 routing

    # Configure IPv6 addresses on GigabitEthernet0/0/0
    'interface gig 0/0/0',
    'ipv6 address 2001:db8:c0de:1::1/64', # Assign global IPv6 address
    'ipv6 address fe80::1:1 link-local',  # Assign link-local address
    'exit',

    # Configure IPv6 addresses on GigabitEthernet0/0/1
```

```
    'interface gig 0/0/1',
    'ipv6 address 2001:db8:c0de:2::1/64', # Assign global IPv6 a
    'ipv6 address fe80::1:2 link-local', # Assign link-local ad
    'exit'
]
```

```
r2_ipv6_addressing = [
    'ipv6 unicast-routing', # Enable IPv6 routing

    # Configure IPv6 addresses on GigabitEthernet0/0/0
    'interface gig 0/0/0',
    'ipv6 address 2001:db8:c0de:3::1/64', # Assign global IPv6 a
    'ipv6 address fe80::2:1 link-local', # Assign link-local ad
    'exit',

    # Configure IPv6 addresses on GigabitEthernet0/0/1
    'interface gig 0/0/1',
    'ipv6 address 2001:db8:c0de:2::2/64', # Assign global IPv6 a
    'ipv6 address fe80::2:2 link-local', # Assign link-local ad
    'exit'
]
```

```
r3_ipv6_addressing = [
    'ipv6 unicast-routing', # Enable IPv6 routing

    # Configure IPv6 addresses on GigabitEthernet0/0/0
    'interface gig 0/0/0',
    'ipv6 address 2001:db8:c0de:4::1/64', # Assign global IPv6 a
    'ipv6 address fe80::3:1 link-local', # Assign link-local ad
    'exit',

    # Configure IPv6 addresses on GigabitEthernet0/0/1
    'interface gig 0/0/1',
```

```

        'ipv6 address 2001:db8:c0de:3::3/64', # Assign global IPv6 address
        'ipv6 address fe80::3:2 link-local', # Assign link-local address
        'exit'
    ]

# List of router management IP addresses
device_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']

# Loop through each router in the list and apply the correct configuration
for device in device_list:

    # Establish an SSH connection to the device
    connection = netmiko.ConnectHandler(
        ip=device,      # Assign the IP address using the device_list
        device_type='cisco_ios',
        username='admin',
        password='cisco',
        secret='spot'   # Enable password if required
    )

    # Enter privileged EXEC mode (required for making configuration changes)
    connection.enable()

    # Apply the appropriate IPv6 configuration commands based on the
    # router's IP address
    if device == '192.168.1.1':
        print("\nConfiguring device:",device)
        connection.send_config_set(r1_ipv6_addressing) # Apply configuration
        print(connection.send_command('show ipv6 interface brief'))

    elif device == '192.168.2.2':
        print("\nConfiguring device:",device)
        connection.send_config_set(r2_ipv6_addressing) # Apply configuration

```

```

        print(connection.send_command('show ipv6 interface brief'))

    elif device == '192.168.3.2':
        print("\nConfiguring device:",device)
        connection.send_config_set(r3_ipv6_addressing) # Apply IPv6 configuration
        print(connection.send_command('show ipv6 interface brief'))

    else:
        print("Mismatch of device address and list")

# Close the SSH connection for this device
connection.disconnect()

print("All devices processed.")

```

This program uses a **for** loop to iterate through the list of three router IP addresses stored in **device_list**. For each router, the program establishes an SSH connection using Netmiko and applies the appropriate IPv6 configuration commands before disconnecting.

The IPv6 link-local address can be the same on all interfaces of the router, as long it is unique on the link (that is, on the IPv6 network). In [Example 4-9](#), we follow a particular naming convention for the interface ID portion of the address:

- The first digit identifies the router (for example, 1 for R1, 2 for R2).
- The second digit indicates the interface number on that router (for example, 1 for Gig0/0/0, 2 for Gig0/0/1).

So, for example, R1's Gig0/0/0 interface is given the link-local address fe80::1:1, and Gig0/0/1 is fe80::1:2.

Inside the loop, an **if-elif** structure is used to match the router's IP address to one of three predefined lists of configuration commands (**r1_ipv6_addressing**, **r2_ipv6_addressing**, and **r3_ipv6_addressing**). When a match is found, the program executes **send_config_set()** to apply the corresponding list of IPv6 commands for that router. After applying the configuration, the program runs the **show ipv6 interface brief** command to verify the changes.

Note

The **if-elif-else** statements allow a program to evaluate multiple conditions sequentially. The **if** block runs if its condition is true; if the condition is not true, the **elif** (short for "else if") checks additional conditions. If none of the conditions are met, the **else** block executes as a fallback.

Once the configuration is complete for a device, the SSH connection is closed, and the loop moves to the next router in **device_list**. This process continues until all devices in the list have been processed. When there are no more devices left, the loop terminates, and the program ends.

[Example 4-10](#) shows the output from the code in [Example 4-9](#).

Example 4-10 Output from [Example 4-9](#)

```
MyPrompt% python3 ex4-9_for_loop_ipv6.py
```

```
Configuring device: 192.168.1.1
```

```
GigabitEthernet0/0/0    [up/up]
```

```
FE80::1:1
2001:DB8:C0DE:1::1
GigabitEthernet0/0/1    [up/up]
FE80::1:2
2001:DB8:C0DE:2::1
GigabitEthernet0/0/2    [administratively down/down]
unassigned
GigabitEthernet0        [administratively down/down]
unassigned

Configuring device: 192.168.2.2
GigabitEthernet0/0/0    [up/up]
FE80::2:1
2001:DB8:C0DE:3::1
GigabitEthernet0/0/1    [up/up]
FE80::2:2
2001:DB8:C0DE:2::2
GigabitEthernet0/0/2    [administratively down/down]
unassigned
GigabitEthernet0        [administratively down/down]
unassigned

Configuring device: 192.168.3.2
GigabitEthernet0/0/0    [up/up]
FE80::3:1
2001:DB8:C0DE:4::1
GigabitEthernet0/0/1    [up/up]
FE80::3:2
2001:DB8:C0DE:3::2
GigabitEthernet0/0/2    [administratively down/down]
unassigned
GigabitEthernet0        [administratively down/down]
unassigned
```



```
All devices processed.  
MyPrompt%
```

Using Dictionaries to Store Device Connection Parameters

When automating network tasks, it is important to manage device connection details efficiently. Instead of storing connection parameters separately or repeating them for each device, you can use dictionaries to organize this information in a structured way. Each device's connection settings, such as IP address, username, password, and device type, are stored in the device's own dictionary. All the dictionaries are then placed inside a list, and you can iterate through them by using a **for** loop. The process is called *dictionary unpacking*.

In this section, we are using dictionary unpacking (****device**) to pass all key/value pairs as keyword arguments to Netmiko's **ConnectHandler()**. This makes the code cleaner, scalable, and easier to modify, especially when managing a large number of devices with different credentials or connection settings. The code in [Example 4-11](#) demonstrates how this method simplifies connecting to multiple devices and executing commands. Notice the different credentials required for each of the routers.

Example 4-11 **ex4-11_connection_parameters_dictionary.py**

```
# Import the Netmiko Library  
import netmiko  
  
# Define device dictionaries containing ConnectHandler paramete
```

```
r1 = {
    'ip': '192.168.1.1',    # IP address of Router 1
    'device_type': 'cisco_ios', # Device type for Netmiko
    'username': 'admin',    # SSH username
    'password': 'cisco',    # SSH password
    'secret': 'spot'        # Enable mode password
}

r2 = {
    'ip': '192.168.2.2',    # IP address of Router 2
    'device_type': 'cisco_ios',
    'username': 'rick',
    'password': 'ucsc',
    'secret': 'cabrillo'
}

r3 = {
    'ip': '192.168.3.2',    # IP address of Router 3
    'device_type': 'cisco_ios',
    'username': 'adrian',
    'password': 'cisco',
    'secret': 'devnet'
}

# List of devices - each dictionary represents a
# router's connection parameters
devices = [r1, r2, r3]

print('\n')

# Iterate over each device in the list
for device in devices:
```

```
# Use dictionary unpacking (**) to pass key-value
# pairs as keyword arguments to ConnectHandler
connection = netmiko.ConnectHandler(**device)

# Print the IP address of the connected device
print("Device IP address:", device['ip'])
print('-' * 35) # Print 35 dashes for readability

# Send the 'show ip interface brief' command and display the
output = connection.send_command('show ip interface brief')
print(output)

print('\n')
connection.disconnect()
```

In this program, each device's connection details—including its IP address, username, password, and device type—are stored in a dictionary. Dictionaries **r1**, **r2**, and **r3** contain all the information needed to establish an SSH connection using Netmiko. Instead of manually using separate connection code for each device, the program stores all the dictionaries in a list called **devices**. The program then goes through the list one device at a time, retrieving its connection details and using them to establish a connection.

As a result, the output, as shown in [Example 4-12](#), displays the interface status and IP configurations for each router.

Example 4-12 *Output from [Example 4-11](#)*

```
MyPrompt% python3 ex4-11_connection_parameters_dictionary.py
```

```
Device IP address: 192.168.1.1
```

```

-----
Interface          IP-Address      OK? Method Status
GigabitEthernet0/0/0 192.168.1.1    YES NVRAM  up
GigabitEthernet0/0/1 192.168.2.1    YES NVRAM  up
GigabitEthernet0/0/2 unassigned     YES NVRAM  administrative
GigabitEthernet0     unassigned     YES NVRAM  administrative

```

Device IP address: 192.168.2.2

```

-----
Interface          IP-Address      OK? Method Status
GigabitEthernet0/0/0 192.168.3.1    YES NVRAM  up
GigabitEthernet0/0/1 192.168.2.2    YES NVRAM  up
GigabitEthernet0/0/2 unassigned     YES NVRAM  administrative
GigabitEthernet0     unassigned     YES NVRAM  administrative

```

Device IP address: 192.168.3.2

```

-----
Interface          IP-Address      OK? Method Status
GigabitEthernet0/0/0 192.168.4.1    YES NVRAM  up
GigabitEthernet0/0/1 192.168.3.2    YES NVRAM  up
GigabitEthernet0/0/2 unassigned     YES NVRAM  administrative
GigabitEthernet0     unassigned     YES NVRAM  administrative

```

MyPrompt%

Understanding Dictionary Unpacking in Python (**device)

For those who are new to Python, dictionary unpacking may seem a little confusing. Although you don't need to understand it to continue with the rest of this book, in case you are interested, we discuss it more here. In Python, dictionary unpacking allows you to pass key/value pairs from a dictionary as separate keyword arguments to a function.

Note

Python dictionaries are explained in [Chapter 5](#), “Introducing NAPALM and Structured Data.”

Take a look at [Example 4-13](#). It shows an example of a Python dictionary with several key/value pairs, including 'ip': '192.168.1.1' and 'device_type': 'cisco_ios'.

Example 4-13 *r1 Connection Parameters as a Dictionary*

```
r1 = {  
    'ip': '192.168.1.1',  
    'device_type': 'cisco_ios',  
    'username': 'admin',  
    'password': 'cisco',  
    'secret': 'spot'  
}  
  
connection = netmiko.ConnectHandler(**r1)
```

When you use dictionary unpacking (for example, ****r1**), Python expands the dictionary into keyword arguments, making it equivalent to the code shown in [Example 4-14](#).

Example 4-14 *r1* Connection Parameters in **ConnectHandler**

```
connection = netmiko.ConnectHandler(  
    ip='192.168.1.1',  
    device_type='cisco_ios',  
    username='admin',  
    password='cisco',  
    secret='spot'  
)
```

Instead of manually specifying each parameter, ****r1** in [Example 4-13](#) automatically expands (or *unpacks*) the dictionary into individual arguments, making the code cleaner and more scalable.

Dictionary unpacking works well when you're dealing with multiple devices because you can store each device's connection details in a dictionary and iterate over a list of dictionaries, passing them into **ConnectHandler()** dynamically.

In this section, we have looked at dictionary unpacking using **r1** as an example. In the program shown in [Example 4-11](#), this concept is applied dynamically within a **for** loop, where **device** represents each dictionary (**r1**, **r2**, **r3**) as the loop goes through the list. When you use **connection = netmiko.ConnectHandler(**device)**, the **device** variable refers to the current dictionary being processed, allowing Netmiko to extract the necessary connection details for each router automatically.

Extracting and Displaying the Device IP Address

Another part of the code in [Example 4-11](#) that may need some explanation is **print("Device IP address:", device['ip'])**.

When executing **print("Device IP address:", device['ip'])**, the program retrieves and displays the IP address of the current device being processed in the **for** loop. Here, **device** is a dictionary that contains connection details such as IP address, username, password, and device type. By using **device['ip']**, you extract only the IP address from the dictionary and print it after the string "Device IP address:".

For example, if **device** contains **{'ip': '192.168.1.1', 'device_type': 'cisco_ios', ...}**, then **device['ip']** will return **'192.168.1.1'**, and the output will be **Device IP address: 192.168.1.1**.

Again, if you are new to Python and you find this a bit confusing, don't worry about it. It is not imperative to understanding this code in order to continue reading this book.

An Alternative Way to Store Device Dictionaries

The code in [Example 4-15](#) demonstrates an alternative way to store device connection details and is especially for those who are a bit more familiar with Python. The program defines each device's parameters directly inside the list instead of creating separate dictionary variables. This program functions exactly the same as the code in [Example 4-11](#), **ex4-11_connection_parameters_dictionary.py**,

but this approach keeps all device details together in a single structure.

Example 4-15 *ex4-15_connection_parameters_list-dictionary.py*

```
# Import the Netmiko Library
import netmiko

# Define a list of dictionaries, each containing connection
# details for a router
devices = [

{
    'ip': '192.168.1.1',    # IP address of Router 1
    'device_type': 'cisco_ios', # Device type for Netmiko
    'username': 'admin',    # SSH username
    'password': 'cisco',    # SSH password
    'secret': 'spot'       # Enable mode password
},

{
    'ip': '192.168.2.2',    # IP address of Router 2
    'device_type': 'cisco_ios',
    'username': 'rick',
    'password': 'ucsc',
    'secret': 'cabrillo'
},

{
    'ip': '192.168.3.2',    # IP address of Router 3
    'device_type': 'cisco_ios',
    'username': 'adrian',
    'password': 'cisco',
```



```

        'secret': 'devnet'
    }

]

# Loop through each device in the list and establish an SSH connection
for device in devices:

    # Use dictionary unpacking (**) to pass key-value pairs from device
    # as keyword arguments to Netmiko's ConnectHandler function
    connection = netmiko.ConnectHandler(**device)

    # Print the IP address of the connected device
    print("Device IP address:", device['ip'])
    print('-' * 35) # Print 35 dashes for readability

    # Send the 'show ip interface brief' command and display the output
    output = connection.send_command('show ip interface brief')
    print(output)

    print('\n')
    connection.disconnect()

```

This approach stores each device's **connection** parameters in a dictionary directly inside the list instead of creating separate dictionary variables (**r1**, **r2**, **r3**). The **devices** variable is a list of dictionaries, where each dictionary contains the connection details for a specific router. The **for** loop then iterates through the list, processing one dictionary at a time. This allows the program to retrieve each device's parameters dynamically and pass them to **ConnectHandler()**, making the code more compact and structured while still maintaining the same functionality.

The output of the code in [Example 4-15](#) would be exactly the same as the for the program **ex4-11_connection_parameters_dictionary.py**, shown in [Example 4-12](#).

Using Python's `getpass()` for Secure Password Input

When automating network tasks, storing passwords directly in a program is not a secure practice. Instead, Python provides the **getpass** module, which allows users to enter sensitive information, such as SSH passwords and enable secrets, without displaying them on the screen. This helps protect credentials from being exposed when running the program, which is especially important in shared or public environments.

In the program in [Example 4-16](#), **getpass()** is used to securely collect the SSH password and enable secret password before establishing connections to multiple devices. Unlike the **input()** function, which displays the typed characters, **getpass()** hides the password input, preventing unauthorized users from seeing the credentials. The username, however, is entered using **input()** because usernames are not typically considered confidential.

This program assumes that all routers share the same username, password, and enable secret, which is the most common scenario in many network environments. However, in cases where devices have different credentials, additional logic would be needed to store and assign credentials individually.

Once the credentials are collected, the program iterates through **device_list** and establishes an SSH connection to each device by using **ConnectHandler()**. The program then

runs the **show ip interface brief** command on each device, prints the output, and closes the connection. This method ensures that user credentials do not need to be hardcoded into the program, improving security while maintaining functionality.

Example 4-16 *ex4-16_get_pass.py*

```
# Import the Netmiko Library
import netmiko

# Import getpass to securely collect passwords
# without displaying them on the screen
from getpass import getpass

# Prompt user for SSH credentials
# Username is displayed in clear text
username_entered = input('Enter SSH username: ')

# Password is hidden when entered
password_entered = getpass('Enter SSH password: ')

# Enable secret password is also hidden
secret_entered = getpass('Enter enable secret password: ')

# List of device IP addresses (assumes all devices share the same)
device_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']

# Loop through each device in the list and establish an SSH connection
for device in device_list:

    # Use the collected credentials to connect to the current device
    connection = netmiko.ConnectHandler(
        ip=device,      # Assign the IP address dynamically
        device_type='cisco_ios',
```

```

        username=username_entered, # Use the entered username
        password=password_entered, # Use the entered password
        secret=secret_entered # Use the entered enable password
    )

    # Execute and display the output of the 'show ip interface brief' command
    print("\n'show ip interface brief' for:", device, "\n")
    print(connection.send_command('show ip interface brief'))

    # Close the SSH connection for this device
    connection.disconnect()

print("All devices processed.")

```

Note

The **getpass()** function can be encapsulated within another function to prompt for SSH credentials securely. This allows for reusable, structured authentication handling within a program. Don't worry about this if you are still learning Python; the **get_pass()** code in [Example 4-16](#) does exactly what we need it to do.

[Example 4-17](#) shows the output of this program. The **show ip interface brief** command executed on three routers (192.168.1.1, 192.168.2.2, and 192.168.3.2), all accessed using the same credentials. Notice that **get_pass()** has prevented the passwords from being displayed.

Example 4-17 Output from [Example 4-16](#)

```

MyPrompt% python3 ex4-16_get_pass.py
Enter SSH username: admin
Enter SSH password:

```

Enter enable secret password:

'show ip interface brief' for: 192.168.1.1

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.1.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.1	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

'show ip interface brief' for: 192.168.2.2

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.3.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.2	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

'show ip interface brief' for: 192.168.3.2

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.4.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.3.2	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

devices processed.

MyPrompt%

Using Netmiko Exceptions for Troubleshooting

When working with network automation, connection issues are inevitable. Common problems such as incorrect credentials, unreachable devices, and timeout errors can prevent successful SSH connections. Netmiko provides built-in exceptions that help identify and troubleshoot these issues more effectively. By wrapping a connection attempt in a **try-except** block, you can catch specific errors—such as timeouts, authentication failures, and read timeouts—and allow the program to respond appropriately instead of crashing.

Although this approach is not required, it makes debugging easier and also ensures that the program can continue running even if one or more devices encounter issues. The code in [Example 4-18](#) demonstrates how to handle common Netmiko exceptions and provides guidance on resolving each type of error. Notice that the second router in the list has the incorrect IPv4 address **192.168.20.1**.

Example 4-18 *ex4-18_exceptions.py*

```
# Import the Netmiko Library
import netmiko

# List of device IP addresses
device_list = ['192.168.1.1', '192.168.20.1', '192.168.3.2']

# Loop through each device in the list and attempt an SSH connection
for device in device_list:
    try:
        # Attempt to establish an SSH connection to the device
        connection = netmiko.ConnectHandler(
            ip=device,          # Assign IP address
            device_type='cisco_ios', # Specify the device type
```

```

        username='admin', # SSH username
        password='cisco', # SSH password
        secret='spot'      # Enable password if required
    )

    # Send the 'show ip interface brief' command and
    # display the output
    print("\nInterface information for:", device, "\n")
    print(connection.send_command('show ip interface brief'))

    # Close the SSH connection for this device
    connection.disconnect()

# Handle connection timeout errors (e.g., unreachable device,
# firewall blocking access)
except netmiko.exceptions.NetmikoTimeoutException:
    print('\nTimeout occurred to', device)
    print(''Common causes of this problem are:
    1. Incorrect hostname or IP address.
    2. Wrong TCP port.
    3. Intermediate firewall blocking access.'')
    print('\n')

# Handle authentication errors (e.g., wrong username/password)
except netmiko.exceptions.NetMikoAuthenticationException:
    print('\nAuthentication error', device)
    print(''Common causes of this problem are:
    1. Invalid username and password
    2. Incorrect SSH-key file
    3. Connecting to the wrong device'')
    print('\n')

# Handle read timeout errors (e.g., device prompt not detected)

```

```

except netmiko.exceptions.ReadTimeout:
    print('\nRead timeout. pattern not detected', device)
    print(''Common causes of this problem are:
    1. Missing or incorrect secret password in ConnectHandler
    2. Adjust the regex pattern to better identify the term:
        string. Note, in many situations the pattern is
        automatically based on the network device's prompt.
    3. Increase the read_timeout to a larger value.'')
    print('\n')

# Catch-all exception handler for any other unexpected error
except Exception as e:
    print("An error occurred:", str(e))

```

In Python, the **try-except** structure is used to handle errors gracefully instead of letting the program crash. The **try** block contains the code that might cause an error, such as establishing an SSH connection to a network device. If an error occurs, Python immediately stops executing the **try** block and looks for a matching **except** block to handle the specific error.

In this way, the program can catch and respond to different types of errors, such as timeouts, authentication failures, and read timeouts, allowing the program to continue running even if one or more devices encounter problems.

The three exceptions used in this program are among the most common Netmiko exceptions encountered when automating network devices. Connection timeouts, authentication failures, and read timeouts are the most common issues when dealing with SSH access to multiple devices.

Here is a brief description of each exception in [Example 4-18](#):

- **netmiko.exceptions.NetmikoTimeoutException:** This error occurs when a device is unreachable, typically due to an incorrect IP address, firewall restrictions, or connectivity issues.
- **netmiko.exceptions.NetMikoAuthenticationException:** This error is raised when authentication fails, usually due to an incorrect username, password, or SSH key.
- **netmiko.exceptions.ReadTimeout:** This exception occurs when the expected command prompt is not detected, often due to a missing enable password, a slow device response, or an incorrect prompt pattern.

Netmiko provides additional exceptions that can help with troubleshooting more specific issues:

- **netmiko.ssh_exception.ConfigInvalidException:** This error is raised when a configuration command fails, typically due to syntax errors or an unsupported command.
- **netmiko.ssh_exception.SSHException:** This generic SSH-related error is often caused by issues like an unsupported cipher, an incompatible SSH key exchange algorithm, or a corrupted SSH session.
- **netmiko.ssh_exception.SessionDownException:** This error is raised when the SSH session unexpectedly closes during execution.
- **netmiko.ssh_exception.IncompleteReadException:** This exception occurs when an incomplete response is

received from a device, usually due to a slow or unresponsive device.

In [Example 4-18](#), notice that the second router in the list has the IPv4 address **192.168.20.1**. This is an incorrect address that should be 192.168.2.1.

[Example 4-19](#) shows what happens when you run this program.

Example 4-19 *Output from [Example 4-18](#)*

```
MyPrompt% python3 ex4-18_exceptions.py
```

```
Interface information for: 192.168.1.1
```

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.1.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.1	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

```
Timeout occurred to 192.168.20.1
```

```
Common causes of this problem are:
```

1. Incorrect hostname or IP address.
2. Wrong TCP port.
3. Intermediate firewall blocking access.

```
Interface information for: 192.168.3.2
```

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.4.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.3.2	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative

```
GigabitEthernet0      unassigned      YES NVRAM  administrative
MyPrompt%
```

The output in [Example 4-19](#) shows the results of executing the **show ip interface brief** command on three routers. As you can see, a timeout error occurred for **192.168.20.1**, which is the incorrect IP address. The program correctly handled this error using **netmiko.exceptions.NetmikoTimeoutException** and displayed a message explaining possible causes, such as an incorrect IP address, TCP connectivity issues, or firewall restrictions. Despite this failure, the program continued running and successfully retrieved interface details for the remaining devices, demonstrating the benefit of using exception handling to prevent a single error from stopping the entire automation process.

Maintaining Multiple SSH Connections Simultaneously

This section is for those who are much more comfortable with Python and are curious about having multiple SSH connections simultaneously with Netmiko. Although it is possible to use multiple SSH connections simultaneously with Netmiko, it is important to be aware that when connecting to a large number of devices at the same time, maintaining multiple open SSH connections can significantly increase system resource usage, including memory and file descriptors. In most cases, it's better to establish a single connection, run the commands, and then disconnect before establishing the next connection.

Note

For large scenarios that require parallel execution, tools such as Threads or AsyncIO (or libraries like Nornir) would be more appropriate.

The program in [Example 4-20](#) demonstrates how to manage multiple simultaneous SSH connections using Netmiko. This is a significant shift from the more common approach where a program connects to one device at a time, executes a command, and disconnects before moving on to the next device. Instead of handling devices sequentially, this program establishes connections to all devices first and then executes commands on all the connected devices; then it disconnects from all of the devices. This approach can improve efficiency, especially when working with a large number of network devices, by reducing the time spent repeatedly opening and closing SSH sessions.

Example 4-20 *ex4-20_mutiple_ssh_connections.py*

```
# Import the Netmiko Library
import netmiko

# Assign IP addresses of the devices
device_list = ['192.168.1.1', '192.168.2.2', '192.168.3.2']

# Dictionary to store active connections
connections = {}

# Establish SSH connections to all devices simultaneously
for device in device_list:
    connections[device] = netmiko.ConnectHandler(
        ip=device,
        device_type='cisco_ios',
        username='admin',
        password='cisco',
```

```
        secret='spot' # Enable password if required
    )
    print(f"\nConnected to {device}")

# Execute a command on all devices while all connections remain
for device, connection in connections.items():
    print(f"\nIPv4 routing table for router with IP address {device}")
    print(connection.send_command('show ip route'))

# Close all SSH connections
for device, connection in connections.items():
    connection.disconnect()
    print(f"\nDisconnected from {device}")

print("\nAll devices processed.")
```

Here is a brief overview of how this program uses simultaneous SSH connections:

1. A dictionary (**connections**) is used to store SSH connection objects.
2. The first loop iterates through **device_list**, creates a connection for each device, and stores it in the **connections** dictionary.
3. The second loop iterates through **connections.items()** and executes the command while all SSH sessions remain open.
4. The third loop closes all active SSH connections.

For those who are more familiar with Python and want to get a little deeper, here is a more detailed look at what this program is doing. The program uses a list of device IP addresses (**device_list**) and iterates through them using a

first **for** loop, where an SSH connection is established for each device. These connections are stored in a dictionary (**connections**), using the device's IP address as the key and the **ConnectHandler()** object as the value.

This is the **connections** dictionary after the first **for** loop is completed:

```
{  
'192.168.1.1': <netmiko.cisco.cisco_ios.CiscoIosSSH object at 0x10...>,  
'192.168.2.2': <netmiko.cisco.cisco_ios.CiscoIosSSH object at 0x11...>,  
'192.168.3.2': <netmiko.cisco.cisco_ios.CiscoIosSSH object at 0x11...>  
}
```

The **connection** object in Netmiko, represented as **<netmiko.cisco.cisco_ios.CiscoIosSSH object at 0x...>**, is an instance of the **ConnectHandler** class that maintains an active SSH session with a network device. This object allows the program to use Netmiko methods, prefaced by the connection object, to send commands (such as **connection.send_command()**), enter configuration mode (such as **connection.send_config_set()**), and manage the connection (such as **connection.disconnect()**).

In the program, these connection objects are stored in a dictionary, **connections**, with device IP addresses as keys. This structure allows the program to maintain open SSH sessions to multiple devices at the same time rather than repeatedly opening and closing connections in each iteration of the loop.

The second **for** loop is responsible for executing commands on all active connections. Instead of opening a new SSH session each time, the program retrieves the existing connection from the **connections** dictionary and runs the **show ip interface brief** command. This separation between connection management and command execution

allows the program to be more flexible, enabling users to run multiple commands on all devices without needing to reconnect. This method also makes it possible to introduce logic that dynamically selects commands based on the device type or state.

Finally, the third **for** loop ensures that all active SSH connections are properly closed. This is an essential step in preventing excessive open sessions that could consume system resources or lead to authentication issues on network devices. By iterating through the **connections** dictionary and calling **disconnect()**, the program ensures that no SSH sessions are left open after all devices are processed.

This structure—first establishing all connections, then executing commands, and finally closing the connections—provides a more scalable and efficient approach than the traditional method of handling one device at a time. You can see from this example how separating different stages of network automation can improve the performance and flexibility of a program. If you are new to Python, all you need to know at this point is that the three-loop structure helps clarify how programs can manage multiple network devices concurrently while maintaining an organized and systematic flow.

The output in [Example 4-21](#) demonstrates the process of establishing multiple SSH connections to network devices, executing commands, and properly closing the sessions. First, the program connects to each router in sequence and displays a confirmation message as each connection is established. Once all devices are connected, the program retrieves and prints the IPv4 routing table for each router while maintaining the SSH sessions open. Finally, the program iterates through the connections again, closing each session and confirming disconnection.

Example 4-21 Output from [Example 4-20](#)

```
MyPrompt% python3 ex4-20_multiple_ssh_connections.py
```

```
Connected to 192.168.1.1
```

```
Connected to 192.168.2.2
```

```
Connected to 192.168.3.2
```

```
IPv4 routing table for router with IP address 192.168.1.1:
```

```
Codes: L - local, C - connected, S - static, R - RIP, M - mobile  
        D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter  
        <output omitted for brevity>
```

```
Gateway of last resort is not set
```

```
      192.168.1.0/24 is variably subnetted, 2 subnets, 2 masks  
C       192.168.1.0/24 is directly connected, GigabitEthernet0/  
L       192.168.1.1/32 is directly connected, GigabitEthernet0/  
      192.168.2.0/24 is variably subnetted, 2 subnets, 2 masks  
C       192.168.2.0/24 is directly connected, GigabitEthernet0/  
L       192.168.2.1/32 is directly connected, GigabitEthernet0/  
S       192.168.3.0/24 [1/0] via 192.168.2.2  
S       192.168.4.0/24 [1/0] via 192.168.2.2
```

```
IPv4 routing table for router with IP address 192.168.2.2:
```

```
Codes: L - local, C - connected, S - static, R - RIP, M - mobile  
        D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter  
        <output omitted for brevity>
```


Gateway of last resort is not set

```
S      192.168.1.0/24 [1/0] via 192.168.2.1
      192.168.2.0/24 is variably subnetted, 2 subnets, 2 masks
C      192.168.2.0/24 is directly connected, GigabitEthernet0/
L      192.168.2.2/32 is directly connected, GigabitEthernet0/
      192.168.3.0/24 is variably subnetted, 2 subnets, 2 masks
C      192.168.3.0/24 is directly connected, GigabitEthernet0/
L      192.168.3.1/32 is directly connected, GigabitEthernet0/
S      192.168.4.0/24 [1/0] via 192.168.3.2
```

IPv4 routing table for router with IP address 192.168.3.2:

Codes: L - local, C - connected, S - static, R - RIP, M - mobile
D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter
<output omitted for brevity>

Gateway of last resort is not set

```
S      192.168.1.0/24 [1/0] via 192.168.3.1
S      192.168.2.0/24 [1/0] via 192.168.3.1
      192.168.3.0/24 is variably subnetted, 2 subnets, 2 masks
C      192.168.3.0/24 is directly connected, GigabitEthernet0/
L      192.168.3.2/32 is directly connected, GigabitEthernet0/
      192.168.4.0/24 is variably subnetted, 2 subnets, 2 masks
C      192.168.4.0/24 is directly connected, GigabitEthernet0/
L      192.168.4.1/32 is directly connected, GigabitEthernet0/
```

Disconnected from 192.168.1.1

Disconnected from 192.168.2.2

Disconnected from 192.168.3.2

```
All devices processed.
```

Summary

In this chapter you have seen how to efficiently manage multiple network devices using Netmiko, from basic concepts to more advanced automation techniques. First, you saw how to store a single IP address in a variable and use it within **ConnectHandler()**, which sets the stage for scaling the program to handle multiple devices. You also learned about the importance of Python's **for** loops. You saw that a **for** loop iterates through a list of IP addresses, establishes SSH connections, executes commands, and disconnects before moving to the next device. This sequential approach provides a structured and beginner-friendly method for automating device interactions.

You then saw how to extend this approach to configuring multiple devices with IPv6. Instead of applying configurations manually, the program matches each device's IP address with a predefined set of IPv6 configuration commands and uses **send_config_set()** to automate the process. This practical example highlights how automation simplifies large-scale configuration tasks, such as transitioning to IPv6, which would otherwise be time-consuming and error prone.

To further improve scalability, you saw how to use dictionaries for storing device connection parameters. Rather than maintaining a simple list of IP addresses, you saw how to store each device's IP, username, password, and other settings in a structured dictionary. Using dictionary unpacking, a program can dynamically retrieve connection details, making it easier to manage multiple devices without

hardcoding credentials into **ConnectHandler()**. You also saw an alternative approach, where device dictionaries are defined directly inside a list to keep the program more compact.

Storing credentials in a program is a security risk, so you learned about Python's **getpass()** function, which allows the user to securely enter SSH credentials at runtime without displaying them on the screen. This method prevents passwords from being exposed, making a program more secure while maintaining flexibility. You saw an example that assumes all devices share the same credentials, which is a common scenario in many network environments.

To make a program more robust, you learned about exception handling with **try-except** blocks. Errors such as connection timeouts, authentication failures, and read timeouts can be caught and handled appropriately, allowing the program to continue processing other devices instead of crashing.

In this chapter you saw an automation approach that involves maintaining multiple simultaneous SSH connections instead of handling one device at a time. Instead of repeatedly opening and closing SSH sessions, the program first establishes connections to all devices, then executes commands on all of them, and finally disconnects. By storing active connections in a dictionary, the program improves efficiency and reduces the overhead of reconnecting for each task. While this approach can be beneficial, it requires more system resources and may not be ideal for very large-scale automation. In addition, for large scenarios that require parallel execution, tools such as Threads or AsyncIO (or libraries like Nornir) would be more appropriate.

This chapter completes a comprehensive introduction to configuring multiple network devices with Netmiko, gradually transitioning from simple concepts to more structured and scalable automation techniques. Beginners are encouraged to focus on the fundamentals of lists, **for** loops, dictionaries, and error management, while readers with more Python experience may recognize opportunities for further optimization, such as parallel execution and dynamic command selection.

Those who are also familiar with both Python and IOS can go even further by leveraging features like Cisco's Embedded Event Manager (EEM) to automate network responses, schedule configurations, or implement safeguards—such as configuring ACLs with a time delay to prevent accidental disconnection from SSH sessions. They can also use Netmiko's textFSM parsing for easier command output processing.

Netmiko is an incredibly powerful yet accessible tool that bridges the gap between traditional network engineering and modern automation. For entry-level network engineers, it provides an intuitive way to start automating repetitive CLI tasks without requiring deep programming expertise. At the same time, experienced engineers can leverage Netmiko to streamline large-scale deployments, enforce configuration consistency, and integrate with more advanced automation frameworks.

Whether you're a new Python programmer looking to apply coding to networking or an experienced developer seeking to enhance automation workflows, Netmiko provides an efficient and reliable way to interact with network devices programmatically. By embracing automation and programmability, engineers can reduce errors, save time, and gain greater control over network infrastructure,

making Netmiko a valuable tool for anyone looking to advance their networking and automation skills.

Part 2: NAPALM

Chapter 5. Introducing NAPALM and Structured Data

The most common NAPALM feature, when getting started with it, is its capability to retrieve information from network devices in a uniform fashion. Any data returned from NAPALM is normalized for all devices in a consistent way.

From the book *Network Programmability & Automation*, 2nd edition, O'Reilly

NAPALM is a Python library that lets you interact with different network device operating systems using a structured data approach. The NAPALM project was initially developed by David Barroso, with significant contributions from Kirk Byers, Mircea Ulinic, Brandon Ewing, and Elisa Jasinska.

This chapter introduces the basic concept of a Python dictionary, the format of a dictionary, and how NAPALM returns information in this structured format. In subsequent chapters, we will explore different types of Python dictionaries used with NAPALM, as well as how to iterate through them using **for** loops.

It is important to remember that, just as with Netmiko, you can easily nest NAPALM programs inside a **for** loop to connect to and manage multiple devices. By iterating through a list or dictionary of device information, you can automate tasks across an entire network, not just a single device. We will explore more structured approaches to managing multiple devices in [Part 3](#), “[Nornir](#).”

What Is NAPALM, and How Is It Different from Netmiko?

According to NAPALM's website (napalm.readthedocs.io), "NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is a Python library that implements a set of functions to interact with different network device Operating Systems using a unified API." The phrase "using a unified API" means that NAPALM provides a single, consistent interface for interacting with network devices, regardless of vendor (for example, Cisco, Juniper, Arista). Whether you are retrieving interface IP addresses from a Cisco device or another vendor's network device, NAPALM returns the information in the same structured data format.

In previous chapters, you saw that Netmiko allows you to use CLI commands and receive output as if you were directly typing the commands at the CLI. The Cisco IOS output is designed to be human readable, meaning it presents information in a way that is easy to understand just by looking at it.

We will continue using the same topology we used in [Chapter 4](#), “[Accessing Multiple Devices with Netmiko](#),” now focusing specifically on Router-R1. [Figure 5-1](#) shows the partial topology.

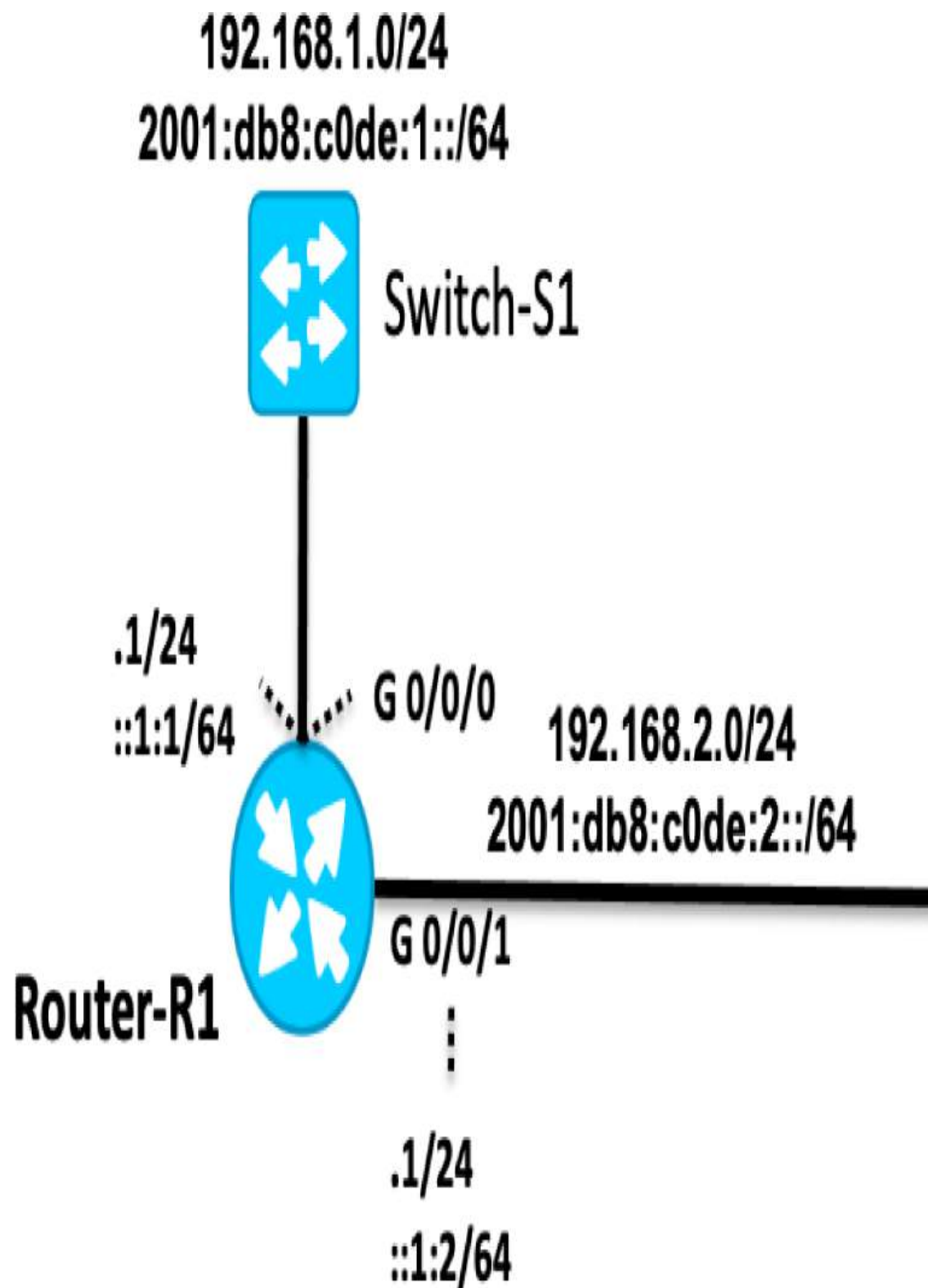


Figure 5-1 *Partial topology with Router-R1*

[Example 5-1](#) shows the output from the Netmiko method **send_command()**, using the **show ip interface brief** command. The output is exactly the same as if you had typed it directly into the router's CLI. The CLI output is designed to be easy for humans to read.

Example 5-1 *Netmiko **send_command('show ip interface brief')** Method Output*

```
MyPrompt% python3 ex4-1_send_command_device.py
Interface                IP-Address    OK? Method Status
GigabitEthernet0/0/0    192.168.1.1   YES NVRAM  up
GigabitEthernet0/0/1    192.168.2.1   YES NVRAM  up
GigabitEthernet0/0/2    unassigned    YES NVRAM  administrative
GigabitEthernet0        unassigned    YES NVRAM  administrative
MyPrompt%
```

[Example 5-2](#) shows how you would use NAPALM to retrieve similar information by using the NAPALM **get_interfaces_ip()** method. You will see how to use this and other methods shortly. For now, just notice how it compares to the Netmiko method output. The data contains the same key details, such as the IPv4 address for each interface, but it is presented in a structured format that is more computer friendly. Although it is less human readable, this structured data approach allows for easier automation, parsing, and integration with other tools. In this part of the book, you will see how to use Python to make this type of output more human readable and to present the data in the way you want it to be viewed.

If you are new to Python dictionaries, don't worry about understanding their format. By reading the chapters in this part of the book, you will gain a basic understanding of how the information is organized and managed.

Example 5-2 NAPALM *get_interfaces_ip()* Method Output

```
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length': 24},  
                               'ipv6': {'2001:db8:c0de:1::1': {'prefix_length': 64},  
                                         'fe80::1:1': {'prefix_length': 64}},  
 'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length': 24},  
                               'ipv6': {'2001:db8:c0de:2::1': {'prefix_length': 64},  
                                         'fe80::2:1': {'prefix_length': 64}}
```

One major advantage of structured data is that it provides a consistent format across different network vendors. In a multivendor environment, where devices from Cisco, Juniper, Arista, and others might have different CLI outputs, NAPALM abstracts the differences and presents the data in a unified way. This makes automation and network management more efficient and scalable.

As of this writing, NAPALM supported the following network operating systems:

- Cisco IOS
- Cisco IOS XR
- Cisco NX-OS
- Arista EOS
- Juniper JunOS

NAPALM is an excellent tool for becoming familiar with structured data, as it provides a consistent, easy-to-use interface for retrieving and modifying network information in a structured format. Unlike raw CLI output, which is unstructured and requires parsing, NAPALM returns data in key/value pairs within Python dictionaries, making it more readable and programmatically accessible. This approach aligns closely with how modern network automation tools

like RESTCONF and NETCONF operate. Both RESTCONF and NETCONF use structured data formats such as JSON and XML to interact with devices. By working with NAPALM, you can develop a foundational understanding of structured data representation, which makes transitioning to RESTCONF, NETCONF, and API-driven automation much easier.

Installing the NAPALM Library

To download and install NAPALM, from your Windows Command Prompt or PowerShell or from a macOS or Linux terminal, type the following command:

```
MyPrompt% pip3 install napalm
```

If you encounter errors during installation, ensure that Python and pip are updated and run the command as an administrator (for example, use **sudo pip3 install napalm** on macOS/Linux or run your Terminal as an administrator on Windows).

Basic NAPALM Framework

Before we begin looking at how dictionary data is organized, let's begin with the basic framework of a NAPALM program. As you can see in [Example 5-3](#), the framework is similar to that of a Netmiko program; therefore, we can skip some of the details.

[Example 5-3](#) shows the basic framework for every NAPALM program, which includes these steps:

1. Import the required library and module.
2. Select a network driver.

3. Create a device object.
4. Establish a connection.
5. Implement NAPALM methods.
6. Close the connection.

Example 5-3 *ex5-3_napalm_framework.py*

```
# 1. Import required library and module
import napalm
from pprint import pprint

# 2. Select network driver
driver = napalm.get_network_driver('ios')

# 3. Create a device object
device = driver( hostname= '192.168.1.1',
                  username= 'admin',
                  password= 'cisco',
                  optional_args= {'secret':'spot'}
                  )

# 4. Establish a connection
device.open()

# 5. Implement NAPALM methods
pprint(device.get_facts())

# 6. Close the connection
device.close()
```

The following is a brief description of the framework in [Example 5-3](#).

Step 1.Import the required library and module. The first step is to import the NAPALM library, which provides a unified interface for interacting with different network devices. In addition, **pprint** (which stands for *pretty-print* and which we'll discuss in a moment) is imported to format output data in a more readable way.

Step 2.Select a network driver. NAPALM supports multiple device vendors, such as Cisco IOS, Juniper Junos, and Arista EOS. Here, we select **ios** as the driver, meaning NAPALM will use its Cisco IOS-specific methods.

Step 3.Create a device object. A device object is created using the selected driver. This object holds connection details such as hostname, SSH username and password, and optional arguments like **secret** (the IOS enable password for privileged mode).

Step 4.Establish a connection. The **open()** method initiates a connection to the network device by using the provided credentials. This allows NAPALM to interact with the device and retrieve or modify its configuration.

Step 5.Implement NAPALM methods. Once you have a connected, you can use NAPALM methods like **get_facts()** to retrieve structured data about essential device information to get a high-level overview of the network device. This method returns a dictionary that contains key details such as the hostname, model, OS version, serial number, vendor, uptime, and a list of network interfaces. We will talk about the **pprint** function and **get_facts()** along with other NAPALM methods later in this chapter. We are starting with

the **get_facts()** method because it is an easier data structure to work with.

Step 6. Close the connection. After completing all tasks, it is good practice to close the connection by using **close()**. This helps free up resources and maintain network security.

Just like Netmiko, NAPALM uses object-oriented programming (OOP) principles, but you don't need to be an expert in OOP to use it effectively. The NAPALM **get_network_driver()** function returns a *class* that is specific to the vendor (for example, an IOS driver for Cisco IOS). When you create a device *object* by using this class, you can interact with the network device through *methods* that are built into NAPALM.

For example, by calling **device.get_facts()**, you retrieve structured IP device information, and by calling **device.open()** and **device.close()**, you manage the connection. These methods are accessed using *dot notation*, just like in Netmiko, making it easy to execute commands and retrieve data in a structured way. Again, understanding OOP is not required; however, recognizing that NAPALM methods belong to a device object can help clarify how the library works.

To display the output in this case, you use the **pprint()** function instead of the standard Python **print()** function. The **pprint** module is imported into Python and used to format and display complex data structures in a more readable way. When working with NAPALM, device data is often returned as nested dictionaries, which can be difficult to read using a standard **print()** statement. Using **pprint()** makes it easier to analyze and debug structured data.

[Example 5-4](#) shows the output you get when you use the **print()** function along with the NAPALM **get_facts()**

method. The data is presented as a single continuous string, which is difficult to read.

Example 5-4 **print()** Command Output

```
print(device.get_facts())

{'uptime': 2820.0, 'vendor': 'Cisco',
 'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), \
RELEASE SOFTWARE (fc8)', 'serial_number': 'FLM2229W1R6', 'model':
'hostname': 'Router-R1', 'fqdn': 'Router-R1.SSH-KEY.com', 'inte
['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet
'GigabitEthernet0']
}
```

In contrast, [Example 5-5](#) uses the **pprint()** function with the same NAPALM method. As you can see, it formats the data in a more structured and readable way, displaying each key/value pair on separate lines. By default, **pprint()** sorts dictionary keys alphabetically (though you can control this setting with the **sort_dicts** parameter). This why the output in [Example 5-5](#) begins with **'fqdn'** and ends with **'vendor'**, whereas the keys are unordered in [Example 5-4](#). Python dictionaries are accessed by key rather than order, so this difference is purely visual when you're retrieving and using structured data. Both representations contain the same information and behave identically when accessed programmatically.

Note

In rare cases where structured data is used to generate device configuration commands, the order in which those commands are applied can matter due to CLI rules (for example, when configuring

VRFs or interface IP addresses). This is not a limitation of dictionaries themselves but of the device's CLI command rules. NAPALM's configuration methods typically account for these ordering requirements internally.

Example 5-5 *pprint()* Command Output

```
pprint(device.get_facts())

{'fqdn': 'Router-R1.SSH-KEY.com',
 'hostname': 'Router-R1',
 'interface_list': ['GigabitEthernet0/0/0',
                    'GigabitEthernet0/0/1',
                    'GigabitEthernet0/0/2',
                    'GigabitEthernet0'],
 'model': 'ISR4331/K9',
 'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),
               '16.6.3, RELEASE SOFTWARE (fc8)',
 'serial_number': 'FLM2229W1R6',
 'uptime': 2820.0,
 'vendor': 'Cisco'}
```

How NAPALM Retrieves Data Without an API on Cisco IOS

So, how does NAPALM get IOS information if Cisco IOS doesn't offer a modern API like RESTCONF or NETCONF? The answer lies in *API abstraction*.

Remember that an *API (application programming interface)* is a set of rules or tools that allow programs to communicate with other software, devices, or systems. An

API defines what information you can request, how to request it, and what kind of response you'll receive.

With *API abstraction*, a tool (like NAPALM) provides a consistent interface (API) to the user and handles the platform-specific commands and complexities behind the scenes. In other words, *NAPALM does the hard work for you!* You interact with NAPALM through a standardized Python interface, and NAPALM takes care of translating your request into device-specific commands and parsing the results.

Regardless of the network operating system (Cisco IOS, Juniper Junos, or others), NAPALM offers a number of benefits:

- It lets you use the Python code structure (framework) and methods, such as **get_facts()**.
- It retrieves consistent data, no matter the network OS.
- It uses vendor CLI commands internally and parses the output.
- It returns results in structured formats (typically Python dictionaries).

For example, when you use the **get_facts()** method to retrieve basic device details, NAPALM might send a **show version** command to a Cisco IOS device. It might also parse the CLI output to extract information such as the hostname, model, and serial number. You don't need to worry about device-specific syntax or parsing; NAPALM abstracts all of that, making your automation tasks much easier and vendor independent.

Good News, Bad News: Understanding Structured Data

The good news is that with NAPALM, the data is structured. The bad news is that with NAPALM the data is structured. In other words, "Great! I have data from multiple devices in the same structured format—but what do I actually do with it?" Don't worry. In the chapter and subsequent chapters, we'll look at how to retrieve and manage this information.

When working with NAPALM, instead of receiving raw text, as you would from a traditional CLI command, you get data returned in a structured format. This means the data is organized in a way that makes it easier for programming languages like Python to process, filter, and automate tasks across different network devices.

If you have ever used a spreadsheet or database or even kept a list of contacts in an address book, you have already worked with structured data! A spreadsheet, for example, organizes data in rows and columns, allowing you to look up and manage information efficiently. Similarly, an address book stores names, phone numbers, and addresses in a structured way so that you can quickly find what you need. Structured data in Python works similarly: It is organized so that it can be accessed and used efficiently.

A *data structure* is a way of organizing, storing, and managing data efficiently to enable specific operations, such as searching, retrieving, and modifying data. In Python, one of the most common data structures is a *dictionary*.

The Basics of a Python Dictionary

If you're already comfortable with Python dictionaries, feel free to skim this section. Otherwise, read on to explore how Python uses dictionaries to organize information.

Unlike a list, where items are accessed by their position, a dictionary allows direct access using meaningful keys. A

dictionary in Python provides a way to store and organize data using key/value pairs, which make it easy to look up, update, and manage information efficiently. Each key in a dictionary is unique, serving as a label that allows quick access to its associated value. This makes dictionaries especially useful when dealing with structured data where each item has a unique identifier, such as interface names, IP addresses, or device configurations.

A good way to think about a dictionary is to compare it to a contact list on a smartphone. Each contact name (the key) is unique and corresponds to a phone number (the value). For example, you might save “Mom” as a contact and store her phone number with the contact. Instead of searching through a long list, you can retrieve her number instantly by looking up “Mom.” This is the same process used to quickly access values in a dictionary by using their keys.

It is easy to visualize dictionary key/value pairs in a table format similar to what you might find in a spreadsheet.

[Table 5-1](#) shows the key/value pairs for a contact list.

Table 5-1 *Key/Value Pairs for a Contact List*

Key	Value
Mom	555-1234
Rick	555-4444
Adrian	555-9999

[Table 5-2](#) shows another table of key/value pairs. These pairs are from the output of the NAPALM **get_facts()** method from [Example 5-6](#).

Table 5-2 *Partial Table Showing Key/Value Pairs from `get_facts()`*

Key	Value
model	ISR4331/K9
os_version	ISR Software Version 16.3.3
vendor	Cisco

Example 5-6 shows what these key/value pairs would look like using a Python dictionary. Each dictionary has a variable that represents the content of that dictionary. The first dictionary uses the variable **contacts**, and the second one uses the variable **device_variable**. You'll learn about the relationship between a variable and a dictionary in the next section.

Example 5-6 *Examples of Key/Value Pairs in a Python Dictionary*

```
# Contact list as a dictionary
contacts = {
    'Mom': '555-1234',
    'Rick': '555-4444',
    'Adrian': '555-9999'
}

# Device information as a dictionary
device_variable = {
    'model': 'ISR4331/K9',
    'os_version': 'ISR Software Version 16.3.3',
```

```
'vendor': 'Cisco'
}
```

How Python Formats a Dictionary

In Python, a dictionary is typically assigned to a variable like this:

```
variable = { dictionary }
```

To explain the format of a dictionary, we will use the variable **device_variable** to store a dictionary:

```
device_variable = { dictionary }
```

To keep things simple, the dictionary will contain just the three key/value pairs retrieved using the **get_facts()** method in [Example 5-6](#). [Example 5-7](#) shows the variable **device_variable** storing the sample dictionary.

Example 5-7 Using a Python Dictionary

```
# Device information as a dictionary
device_variable = {
    'model': 'ISR4331/K9',
    'os_version': 'ISR Software Version 16.3.3',
    'vendor': 'Cisco'
}
```

As you can see in this example, the variable **device_variable** stores the dictionary in curly braces (**{}**):

```
device_variable = { key-value pairs }
```

The curly braces define a Python structure as a dictionary.

You can also see that each entry consists of a key (on the left) and its corresponding value (on the right), separated by a colon (:):

```
'model': 'ISR4331/K9'
```

Multiple key/value pairs are separated by commas (,):

```
'model': 'ISR4331/K9',  
'os_version': 'ISR Software Version 16.3.3',  
'vendor': 'Cisco'
```

This structure allows multiple pieces of information to be stored and is easy to read. The last key/value pair does not use a comma. The keys in a dictionary must be unique and are typically strings, while the values can be of any data type—strings, numbers, lists, or even other dictionaries. You will become more familiar with this syntax as you progress through this book.

Python ignores whitespace within dictionaries, which means you can format them with indentation and spacing for improved readability without affecting functionality.

- These are some other important aspects of a dictionary to know:
- **Keys must be unique.** Each key (for example, **'model'**, **'os_version'**, **'vendor'**) must be *unique* within the dictionary. If a duplicate key exists, Python *overwrites* the previous value with the new one.
- **Keys are typically strings.** In [Example 5-7](#), all the dictionary keys are strings (**'model'**, **'os_version'**, **'vendor'**). Keys can also be numbers, but they *must be immutable* (that is, unchangeable).

- **Values can be any data type.** In [Example 5-7](#), all values are strings ('ISR4331/K9', 'ISR Software Version 16.3.3', 'Cisco'). However, dictionary values can also be: numbers (for example, an uptime counter), lists (for example, a list of interfaces), or other dictionaries (for example, a nested dictionary with detailed device specifications).

get_facts(): Our First NAPALM Method and Dictionary

To better understand Python dictionaries, especially how they are used with NAPALM, let's take a look at a program that uses the NAPALM method **get_facts()**. In this example, we will assign the output from **get_facts()** directly to the variable **device_facts**.

Continuing to use the program from [Example 5-3](#), [Example 5-8](#) uses the **get_facts()** method from NAPALM to retrieve structured information about a network device. This method returns a dictionary that stores each piece of information (such as hostname, model, or vendor) in a key/value pair.

Example 5-8 *ex5-8_get_facts.py*

```
# 1. Import required library and module
import napalm
from pprint import pprint

# 2. Select network driver
driver = napalm.get_network_driver('ios')

# 3. Create a device object
device = driver(
    hostname='192.168.1.1',
```



```
        username='admin',
        password='cisco',
        optional_args={'secret':'spot'}
    )

# 4. Establish a connection
device.open()

# 5. Implement NAPALM methods

print("\nPrint results from get_facts() using pprint(device.get_facts())")
print("-" * 63)
pprint(device.get_facts())

# Assign the output of the get_facts() to the variable device_facts
device_facts = device.get_facts()

print("\nPrint key-value pairs individually using device_facts")
print("-" * 61)
print("fqdn:", device_facts['fqdn'])
print("hostname:", device_facts['hostname'])
print("interface_list:", device_facts['interface_list'])
print("model:", device_facts['model'])
print("os_version:", device_facts['os_version'])
print("serial_number:", device_facts['serial_number'])
print("uptime:", device_facts['uptime'])
print("vendor:", device_facts['vendor'])

# 6. Close the connection
device.close()
```

[Example 5-9](#) shows the output of the program **ex5-8_get_facts.py**.

Example 5-9 *Output from [Example 5-8](#)*

```
MyPrompt% python3 ex5-8_get_facts.py
```

```
Print results from get_facts() using pprint(device.get_facts()):
```

```
-----
{'fqdn': 'Router-R1.SSH-KEY.com',
 'hostname': 'Router-R1',
 'interface_list': ['GigabitEthernet0/0/0',
                    'GigabitEthernet0/0/1',
                    'GigabitEthernet0/0/2',
                    'GigabitEthernet0'],
 'model': 'ISR4331/K9',
 'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),
               '16.6.3, RELEASE SOFTWARE (fc8)',
 'serial_number': 'FD02302A08A',
 'uptime': 3900.0,
 'vendor': 'Cisco'}
```

```
Print key-value pairs individually using device_facts['key']:
```

```
-----
fqdn: Router-R1.SSH-KEY.com
hostname: Router-R1
interface_list: ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1',
                'GigabitEthernet0/0/2', 'GigabitEthernet0']
model: ISR4331/K9
os_version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.6.3, RELEASE SOFTWARE (fc8)
serial_number: FD02302A08A
uptime: 3900.0
```

```
vendor: Cisco
MyPrompt%
```

The output in [Example 5-9](#) shows the structured dictionary returned by the **get_facts()** method when executed on a Cisco ISR4331 router. First, you use **pprint** to print the entire dictionary, with all the key/value pairs in a structured format.

After printing the full dictionary, you use the variable **device_facts**, which holds the dictionary, to reference specific key/value pairs individually. This allows you to extract and display information in a more readable way, such as printing only the hostname (**device_facts['hostname']**), OS version, or vendor, without needing to reprint the entire dictionary. This approach demonstrates how NAPALM's structured output makes it easy to retrieve and work with specific information.

Assigning the Dictionary to a Variable

By assigning the dictionary to **device_facts**, you avoid repeatedly calling **device.get_facts()**. This is especially useful when working with network automation, where retrieving data from a device might take time. Instead, you store the results once and use them multiple times. For example, you assign the dictionary to a variable called **device_facts** like this:

```
device_facts = device.get_facts()
```

device_facts = device.get_facts() assigns the output of the **get_facts()** method to the variable **device_facts**. Because **device_facts** stores this output, it is a variable of type *dictionary*.

Once you have the dictionary stored in **device_facts**, you can access specific values by using their corresponding keys. This is the syntax for accessing specific values:

```
print(variable_name['key'])
```

So in this case, you would specify **print(device_facts)** followed by the key enclosed in single or double quotes and then square brackets (**[]**). This is a lot quicker than searching through a long list. For example, here is how you retrieve the hostname of the device:

```
print(device_facts['hostname'])
```

This tells Python, “Look in the dictionary stored in **device_facts** and return the value associated with the key '**hostname**'.” Because no key name is included in the **print()** statement, the output is simply the value itself, **Router-R1**..

```
Router-R1
```

Each key in the dictionary—such as '**model**', '**hostname**', or '**vendor**'—refers to a specific piece of device information. By placing the keys inside square brackets and passing them to the **print()** function, you can display just the values you’re interested in, without showing the entire dictionary. This makes it easy to organize and format output exactly the way you want. [Example 5-10](#) shows a sampling of how the command is used in [Example 5-8](#).

Example 5-10 *Using Keys to Extract Values from a Python Dictionary*

```
device_facts = device.get_facts()
print("model:", device_facts['model'])           # Output: model: I!
```

```
print("hostname:", device_facts['hostname']) # Output: hostname
print("vendor:", device_facts['vendor'])      # Output: vendor: (
```

[Example 5-11](#) shows what the code would look like if you didn't store the dictionary in a variable and instead called **device.get_facts()** each time.

Example 5-11 *Calling the **get_facts()** Method Each Time*

```
print("model:", device.get_facts()['model'])      # Output: model
print("hostname:", device.get_facts()['hostname']) # Output: hos
print("vendor:", device.get_facts()['vendor'])    # Output: ven
```

In [Example 5-11](#), **device.get_facts()** is executed multiple times, meaning the network device is queried for data each time you need a value. This can lead to unnecessary network traffic and increased response times. Storing the dictionary in **device_facts** instead allows you to retrieve all values from memory rather than repeatedly make calls to the device.

Note

The preferred approach is typically not to call the method each time but to assign the dictionary to a variable.

Because each key in a dictionary is unique, you can always be sure that by accessing a key, you will get the correct value. This makes using dictionaries a very efficient way to store and retrieve structured data in network automation, where you often need to pull specific details about a device quickly. By using dictionaries this way, you can quickly and reliably access important details about a network device without having to parse through raw text or unstructured

data. This is one of the biggest advantages of using structured data in Python.

Understanding Different Types of Values in a Dictionary

In the output from **get_facts()**, you can see that each key has an associated value, but these values are not always simple text known as *strings*. While some keys store values as strings (for example, **'hostname': 'Router-R1'**), other keys store values that contain lists, numbers, and, as you will see in later methods, even nested dictionaries. In Python, a list is an ordered collection of items that can store multiple values in a single variable. A list is defined using square brackets (**[]**), and list elements are separated by commas.

For example, the key **'interface_list'** contains a list of network interfaces:

```
'interface_list': ['GigabitEthernet0/0/0',  
                  'GigabitEthernet0/0/1',  
                  'GigabitEthernet0/0/2',  
                  'GigabitEthernet0']
```

This list allows you to store multiple values under a single key.

Another example is the key **'uptime'**, which holds a number of type float (that is, a floating-point number) that represents the time the device has been running (in seconds):

```
'uptime': 8040.0
```

Because this value is a number (a float), it can be used in calculations, such as converting uptime to hours or

comparing it with other devices.

As yet another example, the key **'os_version'** is a long string that contains detailed information about the software version:

```
'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),  
RELEASE SOFTWARE (fc8)'
```

Note

Even though this is a long string, you can extract specific parts of it, if needed, by using built-in Python string methods like **strip()** and **split()**. Don't worry about this if you're new to Python.

You can use a simple **for** loop to display each value in the key **'interface_list'** on a separate line, as shown in [Example 5-12](#). The code in this example focuses just on the section where you would add the **for** loop, beginning with assigning the resulting dictionary from **device.get_facts()** to the variable **device_facts**. The syntax **device_facts['interface_list']** accesses the value associated with the key **'interface_list'** in the **device_facts** dictionary.

Because **['interface_list']** is a list of interface names (for example, **'GigabitEthernet0/0/0'**), the **for** loop iterates through the elements in the list, allowing individual interface names to be printed separately. During each iteration, the variable **interface** holds the value of the current element from the list, allowing it to be printed separately. This approach ensures that all interfaces stored in the dictionary are displayed in the same format.

Example 5-12 *ex5-12_for_loop.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)

device.open()

# Assign the output of the get_facts() to the variable device_facts
device_facts = device.get_facts()

# Printing each interface separately
print("interface_list:")
for interface in device_facts['interface_list']:
    print(" - ", interface)    # Alternative f-string: print(f'

device.close()
```

[Example 5-13](#) shows the output from running the code in [Example 5-12](#).

Example 5-13 *Output from [Example 5-12](#)*

```
interface_list:
- GigabitEthernet0/0/0
- GigabitEthernet0/0/1
```


- GigabitEthernet0/0/2
- GigabitEthernet0

Creating a NAPALM Dictionary Without a Device

When working with NAPALM, retrieving live data from a network device is often the goal. However, there are some situations in which accessing a real device may not be practical. For example, you might be developing or testing Python code without a network device available, you might be a student who doesn't have access to lab equipment, or you might simply be experimenting with structured data. In such cases, using a *predefined dictionary* that mimics the structure of NAPALM's output is an effective alternative.

The two programs in [Example 5-14](#) and [Example 5-16](#) demonstrate this concept. In [Example 5-14](#), the dictionary **device_facts** is populated dynamically by retrieving live data using **device.get_facts()**. You have already seen this method of retrieving device information.

Example 5-14 *ex5-14_get_facts.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
```

```
)  
  
device.open()  
  
device_facts = device.get_facts()  
  
pprint(device_facts)  
  
device.close()
```

The program in [Example 5-14](#) retrieves live device data using **device.get_facts()** and returns structured information in dictionary format. [Example 5-15](#) shows output from a real device.

Example 5-15 *Output from [Example 5-14](#)*

```
{'fqdn': 'Router-R1.SSH-KEY.com',  
 'hostname': 'Router-R1',  
 'interface_list': ['GigabitEthernet0/0/0',  
                    'GigabitEthernet0/0/1',  
                    'GigabitEthernet0/0/2',  
                    'GigabitEthernet0'],  
 'model': 'ISR4331/K9',  
 'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),  
               '16.6.3, RELEASE SOFTWARE (fc8)',  
 'serial_number': 'FD02302A08A',  
 'uptime': 3900.0,  
 'vendor': 'Cisco'}
```

In case you are learning network automation but don't have access to Cisco devices, [Example 5-16](#) provides an alternative approach: It manually defines **device_facts** as a

dictionary that mirrors the structure and key/value pairs returned by the **get_facts()** method in [Example 5-14](#).

To effectively use a predefined dictionary in place of live device data, you need access to the same dictionary structure that the NAPALM method returns. Having this access ensure that your non-device program mimics the actual output format for consistency and compatibility. [Appendix A, “Python Virtual Environments,”](#) provides several examples.

Example 5-16 *ex5-16_device_facts.py*

```
from pprint import pprint

device_facts = {
    'fqdn': 'Router-R1.SSH-KEY.com',
    'hostname': 'Router-R1',
    'interface_list': [ 'GigabitEthernet0/0/0',
                        'GigabitEthernet0/0/1',
                        'GigabitEthernet0/0/2',
                        'GigabitEthernet0'],
    'model': 'ISR4331/K9',
    'os_version': 'ISR Software (X86_64_LINUX_IOSD-1
                  'Version 16.6.3, RELEASE SOFTWARE
    'serial_number': 'FLM2229W1R6',
    'uptime': 2820.0,
    'vendor': 'Cisco'
}

pprint(device_facts)
```

Because the two approaches result in dictionaries with identical format, any Python code written to process

device_facts will work the same way in either scenario. For example, the **pprint(device_facts)** statement in both programs will result in exactly the same output. You could also use other statements we have previously discussed, such as **print(device_facts['hostname'])** or a **for** loop, to display interfaces.

Predefined dictionaries provide flexibility, allowing developers, students, and engineers to work with structured network data without needing direct access to a device.

NAPALM Methods

This section provides a list of the NAPALM methods available for Cisco IOS at the time of this writing. While NAPALM provides a unified API, the exact methods supported can vary depending on the network operating system. Different vendors and vendor operating systems may have additional or slightly different capabilities. We will explore some of these methods in the following chapters to help get you started with NAPALM and structured data. For a complete and up-to-date list of supported methods across various network operating systems, refer to the official NAPALM documentation at <https://napalm.readthedocs.io/en/latest/support/index.html>.

General Device Information

These methods retrieve general information about devices:

- **get_config():** Retrieves the current device configuration, including startup, running, and candidate configurations.
- **get_facts():** Retrieves general device information, such as hostname, model, OS version, serial number, vendor, uptime, and interfaces.

- **get_environment():** Retrieves environmental metrics such as CPU usage, memory utilization, and temperature.
- **get_snmp_information():** Retrieves SNMP configuration details, including SNMP communities, location, and contact information.
- **get_users():** Returns a list of configured local users and their privilege levels.
- **is_alive():** Checks whether the device is reachable and responding.

Interface and Networking Information

These methods get information on interfaces and networking:

- **get_interfaces():** Returns the status and settings of all interfaces, including speed, MAC address, and administrative status.
- **get_interfaces_counters():** Provides interface-level statistics, including packet counts, errors, and discards.
- **get_interfaces_ip():** Retrieves assigned IP addresses for each interface.
- **getlldp_neighbors():** Provides LLDP neighbor information, listing directly connected devices.
- **getlldp_neighbors_detail():** Returns detailed LLDP neighbor information, including remote port descriptions and capabilities.

Routing and Network Instances

These methods retrieve information on routing and network instances:

- **get_bgp_config():** Retrieves the configured BGP settings for the device.
- **get_bgp_neighbors():** Provides BGP neighbor information, including session state and AS numbers.
- **get_bgp_neighbors_detail():** Returns detailed information about BGP neighbors, including policies and attributes.
- **get_network_instances():** Retrieves information about virtual routing and forwarding instances on the device.
- **get_route_to(destination):** Returns routing table information for a specific destination.

Layer 2 and Neighbor Discovery

These methods are used for Layer 2 and neighbor discovery:

- **get_arp_table():** Returns the ARP table entries, including MAC addresses, IP addresses, and interface mappings.
- **get_ipv6_neighbors_table():** Provides IPv6 neighbor discovery table entries, similar to ARP for IPv4.
- **get_mac_address_table():** Retrieves the MAC address table, listing learned MAC addresses and their associated interfaces.
- **get_vlans():** Returns VLAN information, including VLAN ID and associated interfaces.

Optical and NTP Information

These methods get information related to NTP and optical transceivers:

- **get_ntp_peers():** Returns a list of configured NTP peers.
- **get_ntp_servers():** Retrieves configured NTP servers.
- **get_ntp_stats():** Provides NTP synchronization details and peer status.
- **get_optics():** Retrieves optical transceiver details, including power levels and operational status.

Connectivity and Testing

These methods are used for connectivity and testing:

- **ping(destination):** Sends ICMP pings to a destination and returns response statistics.
- **traceroute(destination):** Performs a traceroute to a specified IP address, showing the path that packets take through the network.

Using the CLI

When there is not a NAPALM method to retrieve the data required, a network CLI command can be used to obtain the information in a traditional format:

- **cli():** Sends raw CLI commands to a network device and retrieve the output as a string.

Note

The **cli()** method, which allows direct execution of CLI commands through NAPALM, is covered in

[Chapter 8, “Configuring Devices with NAPALM.”](#)

Before we get there, we will further explore how to work with NAPALM dictionaries.

Configuration-Related Methods

The following configuration-related methods allow you to apply changes to a device, such as pushing configurations or rolling back changes:

- **load_merge_candidate(filename=None, config=None):** Loads a partial configuration to merge with the current running config.
- **load_replace_candidate(filename=None, config=None):** Loads a full configuration to replace the current running config.
- **compare_config():** Compares the candidate configuration with the running configuration and returns the differences.
- **commit_config():** Applies the candidate configuration changes to the device.
- **discard_config():** Discards the candidate configuration without applying it.
- **rollback():** Rolls back to the previously saved configuration.

Note

These configuration-related methods are covered in detail in [Chapter 8](#), where you'll learn how to safely push, preview, commit, discard, and roll back configuration changes using NAPALM.

Using Python Interactive Mode to Experiment with NAPALM

Just as we did earlier with Netmiko, we can also use Python's interactive mode to explore and experiment with

NAPALM. This is especially useful when you're learning how NAPALM retrieves and presents device information and you want to get feedback immediately. It is also helpful when you want to try out specific methods, such as **get_facts()** or **get_interfaces()**, without writing an entire program.

[Example 5-17](#) shows a sample interactive session using Python with NAPALM on macOS.

Example 5-17 *Sample Interactive Session Using NAPALM*

```
MyPrompt% python3
Python 3.10.3 (v3.10.3:a342a49189) [Clang 13.0.0 (clang-1300.0.22.1)]
Type "help", "copyright", "credits" or "license()" for more
>>> import napalm
>>> from pprint import pprint
>>>
>>> driver = napalm.get_network_driver('ios')
>>>
>>> device = driver(
...     hostname='192.168.1.1',
...     username='admin',
...     password='cisco',
...     optional_args={'secret': 'spot'})
>>> device.open()
>>>
>>> pprint(device.get_facts())
{'fqdn': 'Router-R1.SSH-KEY.com',
 'hostname': 'Router-R1',
 'interface_list': ['GigabitEthernet0/0/0',
                    'GigabitEthernet0/0/1',
                    'GigabitEthernet0/0/2',
                    'GigabitEthernet0'],
 'model': 'ISR4331/K9',
```

```
'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),  
              '16.6.3, RELEASE SOFTWARE (fc8)',  
'serial_number': 'FD02302A08A',  
'uptime': 16140.0,  
'vendor': 'Cisco'}  
>>>  
>>> print(device.get_facts()['hostname'])  
Router-R1  
>>>  
>>> device_facts = device.get_facts()  
>>> print("hostname:", device_facts['hostname'])  
hostname: Router-R1  
>>>  
>>>  
>>> response = device.ping(destination="192.168.2.2", count=5)  
>>> pprint(response)  
{'success': {'packet_loss': 0,  
             'probes_sent': 5,  
             'results': [{'ip_address': '192.168.2.2', 'rtt': 0  
                          {'ip_address': '192.168.2.2', 'rtt': 0  
                          {'ip_address': '192.168.2.2', 'rtt': 0  
                          {'ip_address': '192.168.2.2', 'rtt': 0  
                          {'ip_address': '192.168.2.2', 'rtt': 0  
             'rtt_avg': 1.0,  
             'rtt_max': 1.0,  
             'rtt_min': 1.0,  
             'rtt_stddev': 0.0}}}  
>>>  
>>>  
>>> ^D  
MyPrompt%
```

Notice that [Example 5-17](#) uses NAPALM's **ping()** method, which is one of the connectivity and testing methods listed in the previous section. This method is used to send ICMP echo requests to a destination IP address and receive structured results. For example, the command **device.ping(destination="192.168.2.2", count=5)** sends five pings to the specified IP address and returns the results as a Python dictionary. The output includes useful statistics such as the number of packets sent, packet loss, and round-trip times (average, minimum, maximum, and standard deviation). Each individual ping result is also listed with its round-trip time and the IP address that responded. This method provides a consistent, structured way to test connectivity across different network operating systems without relying on parsing raw CLI output.

Summary

This chapter introduces NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor Support) as a powerful Python library for interacting with network devices in a structured and vendor-agnostic way. It contrasts NAPALM with Netmiko, highlighting that whereas Netmiko retrieves raw CLI output, NAPALM normalizes and structures data across multiple network vendors, making it easier to automate and analyze.

In this chapter we have discussed installing NAPALM and setting up a basic NAPALM program, following a consistent framework that includes selecting a network driver, creating a device object, opening a connection, running NAPALM methods, and closing the session.

This chapter includes a detailed explanation of Python dictionaries, ensuring that beginners can understand how NAPALM organizes data into key/value pairs. It also explores

different types of dictionary values, including lists (for example, interface names), numbers (for example, uptime), and nested dictionaries (for example, IP configuration details).

This chapter examines the NAPALM **get_facts()** method in depth, demonstrating how to retrieve key device information such as hostname, model, OS version, interfaces, and vendor details.

Finally, the chapter introduces additional NAPALM methods that are available for verification and configuration and lists methods that retrieve information as well as configuration methods. It mentions the **cli()** method for executing raw CLI commands within NAPALM.

The chapter reinforces the concept that NAPALM helps users become familiar with structured data, preparing them for working with RESTCONF, NETCONF, and other API-driven automation tools.

In the next chapter, we will examine the three types of dictionaries NAPALM uses and discuss how to work effectively with each one of them. We will also discuss nested dictionaries, where a single key holds another dictionary that contains multiple related key/value pairs.

Chapter 6. Understanding Python Dictionaries with NAPALM

Before we dive into the structure of NAPALM's data, it's important to remember why NAPALM is so useful. One of its primary goals is to provide a vendor-agnostic interface for managing network devices. Whether you're working with Cisco IOS or another vendor's devices, NAPALM's methods return data in a consistent format. This means you can retrieve device information in the same structured way, regardless of the underlying vendor or operating system.

We will begin this chapter by examining how NAPALM organizes data and the three types of dictionaries that NAPALM uses. Then we will look at how to use a **for** loop to iterate through each type of dictionary. Although this is not a book on Python, it is intended for someone who might be new to these concepts, so we cover some sample use cases.

We will continue using the same topology as in [Chapter 5, "Introducing NAPALM and Structured](#)

Data,” focusing specifically on Router-R1 (see [Figure 6-1](#)).

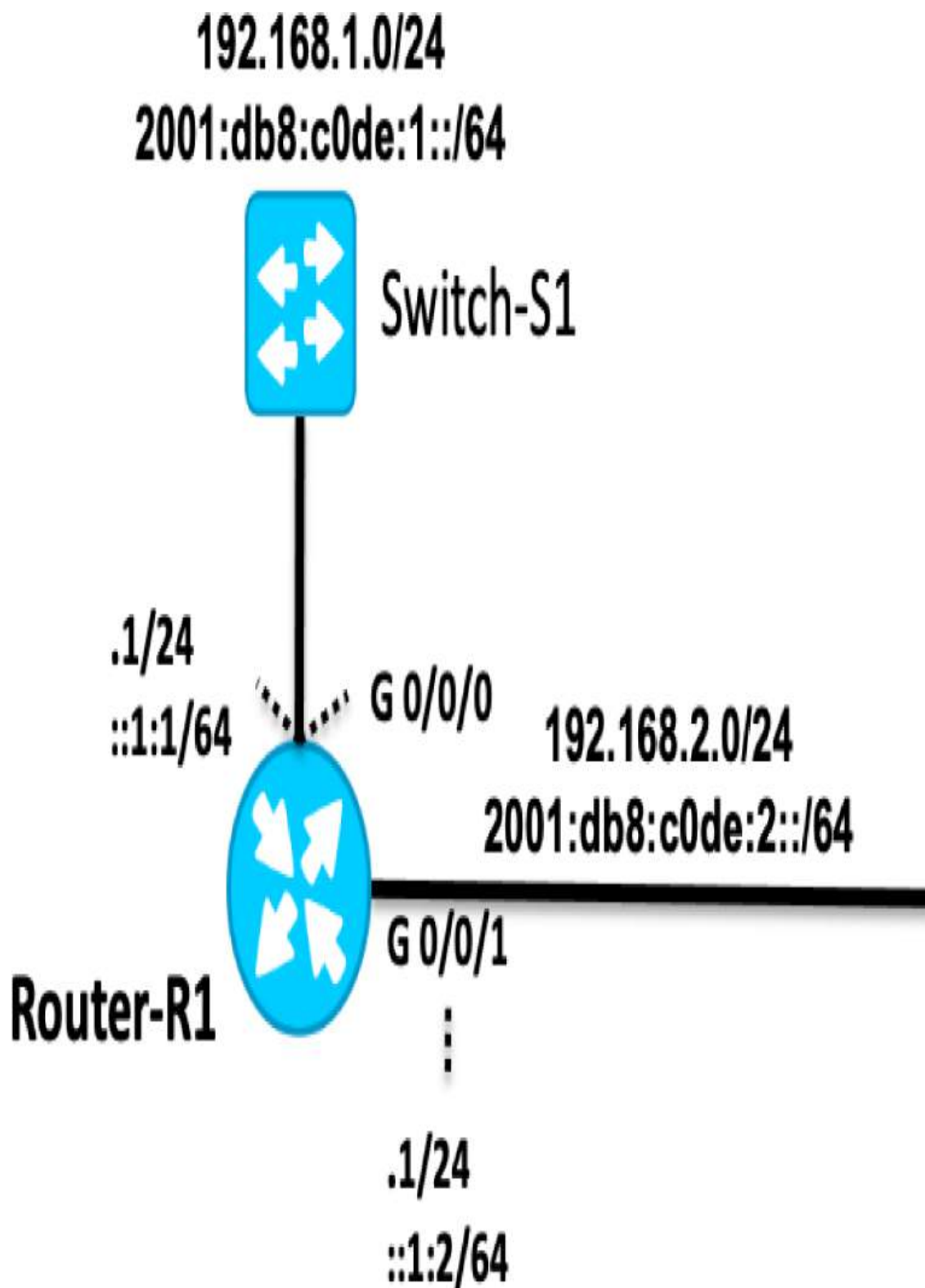


Figure 6-1 *Partial topology with Router-R1*

How NAPALM Organizes Data

NAPALM structures its retrieved data into three main data structures:

- A single main dictionary
- A dictionary of dictionaries
- A list of dictionaries

Understanding these distinctions will help you interact with and process data. As we explore each of these data structures, you will notice differences between them and see how different types of information can fit into each of these types of dictionaries.

At the end of this chapter, we include a sample program that shows all the NAPALM methods referred to in this chapter.

A Single Dictionary

The first type of NAPALM data structure is a single dictionary. A single dictionary is best used for storing a collection of related information about a single entity—whether it's a device, a person, or any other object that has various attributes.

For example, just as a person's profile might include details like height, weight, age, date of birth, and place of birth, a network device profile contains information such as hostname, model, OS version, serial number, and uptime.

A single main dictionary has a *static* structure. This means it contains a variety of different key/value pairs within a single

structure, where the keys remain the same, but the value for each key may change dynamically over time. A single main dictionary is useful for storing general device information that maintains a consistent structure while allowing updates to specific values.

get_facts(): A Single Dictionary

[Example 6-1](#) is an example of this a single main NAPALM dictionary. In this case, the **get_facts()** method, using the **device** object with the selected driver, returned the dictionary. The dictionary contains key/value pairs that provide a high-level overview of the device, including its hostname, operating system, model, vendor, and network interfaces. The structure is static so that the keys are always present, but their values can change dynamically during the operation of the device.

Example 6-1 *Example of Output Returned from the get_facts() Method*

```
pprint(device.get_facts())

{'fqdn': 'Router-R1.SSH-KEY.com',
 'hostname': 'Router-R1',
 'interface_list': ['GigabitEthernet0/0/0',
                   'GigabitEthernet0/0/1',
                   'GigabitEthernet0/0/2',
                   'GigabitEthernet0'],
 'model': 'ISR4331/K9',
 'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),
               '16.6.3, RELEASE SOFTWARE (fc8)',
 'serial_number': 'FLM2229W1R6',
```



```
'uptime': 2820.0,  
'vendor': 'Cisco'}
```

A single main dictionary is a straightforward collection of key/value pairs, where each key represents a distinct attribute of the device, and its corresponding value provides the relevant information:

```
{  
    Key_1: Value,  
    Key_2: Value,  
    Key_3: Value,  
    Etc.  
    Key_n: Value  
}
```

This simplicity makes this type of dictionary ideal for storing general device information that remains structurally consistent while allowing for dynamic updates to values such as uptime or OS version.

Viewing `get_facts()` as a Table

Sometimes it is easier to visualize a dictionary as a table that is a lot like a table you might find in a spreadsheet. [Table 6-1](#) displays the dictionary returned from the `get_facts()` method as a simple table of keys and values.

Table 6-1 *`get_facts()` Dictionary Displayed as a Table*

Key	Value
fqdn	'Router-R1.SSH-KEY.com'
hostname	'Router-R1'
interface list	'GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0'
model	'ISR4331/K9'
os_version	'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.6.3, RELEASE SOFTWARE (fc8)'
serial_number	'FLM2229W1R6'
uptime	2820.0
vendor	'Cisco'

This is a *heterogeneous dictionary* because it contains a variety of different key/value pairs, where each key represents a distinct type of information (for example, hostname, OS version, uptime). The values can be different data types; for example, **hostname** is type string, **interface_list** is type list, and **uptime** is type float. The different data types make the structure diverse, or *heterogeneous*, rather than uniform, or *homogenous*. (You will see examples of homogenous data structures later in this chapter.)

A single main dictionary is best suited for data that describes a single entity with multiple attributes—such as a network device, a user profile, or a system status report.

get_environment(): A Single Dictionary

Another single main NAPALM dictionary is returned using the **get_environment()** method, as shown in [Example 6-2](#). The output of this method is a dictionary that provides real-time status indicators such as CPU usage, memory, and temperature. The data returned in this case looks a little more complicated than the dictionary from the **get_facts()** method because it contains *nested dictionaries*.

Example 6-2 Example of Output Returned from the **get_environment()** Method

```
pprint(device.get_environment())

{'cpu': {0: {'%usage': 2.0}},
 'fans': {'invalid': {'status': True}},
 'memory': {'available_ram': 1845474744, 'used_ram': 345748260},
 'power': {'invalid': {'capacity': -1.0, 'output': -1.0, 'status': -1.0}},
 'temperature': {'invalid': {'is_alert': False,
                             'is_critical': False,
                             'temperature': -1.0}}}
```

A *nested dictionary* is a dictionary where some values are themselves dictionaries. In [Example 6-2](#), each key (such as **cpu**, **fans**, and **temperature**) contains another dictionary as its value, and that nested dictionary holds more detailed information. For example, the **'cpu'** key contains a nested dictionary with CPU usage details, and the **'temperature'** key contains another dictionary with multiple subkeys (for example, **is_alert**, **is_critical**, **temperature**). This structure allows for organized, hierarchical data storage,

making it easier to access specific details about different components.

[Example 6-3](#) will help you visualize the structure of the dictionary returned by the **get_environment()** method. [Example 6-3](#) breaks down the individual key/value pairs, presenting each top-level key separately, along with its associated nested dictionaries. It begins with each top-level key and expands to reveal the corresponding value.

If the value is a nested dictionary, it is further broken down to show how the subkeys and their values are structured. This visualization allows for a clearer understanding of how different components—such as CPU usage, memory, and temperature—are represented in a hierarchical format. In the [Chapter 7, “Iterating Through NAPALM Dictionaries,”](#) you will see how to can navigate and extract specific details when working with nested dictionaries.

Example 6-3 *get_environment():Top-Level Key/Value Pairs Containing Nested Dictionaries*

Key:	Value
'cpu':	{ Key: Value 0: { Key: Value '%usage': 2.0 } },
Key:	Value
'fans':	{ Key: Value 'invalid': { Key: Value }

	'status': True
	}
	},
Key: Value	
'memory': {	
	Key: Value
	'available_ram': 1845474744,
	'used_ram': 345748260
	},
Key: Value	
'power': {	
	Key: Value
	'invalid': {
	Key: Value
	'capacity': -1.0,
	'output': -1.0,
	'status': True
	}
	},
Key: Value	
'temperature': {	
	Key: Value
	'invalid': {
	Key: Value
	'is_alert': False,
	'is_critical': False,
	'temperature': -1.0
	}
	}

Why Use Nested Dictionaries?

A nested dictionary is used when a value itself contains multiple related pieces of information or when there are one or more logical subcategories (key/value pairs) within the data structure. This approach helps keep related data organized and easy to access.

For example, in **get_environment()**, the **cpu** key has a dictionary as its value because a device may have multiple CPUs, each with its own usage statistics. Even though there is only a **cpu 0** entry in this case, a second CPU could exist (for example, **cpu 1**), making it necessary to store CPU data in a dictionary.

Similarly, **fans** is also a dictionary because a device may have multiple cooling fans, each with an independent status. In this example, there's only an **'invalid'** fan entry, but if the device had multiple fans, each would have its own dictionary entry.

The **memory** key provides a clearer example of why a nested dictionary is useful. The device reports two separate values: **available_ram** and **used_ram**. Instead of using separate top-level keys, these values are grouped together under **memory**, which improves clarity and organization.

You might be wondering if top-level keys could instead be used (for example, **available_ram_memory** and **used_ram_memory**); the answer is yes. However, the developer decided to structure **memory** as a dictionary with two separate sub-key/value pairs.

Note

It is important to recognize that in many cases, the same data can be organized in different ways, such as using a dictionary with nested dictionaries or

without nested dictionaries, a simple list, or a list of dictionaries. Although generally some formats are better than other, how the data is organized or structured is up to the developer.

It is important to understand what the data means. The key **'invalid'** under **fans** suggests that the device does not have an identifiable fan sensor or that the fan status is not being reported correctly. However, the status value is **True**, which may indicate that the device assumes the fan is functioning properly, even though specific fan details are unavailable. Similarly, the **'invalid'** key under **temperature** means the device does not provide a valid temperature reading. A value of **-1.0** is often used as a placeholder to indicate that the temperature is not being monitored.

Overall, nested dictionaries help structure complex data logically by keeping related values grouped together. This makes it easier to retrieve and work with specific pieces of information while maintaining a clear hierarchy within the data.

Because the values in dictionaries can contain different types of data, accessing their values requires different techniques. Some values may be simple key/value pairs, and others involve lists or nested dictionaries. In [Chapter 7](#), we will explore how to use a **for** loop to iterate over the different types of dictionaries described in this chapter to extract specific information.

A Dictionary of Dictionaries

The next type of NAPALM data structure is a dictionary of dictionaries. This is a dictionary that consists of a fixed set of keys, where each key represents a unique entity (such as a network interface). The value for each key is another

dictionary with a consistent structure, meaning every entry follows the same format but holds different data.

get_interfaces(): A Dictionary of Dictionaries

[Example 6-4](#) shows an example of a dictionary of dictionaries that was returned by the NAPALM **get_interfaces()** method. Each interface, **GigabitEthernet0**, **GigabitEthernet0/0/0**, **GigabitEthernet0/0/1**, and **GigabitEthernet0/0/2**, is an individual key, that is associated with a value, which is another dictionary or nested dictionary. The nested dictionaries contain identical attributes (keys), such as the MTU, MAC address, and operational state for each interface.

While the number of interfaces, is *finite* (for example, a router may have four physical interfaces), the details about each interface—such as its status and description—can change dynamically. For example, the GigabitEthernet0/0/0 interface will always be in this dictionary for this device, but the **description** or the operational status (**is_up**) of the interface is subject to change.

Example 6-4 *Example of Output Returned from the get_interfaces() Method*

```
pprint(device.get_interfaces())

{'GigabitEthernet0': {'description': '',
                      'is_enabled': False,
                      'is_up': False,
                      'last_flapped': -1.0,
                      'mac_address': 'DC:F7:19:94:A8:BF',
                      'mtu': 1500,
```



```
        'speed': 1000.0},
'GigabitEthernet0/0/0': {'description': '',
                          'is_enabled': True,
                          'is_up': True,
                          'last_flapped': -1.0,
                          'mac_address': 'DC:F7:19:94:A8:30',
                          'mtu': 1500,
                          'speed': 100.0},
'GigabitEthernet0/0/1': {'description': '',
                          'is_enabled': True,
                          'is_up': True,
                          'last_flapped': -1.0,
                          'mac_address': 'DC:F7:19:94:A8:31',
                          'mtu': 1500,
                          'speed': 1000.0},
'GigabitEthernet0/0/2': {'description': '',
                          'is_enabled': False,
                          'is_up': False,
                          'last_flapped': -1.0,
                          'mac_address': 'DC:F7:19:94:A8:32',
                          'mtu': 1500,
                          'speed': 1000.0}}
```

What makes this dictionary of dictionaries different from the nested dictionary that we saw with **get_environment()** is that this type of dictionary is homogeneous. We call this a *homogeneous dictionary* because each key in the dictionary represents an instance of the same type of entity (for example, network interface, IP address), and the values follow a consistent structure across all entries. This consistent structure begins with the same top-level keys as interfaces: **GigabitEthernet0**, **GigabitEthernet0/0/0**, **GigabitEthernet0/0/1**, and **GigabitEthernet0/0/2**. Also,

every nested dictionary contains the same set of attributes (key/value pairs), such as MTU, MAC address, and status for interfaces, and the data remains uniform and predictable. As you will see in [Chapter 7](#), this consistent structure makes it easier to process the data programmatically.

Keys, Not Values

In [Example 6-4](#), at first glance, **GigabitEthernet0**, **GigabitEthernet0/0/0**, and so on might look like values, but they are actually keys. Compare [Example 6-4](#) with the **get_facts()** output in [Example 6-1](#), where **interface_list** is the key, and **GigabitEthernet0**, **GigabitEthernet0/0/0**, and so on are values in a list. In [Example 6-4](#), where the interfaces are keys (not values), each key, such as **GigabitEthernet0**, uniquely identifies a network interface, and its corresponding value is another dictionary. [Example 6-5](#) helps visualize this, showing how this homogeneous dictionary is structured and emphasizing that interface names are used as keys, and each value is a nested dictionary of interface attributes.

In [Examples 6-4](#) and [6-5](#), each key, such as **GigabitEthernet0** and **GigabitEthernet0/0/0**, has a nested dictionary that contains the same interface attributes (key/value pairs):

- **description:** string
- **is_enabled:** Boolean (True or False)
- **is_up:** Boolean (True or False)
- **last_flapped:** float
- **mac_address:** string
- **mtu:** integer
- **speed:** float

This structure, with exactly the same key/value pairs, is the same for every interface. This is why we call it a *homogenous* dictionary.

Example 6-5 Understanding Keys and Values

```
pprint(device.get_interfaces())
```

```
Key          : Value (dictionary)
{'GigabitEthernet0': {'description': '',
                      'is_enabled': False,
                      'is_up': False,
                      'last_flapped': -1.0,
                      'mac_address': 'DC:F7:19:94:A8:BF',
                      'mtu': 1500,
                      'speed': 1000.0},
```

```
Key          : Value (dictionary)
'GigabitEthernet0/0/0': {'description': '',
                        'is_enabled': True,
                        'is_up': True,
                        'last_flapped': -1.0,
                        'mac_address': 'DC:F7:19:94:A8:30',
                        'mtu': 1500,
                        'speed': 1000.0},
```

```
Key          : Value (dictionary)
'GigabitEthernet0/0/1': {'description': '',
                        'is_enabled': True,
                        'is_up': True,
                        'last_flapped': -1.0,
                        'mac_address': 'DC:F7:19:94:A8:31',
                        'mtu': 1500,
                        'speed': 1000.0},
```

Key	: Value (dictionary)	
'GigabitEthernet0/0/2'	{	ke
	'description': '',	ke
	'is_enabled': False,	ke
	'is_up': False,	ke
	'last_flapped': -1.0,	ke
	'mac_address': 'DC:F7:19:94:A8:32',	ke
	'mtu': 1500,	ke
	'speed': 1000.0}}	ke

Viewing a Dictionary of Dictionaries as a Table

As we saw earlier, a dictionary can sometimes be more easily visualized as a table. [Table 6-2](#) displays the dictionary returned from the **get_interfaces()** method as a table. In this case, the nested dictionaries are displayed as sub-keys and values.

Table 6-2 *get_interfaces()* Dictionary Displayed as a Spreadsheet Table

Key	Value	Sub-Key	Value
'GigabitEthernet0'	Dictionary		
		'description'	' '
		'is_enabled'	False
		'is_up'	False
		'last_flapped'	-1.0
		'mac_address'	'DC:F7:19:94:A8:BF'
		'mtu'	1500
		'speed'	1000.0
'GigabitEthernet0/0/0'	Dictionary		
		'description'	' '
		'is_enabled'	True
		'is_up'	True
		'last_flapped'	-1.0
		'mac_address'	'DC:F7:19:94:A8:30'
		'mtu'	1500

		'speed'	1000.0
'GigabitEthernet0/0/1'	Dictionary		
		'description'	' '
		'is_enabled'	True
		'is_up'	True
		'last_flapped'	-1.0
		'mac_address'	'DC:F7:19:94:A8:31'
		'mtu'	1500
		'speed'	1000.0
'GigabitEthernet0/0/2'	Dictionary		
		'description'	' '
		'is_enabled'	False
		'is_up'	False
		'last_flapped'	-1.0
		'mac_address'	'DC:F7:19:94:A8:32'
		'mtu'	1500
		'speed'	1000.0

Why Use a Dictionary of Dictionaries?

A dictionary of dictionaries is useful when each entry has a *unique key*, making it easy to access specific information. In [Example 6-4](#), that unique key is each interface:

GigabitEthernet0, **GigabitEthernet0/0/0**,
GigabitEthernet0/0/1, and **GigabitEthernet0/0/2**.

There are several advantages to using unique keys in a dictionary of dictionaries:

- **Efficient lookups:** Because each key uniquely identifies an entity (for example, an interface name or IP address), retrieving information is fast and direct.
- **Logical hierarchy:** The dictionary structure helps group related data under a single key (interface), keeping information well organized.
- **Consistent format:** Every nested dictionary follows the same structure, with the same key/value pairs, ensuring uniformity when processing multiple entries.

get_interfaces_ip(): A Dictionary of Dictionaries

Example 6-6 shows another dictionary of dictionaries. The **get_interfaces_ip()** method provides IP addressing details for each interface. Like **get_interfaces()**, the **get_interfaces_ip()** method returns a homogeneous dictionary where each interface name is a key, and the associated dictionary contains nested dictionaries for IPv4 and IPv6 addresses. However, instead of having one nested dictionary for each interface, there are three levels of nesting.

Example 6-6 Example of Output Returned from the `get_interfaces_ip()` Method

[illegible]

```
'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length': 24},
                                   '10.10.10.1': {'prefix_length': 24}},
                          'ipv6': {'2001:db8:c0de:2::1': {'prefix_length': 64},
                                   'fe80::2:1': {'prefix_length': 64}}}
```

In [Example 6-6](#), each interface key contains another dictionary, which holds separate dictionaries for IPv4 and IPv6 addresses:

- The key '**GigabitEthernet0/0/0**' contains a dictionary with two subkeys: '**ipv4**' and '**ipv6**'.
- Inside '**ipv4**', another dictionary maps the IP address (**192.168.1.1**) to its prefix length (**/24**).
- The same structure applies to IPv6, ensuring consistent formatting for both addressing schemes.

A Structured Breakdown of the `get_interfaces_ip()` Output

For those new to nested dictionaries, the output from **`get_interfaces_ip()`** might seem a bit intimidating, and it may be difficult to see the individual key/value pairs. Let's break it down. [Example 6-7](#) shows the **`get_interfaces_ip()`** output formatted to clearly show the nested dictionary structure.

Example 6-7 *Structured Breakdown of the `get_interfaces_ip()` Output*

```
pprint(device.get_interfaces_ip())

{
  Key:                Value
  'GigabitEthernet0/0/0': {
                        Key:  Value
```



```

    'ipv4': {
        Key:      Value
        '192.168.1.1': {
            Key:
            'prefix_length'
        }
    },
    Key:      Value
    'ipv6': {
        Key:      Value
        '2001:db8:c0de:1::1': {
            Key:
            'prefix_length'
        },
        Key:      Value
        'fe80::1:1': {
            Key:
            'prefix_length'
        }
    }
},

```

Etc.

The breakdown shown in [Example 6-7](#) is designed to help you understand and visualize the structure of the nested dictionary returned by **get_interfaces_ip()**. Each level of nesting represents a logical grouping of data. By following this hierarchy, you can see how interface names, IP versions, IP addresses, and prefix lengths are structured in a consistent, organized, and homogenous way.

The **get_interfaces_ip()** method returns a nested dictionary where:

- Top-level keys represent interface names (for example, **'GigabitEthernet0/0/0'**).
- Each interface contains IPv4 and IPv6 dictionaries that group assigned addresses by protocol.
- Inside the **ipv4** and **ipv6** dictionaries, the keys are IPv4 addresses, IPv6 global unicast addresses, and IPv6 link-local addresses.
- For each of these IP addresses, the values are dictionaries, each containing a single key/value pair that looks like this: **{ 'prefix_length': *value* }**.

A List of Dictionaries

The third type of NAPALM data structure is a list of dictionaries. A list of dictionaries is like a spreadsheet table because each row in the table represents a dictionary, and each column represents a key/value pair within that dictionary. Just as a spreadsheet maintains consistent column headers for structured data, each dictionary in the list follows the same structure, with identical keys acting as column headers and values corresponding to the specific data in each row.

get_mac_address_table(): A List of Dictionaries

[Example 6-8](#) shows how NAPALM uses the **get_mac_address_table()** method on an Ethernet switch (not the router we have been using so far). This method is used to retrieve the MAC table information, and the output is a list of dictionaries.

Note

For simplicity, the output in [Example 6-8](#) does not show all the information obtained by the **get_mac_address_table()** method and has been reformatted to show each dictionary on a single line.

Example 6-8 Output from the **get_mac_address_table()** Method

```
pprint(device.get_mac_address_table())
[
  {'interface': 'GigabitEthernet0/0/0', 'mac': 'DC:F7:19:94:A8:30'
  'age': -1.0},
  {'interface': 'GigabitEthernet0/0/0', 'mac': '00:E0:4C:68:05:B6'
  'age': 1.0},
  {'interface': 'GigabitEthernet0/0/1', 'mac': 'DC:F7:19:94:A8:31'
  'age': -1.0},
  {'interface': 'GigabitEthernet0/0/1', 'mac': '04:62:73:49:4B:E1'
  'age': 47.0}
]
```

This type of dictionary differs from the previous two because it is a *list*, where each element in the list is a dictionary. In the case of [Example 6-8](#), each dictionary is an entry from the switch's MAC address table.

Unlike a dictionary of dictionaries, where the keys remain static, a list of dictionaries is *dynamic*: The list of dictionaries can dynamically grow and shrink based on real-time network activity as the switch adds new entries to the MAC address table or removes entries that timeout. Each dictionary within the list follows the same structure, meaning every dictionary contains identical fields, making a list of dictionaries a *homogeneous* data structure.

Viewing a List of Dictionaries as a Table

As shown in [Table 6-3](#), a MAC address table is comparable to a simple list of items in a spreadsheet. It stores each MAC address entry (the values) as a row, and attributes (keys) such as the interface, MAC address, VLAN, and age act as columns.

Table 6-3 *MAC Address Table Displayed as a Spreadsheet Table*

Interface	MAC Address	VLAN	Age
GigabitEthernet0/0/0	DC:F7:19:94:A8:30	10	-1.0
GigabitEthernet0/0/0	00:E0:4C:68:05:B6	10	1.0
GigabitEthernet0/0/1	DC:F7:19:94:A8:31	20	-1.0
GigabitEthernet0/0/1	04:62:73:49:4B:E1	20	47.0

As you can see, each dictionary in a list of dictionaries follows the same structure, with identical keys representing the attributes. Next, we will take a closer look at the list of dictionaries returned by the **get_mac_address_table()** method.

A Closer Look at the **get_mac_address_table()** Method

Let's take a closer look at how a MAC address table is a list of dictionaries, focusing on the **get_mac_address_table()** method.

An Ethernet switch has a MAC address table that the switch builds dynamically. The switch learns MAC addresses by examining the source MAC addresses of incoming Ethernet

frames and associating each one with the port that received the frame and the VLAN ID of the port. This learning process helps a switch build the MAC address table, which is a forwarding table. After learning the source MAC address and incoming port number for a frame, the switch can forward the frame by examining its destination MAC address. If the destination MAC address is in the switch's MAC address table, the switch forwards the frame out that port. Otherwise, the switch floods the frame out all ports (with the same VLAN ID) except the existing port.

Because switches learn MAC addresses from incoming Ethernet frames, new entries are dynamically added as devices send traffic, and old entries are removed when they are no longer active.

[Example 6-9](#) shows how NAPALM uses the **get_mac_address_table()** method to retrieve the MAC table information from an Ethernet switch.

Example 6-9 *Sample Output from the **get_mac_address_table()** Method*

```
pprint(device.get_mac_address_table())
```

```
[{'active': True,
  'interface': 'Gi1/0/47',
  'last_move': -1.0,
  'mac': 'BC:B1:D3:09:F9:5C',
  'moves': -1,
  'static': False,
  'vlan': 187},
 {'active': True,
  'interface': 'Gi1/0/47',
  'last_move': -1.0,
  'mac': 'BC:B1:D3:09:F9:5C',
```

```
'moves': -1,  
'static': False,  
'vlan': 998},  
{ 'active': True,  
  'interface': 'Gi4/0/44',  
  'last_move': -1.0,  
  'mac': 'A4:18:75:92:E3:B1',  
  'moves': -1,  
  'static': False,  
  'vlan': 999}]
```

In a MAC address table, every entry contains the same structured details about a specific port-to-MAC address mapping, including these:

- **active:** Can be set to either True or False:
 - **True:** Indicates that the MAC address is currently active in the forwarding table.
 - **False:** Indicates that the MAC address is currently inactive in the forwarding table. This typically means that the switch has learned the MAC address in the past but is no longer receiving frames from it
- **interface:** Specifies the switch port where this MAC address was learned.
- **last_move:** Represents the time (in seconds) since the MAC address last moved to a different port. (**-1.0** means unknown or never moved.)
- **mac:** Specifies the learned MAC address associated with this entry.
- **moves:** Indicates how many times this MAC address has moved between interfaces. (**-1** may indicate an unknown value.)

- **static:** Specifies whether the MAC address is statically configured (**True**) or dynamically learned (**False**).

Looking again at [Table 6-3](#) will help you visualize this type of dictionary as rows and columns similar to a list in a spreadsheet.

get_arp_table(): A List of Dictionaries

Another NAPALM example that is a list of dictionaries is an Address Resolution Protocol (ARP) table. First, we'll do a quick review of ARP tables for those who may be still new to networking. This will help you understand why an ARP table is structured as a list of dictionaries.

Every device with an Ethernet or Wi-Fi NIC that also has an IPv4 address has an ARP table. An ARP table is a data structure maintained by most networked devices—including computers, routers, and switches (that have IPv4 addresses)—that maps known IPv4 addresses to their corresponding MAC addresses. When a device needs to send an IPv4 packet to a device on the *same IPv4 network* but does not know that device's MAC address, it sends an ARP request to ask for the MAC address associated with that IPv4 address. If the destination IPv4 address is on a *different IPv4 network*, the MAC address that the sender needs is that of its default gateway (router), which is responsible for forwarding the packet toward its final destination.

The device that has the requested IPv4 address responds with an ARP reply, which includes the device's MAC address. After the device that sent the ARP request receives the ARP reply, the destination IPv4 address and the newly learned associated MAC address are temporarily stored in the device's ARP table for a specific time that depends on that device's operating system.

Note

Remember that any device with an Ethernet or Wi-Fi NIC and an IPv4 address will have an ARP table.

Here are a few items to keep in mind about ARP tables:

- End devices (such as laptops) usually have both Ethernet and Wi-Fi, meaning the ARP table contains entries for both interfaces.
- Routers often have multiple Ethernet interfaces, so it is important to know which interface learned the MAC address.
- A Layer 2 switch that is configured with an IPv4 address, typically for remote management, will also participate in ARP and have an ARP table.

Because devices communicate using IPv4, new ARP entries are dynamically added as devices send traffic, and old entries are removed when they time out. [Example 6-10](#) show how NAPALM uses the **get_arp_table()** method to retrieve the ARP table information from a device—in this case, a Cisco router.

In an ARP table, every entry contains the same structured details about a specific IPv4-to-MAC address mapping, including these:

- **IPv4 address:** The destination IP address.
- **MAC address:** The MAC address associated with that IP address.
- **Interface:** The interface where this MAC address was learned.
- **Age:** The time (in seconds) since this ARP entry was last updated. A value of **-1.0** typically means that the

entry is static, meaning it was manually configured and does not age out automatically.

Example 6-10 *Sample Output from the **get_arp_table()** Method*

```
pprint(device.get_arp_table())

[{'age': -1.0,
  'interface': 'GigabitEthernet0/0/0',
  'ip': '192.168.1.1',
  'mac': 'DC:F7:19:94:A8:30'},
 {'age': 1.0,
  'interface': 'GigabitEthernet0/0/0',
  'ip': '192.168.1.10',
  'mac': '00:E0:4C:68:05:B6'},
 {'age': -1.0,
  'interface': 'GigabitEthernet0/0/1',
  'ip': '192.168.2.1',
  'mac': 'DC:F7:19:94:A8:31'},
 {'age': 47.0,
  'interface': 'GigabitEthernet0/0/1',
  'ip': '192.168.2.2',
  'mac': '04:62:73:49:4B:E1'}]
```

Once again, we can visualize this type of dictionary as a table, as shown in [Table 6-4](#).

Table 6-4 *ARP Table Displayed as a Spreadsheet Table*

Interface	IPv4 Address	MAC Address	Age (seconds)
GigabitEthernet0/0/0	192.168.1.1	DC:F7:19:94:A8:30	-1.0
GigabitEthernet0/0/0	192.168.1.10	00:E0:4C:68:05:B6	1.0
GigabitEthernet0/0/1	192.168.2.1	DC:F7:19:94:A8:31	-1.0
GigabitEthernet0/0/1	192.168.2.2	04:62:73:49:4B:E1	47.0

Much as with the MAC address table discussed previously, new ARP entries are dynamically added when the device needs the MAC address of an IPv4 address on its network, and old entries are removed when the entry times out.

A list of dictionaries is useful when the data represents a dynamic collection of similar items without the requirement of a single unique identifier. This type of dictionary is sometimes used to iterate over all items when no direct key-based access is needed.

Sample Program Using All Three Types of Dictionaries

[Example 6-11](#) shows a sample program that uses most of the NAPALM methods discussed in this chapter on a router. The only method not included is the **get_mac_address_table()** method, which would be performed on an Ethernet switch.

Example 6-11 *ex6-11_napalm_sample_dictionaries.py*

```
# 1. Import required library and module
import napalm
from pprint import pprint
```

```
# 2. Select network driver
driver = napalm.get_network_driver('ios')

# 3. Create a device object
device = driver( hostname='192.168.1.1',
                 username='admin',
                 password='cisco',
                 optional_args={'secret':'spot'} )

# 4. Establish a connection
device.open()

# 5. Implement NAPALM methods

# Single dictionary:
print('\nSingle dictionary:')

print('\nget_facts()')
print('-' * 11)
pprint(device.get_facts())

print('\nget_environment()')
print('-' * 17)
pprint(device.get_environment())

# Dictionary of dictionaries:
print('\nDictionary of dictionaries:')

print('\nget_interfaces()')
print('-' * 16)
pprint(device.get_interfaces())
```

```
print('\nget_interfaces_ip()')
print('-' * 19)
pprint(device.get_interfaces_ip())

# List of dictionaries:
print('\nList of dictionaries:')

print('\nget_arp_table()')
print('-' * 15)
pprint(device.get_arp_table())

# 6. Close the connection
device.close()
```

Example 6-12 show sample output from the program in **Example 6-11**.

Example 6-12 *Output from [Example 6-11](#)*

```
Single dictionary:

get_facts()
-----
{'fqdn': 'Router-R1.SSH-KEY.com',
 'hostname': 'Router-R1',
 'interface_list': ['GigabitEthernet0/0/0',
                    'GigabitEthernet0/0/1',
                    'GigabitEthernet0/0/2',
                    'GigabitEthernet0'],
 'model': 'ISR4331/K9',
 'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),
               '16.6.3, RELEASE SOFTWARE (fc8)',
 'serial_number': 'FLM2229W1R6',
```

```
'uptime': 2820.0,  
'vendor': 'Cisco'}
```

```
get_environment()  
-----
```

```
{'cpu': {0: {'%usage': 2.0}},  
 'fans': {'invalid': {'status': True}},  
 'memory': {'available_ram': 1845474744, 'used_ram': 345748260},  
 'power': {'invalid': {'capacity': -1.0, 'output': -1.0, 'status': -1.0}},  
 'temperature': {'invalid': {'is_alert': False,  
                             'is_critical': False,  
                             'temperature': -1.0}}}
```

Dictionary of dictionaries:

```
get_interfaces()  
-----
```

```
{'GigabitEthernet0': {'description': '',  
                      'is_enabled': False,  
                      'is_up': False,  
                      'last_flapped': -1.0,  
                      'mac_address': 'DC:F7:19:94:A8:BF',  
                      'mtu': 1500,  
                      'speed': 1000.0},  
 'GigabitEthernet0/0/0': {'description': '',  
                          'is_enabled': True,  
                          'is_up': True,  
                          'last_flapped': -1.0,  
                          'mac_address': 'DC:F7:19:94:A8:30',  
                          'mtu': 1500,  
                          'speed': 100.0},  
 'GigabitEthernet0/0/1': {'description': '',
```

```

        'is_enabled': True,
        'is_up': True,
        'last_flapped': -1.0,
        'mac_address': 'DC:F7:19:94:A8:31',
        'mtu': 1500,
        'speed': 1000.0},
    'GigabitEthernet0/0/2': {'description': '',
        'is_enabled': False,
        'is_up': False,
        'last_flapped': -1.0,
        'mac_address': 'DC:F7:19:94:A8:32',
        'mtu': 1500,
        'speed': 1000.0}}

get_interfaces_ip()
-----
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length': 24},
    'ipv6': {'2001:db8:c0de:1::1': {'prefix_length': 64},
    'fe80::1:1': {'prefix_length': 64}},
    'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length': 24},
    'ipv6': {'2001:db8:c0de:2::1': {'prefix_length': 64},
    'fe80::2:1': {'prefix_length': 64}},

List of dictionaries:

get_arp_table()
-----

[{'age': -1.0,
    'interface': 'GigabitEthernet0/0/0',
    'ip': '192.168.1.1',
    'mac': 'DC:F7:19:94:A8:30'},

```

```
{'age': 1.0,  
  'interface': 'GigabitEthernet0/0/0',  
  'ip': '192.168.1.10',  
  'mac': '00:E0:4C:68:05:B6'},  
{'age': -1.0,  
  'interface': 'GigabitEthernet0/0/1',  
  'ip': '192.168.2.1',  
  'mac': 'DC:F7:19:94:A8:31'},  
{'age': 47.0,  
  'interface': 'GigabitEthernet0/0/1',  
  'ip': '192.168.2.2',  
  'mac': '04:62:73:49:4B:E1'}]
```

Comparing the Three Types of Dictionaries

In general, the same data can be organized and stored in various ways, depending on the developer's goals and the needs of the end user. It is the developer's responsibility to structure the data in a way that aligns with the intended purpose, ensuring that it is both efficient and intuitive for those who will interact with it. This involves considering factors such as the type of data being represented, how it will be accessed or queried, and how the users will be using it. Ultimately, the chosen structure should strike a balance between usability, performance, and maintainability to best serve its intended use case.

Table 6-5 summarizes the three primary dictionary structures used in NAPALM, highlighting their characteristics, best use cases, and relevant methods. A *single main dictionary* is best for storing diverse attributes about a single entity, such as a device's system information.

A *dictionary of dictionaries* organizes multiple similar entities, such as network interfaces, where each key represents a unique item with structured details. A *list of dictionaries* is useful for dynamically changing datasets, such as ARP or MAC address tables, where entries follow the same structure and don't require a single unique key. Understanding these distinctions can help in understanding how network data can be organized and accessed.

Table 6-5 *Comparison of Different NAPALM Data Structures*

Dictionary Type	Structure and Characteristics	Best Used For	Examples of NAPALM Methods
Single main dictionary (heterogeneous)	A single dictionary with multiple unique key/value pairs, where each key represents a different type of data	Storing attributes about a single entity, such as a device's hostname, model, OS version, and uptime	<code>get_facts()</code> , <code>get_environment()</code>
Dictionary of dictionaries (homogeneous)	A dictionary where each key represents a unique entity (for example, an interface), and each value is another dictionary with a consistent structure	Grouping multiple similar entities (for example, interfaces on a device), where each entry follows the same format	<code>get_interfaces()</code> , <code>get_interfaces_ip()</code>
List of dictionaries (typically homogeneous)	A list where each element is a dictionary containing identical fields, and the number of entries changes dynamically	Storing tabular data without a single unique key, such as ARP tables or MAC address tables	<code>get_arp_table()</code> , <code>get_mac_address_table()</code>

To more easily compare the three types of dictionaries, [Table 6-6](#) through [Table 6-8](#) show examples of how the dictionaries can be visualized as tables. [Table 6-6](#) shows a single main dictionary (heterogeneous), [Table 6-7](#) shows a dictionary of dictionaries (homogeneous), and [Table 6-8](#) shows a list of dictionaries (typically homogeneous).

Table 6-6 `get_facts()` Dictionary Displayed as a Table
(Single Main Dictionary)

Key	Value
fqdn	'Router-R1.SSH-KEY.com'
hostname	'Router-R1'
interface list	'GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0'
model	'ISR4331/K9'
os_version	'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.6.3, RELEASE SOFTWARE (fc8)'
serial_number	FLM2229W1R6'
uptime	2820.0
vendor	'Cisco'

Table 6-7 `get_interfaces()` Dictionary Displayed as a Table
(Dictionary of Dictionaries)

Key	Value	Sub-Key	Value
'GigabitEthernet0'	Dictionary		
		'description'	' '
		'is_enabled'	False
		'is_up'	False
		'last_flapped'	-1.0
		'mac_address'	'DC:F7:19:94:A8:BF'
		'mtu'	1500
		'speed'	1000.0
'GigabitEthernet0/0/0'	Dictionary		
		'description'	' '
		'is_enabled'	True
		'is_up'	True
		'last_flapped'	-1.0
		'mac_address'	'DC:F7:19:94:A8:30'
		'mtu'	1500
		'speed'	1000.0
Etc.			

Table 6-8 *MAC Address Table Displayed as a Table (List of Dictionaries)*

Interface	MAC Address	VLAN	Age
GigabitEthernet0/0/0	DC:F7:19:94:A8:30	10	-1.0
GigabitEthernet0/0/0	00:E0:4C:68:05:B6	10	1.0
GigabitEthernet0/0/1	DC:F7:19:94:A8:31	20	-1.0
GigabitEthernet0/0/1	04:62:73:49:4B:E1	20	47.0

Methods by Type of Data Structure

Finally, this section lists methods categorization based on the three dictionary structures discussed. The specific category may vary depending on the implementation.

Single Main Dictionary

Each of these methods returns a single dictionary that contains key/value pairs representing general device information:

- **get_facts():** Retrieves general device information, such as hostname, model, OS version, serial number, vendor, uptime, and interfaces.
- **get_environment():** Retrieves environmental metrics such as CPU usage, memory utilization, and temperature.
- **get_config():** Retrieves the current device configuration, including startup, running, and candidate configurations.
- **get_users():** Returns a list of configured local users and their privilege levels.
- **get_snmp_information():** Retrieves SNMP configuration details, including SNMP communities,

location, and contact information.

- **is_alive():** Checks whether the device is reachable and responding.

Dictionary of Dictionaries

Each of these methods returns a dictionary where each key maps to another dictionary that contains structured information.

These methods are used for interface and networking information:

- **get_interfaces():** Returns the status and settings of all interfaces, including speed, MAC address, and administrative status.
- **get_interfaces_counters():** Provides interface-level statistics, including packet counts, errors, and discards.
- **get_interfaces_ip():** Retrieves the assigned IP address for each interface.
- **get_lldp_neighbors():** Provides LLDP neighbor information, listing directly connected devices.
- **get_lldp_neighbors_detail():** Returns detailed LLDP neighbor information, including remote port descriptions and capabilities.

These methods are used for routing and network instances:

- **get_bgp_config():** Retrieves the configured BGP settings for the device.
- **get_bgp_neighbors():** Provides BGP neighbor information, including session state and AS numbers.
- **get_bgp_neighbors_detail():** Returns detailed information about BGP neighbors, including policies and

attributes.

- **get_network_instances():** Retrieves information about virtual routing and forwarding instances on the device.
- **get_route_to(destination):** Returns routing table information for a specific destination.

This method is used for Layer 2 and neighbor discovery:

- **get_vlans():** Returns VLAN information, including VLAN IDs and associated interfaces.

These methods are used for optical and NTP information:

- **get_ntp_peers():** Returns a list of configured NTP peers.
- **get_ntp_servers():** Retrieves configured NTP servers.

List of Dictionaries

Each of these methods returns a list where each item is a dictionary that contains structured data entries.

These methods are used for Layer 2 and neighbor discovery:

- **get_arp_table():** Returns the ARP table entries, including MAC addresses, IP addresses, and interface mappings.
- **get_ipv6_neighbors_table():** Provides IPv6 neighbor discovery table entries. (This table is similar to an ARP table for IPv4.)
- **get_mac_address_table():** Retrieves the MAC address table, listing learned MAC addresses and their associated interfaces.

These methods are used for optical and NTP information:

- **get_optics():** Retrieves optical transceiver details, including power levels and operational status.
- **get_ntp_stats():** Provides NTP synchronization details and peer status.

These methods are used for connectivity and testing:

- **ping(destination):** Sends ICMP pings to a destination and returns response statistics.
- **traceroute(destination):** Performs a traceroute to a specified IP address, showing the path packets take through the network.

Note

Some methods, including the following, do not return data but are used to configure a device:

- **load_merge_candidate(filename=None, config=None)**
- **load_replace_candidate(filename=None, config=None)**
- **compare_config()**
- **commit_config()**
- **discard_config()**
- **rollback()**

You will learn how to configure devices using these methods in [Chapter 8, “Configuring Devices with NAPALM.”](#)

Summary

This chapter introduces the three primary dictionary structures used in NAPALM: a single main dictionary, a

dictionary of dictionaries, and a list of dictionaries. Each of these structures organizes network data differently, depending on whether the data describes a single entity, multiple structured entities, or a dynamic set of similar entries:

- A *single main dictionary* is a straightforward collection of key/value pairs that is best suited for storing diverse attributes about a single device, such as hostname, OS version, and uptime. This structure is static, with consistent keys and dynamic values.
- A *dictionary of dictionaries* is ideal for organizing multiple similar entities, such as network interfaces. Each key represents a unique entity (for example, an interface name), and its value is a dictionary with structured attributes like MTU, status, and MAC address. This structure enables efficient lookups and maintains a logical hierarchy.
- A *list of dictionaries* represents tabular data, where each dictionary follows the same structure but lacks a natural unique key. This format is useful for dynamically changing datasets, such as ARP tables and MAC address tables, which grow and shrink as devices communicate.

By understanding these dictionary types, network administrators can better understand how and why data is formatted as it is, and they can efficiently retrieve, process, and analyze network data by using NAPALM's Python-based methods.

Chapter 7. Iterating Through NAPALM Dictionaries

In this chapter, we will examine how to loop through all the key/value pairs of a dictionary across the three types of dictionaries discussed in the previous chapter:

- Single heterogeneous dictionary
- Dictionary of dictionaries
- List of dictionaries

For someone who is new to Python, some of this may seem a bit daunting at first. Just take it step by step and remember that the more you work with iterating through nested dictionaries, the more familiar they will become.

Note

You might consider reading this chapter after reading [Chapter 8, “Configuring Devices with NAPALM,”](#) or just skimming it for now and coming back to it later if you need more details.

Before we begin iterating through structured data, it's helpful to review the common types of dictionaries that NAPALM returns. Each type requires

a slightly different approach when using loops in Python. [Table 7-1](#) lists the three most common dictionary types you'll encounter and provides examples of when they are used and which NAPALM method returns each of them.

Table 7-1 *Common Types of NAPALM Dictionaries*

Dictionary Type	Structure	Use Case Example	Example of a NAPALM Method
Single heterogeneous dictionary	A single dictionary with mixed key/value types (strings, numbers, lists, and so on)	General device information	<code>get_facts()</code>
Dictionary of dictionaries	A dictionary where each key maps to another dictionary that has the same structure	Interface status or attributes	<code>get_interfaces()</code>
List of dictionaries	A list in which each item is a dictionary with the same set of keys	ARP table entries or route entries	<code>get_arp_table()</code>

To reinforce these dictionary types, [Table 7-2](#) shows a short example of how each one is typically structured in Python. (Refer to [Chapter 6](#), “[Understanding Python Dictionaries with NAPALM](#),” for more details.)

Table 7-2 *Example of Each Type of Dictionary*

Type	Python Format	Sample Data
Single dictionary	<code>{'hostname': 'R1', 'uptime': 3500 }</code>	Simple key/value pairs
Dictionary of dictionaries	<code>{'Gig0/0': {'mac': 'abc', 'mtu': 1500 } }</code>	Interface data
List of dictionaries	<code>[{'ip': '192.168.1.1', 'mac': 'aa:bb' }, {...}]</code>	ARP table rows

Because each dictionary type has its own structure and variations, there is no single piece of code that works universally across all dictionaries of that type. However, by the end of this chapter, you will understand how to approach each type systematically and retrieve the information you need.

These dictionary techniques are especially useful when building network reports, dashboards, or data exports.

Live or Simulated Data

In [Chapter 5, “Introducing NAPALM and Structured Data,”](#) we introduced the concept of working with a predefined dictionary to simulate NAPALM’s structured output for those who may not have access to a Cisco device. In this chapter, we revisit this idea as we begin iterating through NAPALM dictionaries.

[Example 7-1](#) retrieves live device data using **device.get_facts()**, and [Example 7-2](#) defines a manually created dictionary with the same structure.

Example 7-1 *ex7-1_get_facts.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
```

```
        optional_args={'secret':'spot'}
    )

device.open()

device_facts = device.get_facts()

pprint(device_facts)

device.close()
```

Example 7-2 *ex7-2_device_facts.py*

```
from pprint import pprint

device_facts = {
    'fqdn': 'Router-R1.SSH-KEY.com',
    'hostname': 'Router-R1',
    'interface_list': [ 'GigabitEthernet0/0/0',
                        'GigabitEthernet0/0/1',
                        'GigabitEthernet0/0/2',
                        'GigabitEthernet0'],
    'model': 'ISR4331/K9',
    'os_version': 'ISR Software (X86_64_LINUX_IOSD-1
                  'Version 16.6.3, RELEASE SOFTWARE
    'serial_number': 'FLM2229W1R6',
    'uptime': 2820.0,
    'vendor': 'Cisco'
}

pprint(device_facts)
```

The two programs produce identical output, as shown in [Example 7-3](#).

Example 7-3 *Output from [Examples 7-1](#) and [7-2](#)*

```
{'fqdn': 'Router-R1.SSH-KEY.com',  
  'hostname': 'Router-R1',  
  'interface_list': ['GigabitEthernet0/0/0',  
                    'GigabitEthernet0/0/1',  
                    'GigabitEthernet0/0/2',  
                    'GigabitEthernet0'],  
  'model': 'ISR4331/K9',  
  'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),  
                '16.6.3, RELEASE SOFTWARE (fc8)',  
  'serial_number': 'FD02302A08A',  
  'uptime': 3900.0,  
  'vendor': 'Cisco'}
```

The results are the same, regardless of which approach you use. You can follow all the exercises and examples in this chapter by using either approach. If you don't have access to a Cisco device, you can copy and use the predefined dictionaries provided in this chapter to practice iterating through NAPALM data.

Working with .keys(), .values(), and .items() in NAPALM Dictionaries

When working with dictionaries in Python, you often need to access different parts of the data. Python provides three methods for this:

- **.keys()** returns all the dictionary's keys, allowing you to see what data points are available.

- **.values()** returns all the values, giving you direct access to the stored information.
- **.items()** returns key/value pairs as tuples. This is the most useful method for iterating over dictionaries.

Note

A *tuple* in Python is an ordered, immutable collection of values, meaning its elements cannot be changed after the tuple is created. A tuple is typically used to store related pieces of data together.

These methods allow you to efficiently navigate NAPALM's structured data, whether you retrieve it from a live device using **get_facts()** or work with a predefined dictionary.

[Example 7-4](#) demonstrates how all three dictionary methods, **.keys()**, **.values()**, and **.items()**, can be used with the dictionary returned by the **get_facts()** method and assigned to the variable **device_facts**.

Example 7-4 *ex7-4_dictionary_methods.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)
```

```
device.open()

device_facts = device.get_facts()

# Example of using .keys() method
print('\nKeys in the dictionary:')
print(device_facts.keys())

print('\nKeys in the dictionary using a for loop:')
for key in device_facts.keys():
    print(f'Key: {key}')

# Example of using .values() method
print('\nValues in the dictionary:')
print(device_facts.values())

print('\nValues in the dictionary using a for loop:')
for value in device_facts.values():
    print(f'Value: {value}')

# Example of using .items() method
print('\nKey-Value pairs in the dictionary:')
print(device_facts.items())

print('\nKey-Value pairs in the dictionary using a for loop:')
for key, value in device_facts.items():
    print(f'{key}: {value}')

device.close()
```

The following sections show snippets of the code in [Example 7-4](#) and the specific output of each snippet.

Using .keys() to Retrieve All Keys

[Example 7-5](#) shows how the **.keys()** method uses the **print()** function and the **device_facts** variable to return a list-like view of all the dictionary's keys, which allows you to see what attributes are available. This method is useful when you need to check what type of data is available in a dictionary, but it does not provide access to the values themselves.

Example 7-5 Using the .keys() Dictionary Method

```
print(device_facts.keys())

dict_keys(['fqdn', 'hostname', 'interface_list', 'model', 'os_ver',
'serial_number', 'uptime', 'vendor'])
```

Note

The **.keys()**, **.values()**, or **.items()** method displays a dictionary as **dict_keys([...])** to indicate that it's not a regular list but a special *dictionary view object*. You cannot use the **pprint()** function on this type of object. Because dictionary view objects are difficult to read, they are not commonly used in this manner. The next section shows how to use a **for** loop to more easily access the dictionary components.

Using .values() to Retrieve All Values

The **.values()** method returns a list-like view of all the values stored in a dictionary, as shown in [Example 7-6](#). This method is helpful when you only care about the stored data

itself, but it does not indicate which key each value belongs to.

Example 7-6 *Using the **.values()** Dictionary Method*

```
print(device_facts.values())
```

```
dict_values(['Router-R1.SSH-KEY.com', 'Router-R1', ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0/0/3'], 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.6.3 RELEASE SOFTWARE (fc8)', 'FLM2229W1R6', 2820.0, 'Cisco'])
```

Using .items() to Retrieve Key/Value Pairs

The **.items()** method, as shown in [Example 7-7](#), returns key/value pairs as tuples. It is the most common and most useful method for iterating through dictionaries. The **.items()** method makes it possible to process both the keys and values in a single **for** loop and provides structured output, filtering capabilities, and simplified data manipulation, especially when working with NAPALM dictionaries. **.items()** is the preferred method for working with NAPALM dictionaries because it enables structured output, filtering, and further data manipulation.

Example 7-7 *Using the **.items()** Dictionary Method*

```
print(device_facts.items())
```

```
dict_items([('fqdn', 'Router-R1.SSH-KEY.com'), ('hostname', 'Router-R1'), ('interface_list', ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0/0/3']), ('model', 'ISR4331-K9'), ('os_version', 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.6.3 RELEASE SOFTWARE (fc8)')])
```



```
RELEASE SOFTWARE (fc8)'), ('serial_number', 'FLM2229W1R6'), ('u  
('vendor', 'Cisco']])
```

It can be difficult to examine the results of the **.keys()**, **.values()**, and **.items()** methods by using the **print()** function alone. A **for** loop makes this output easier to read and understand.

Using a for Loop with Dictionary Methods

Continuing with our program in [Example 7-4](#), we will now examine how a **for** loop can be used with these same methods to make it easier to access each of the dictionary components.

Using a for Loop with .keys() to Retrieve Keys

The **.keys()** method allows you to iterate through only the keys of the dictionary, which means it is useful when you need to analyze or check attribute names without accessing their corresponding values. It is particularly helpful when verifying the presence of specific attributes, such as checking whether **'os_version'** exists in the dictionary before performing an operation.

[Example 7-8](#) shows how a **for** loop is used to iterate over the dictionary keys and print each one on a separate line. This example uses the variable **key**, but any valid variable name could be used. The **print(f'Key: {key}')** statement formats and prints each key in a readable way.

Example 7-8 *Using a **for** Loop and **.keys()***

```
print('\nKeys in the dictionary with for loop:')
for key in device_facts.keys():
    print(f'Key: {key}')
```

```
Keys in the dictionary with for loop:
Key: fqdn
Key: hostname
Key: interface_list
Key: model
Key: os_version
Key: serial_number
Key: uptime
Key: vendor
```

Using a for Loop with `.values()` to Retrieve Values

The `.values()` method allows you to iterate through only the values stored in the dictionary, without referencing their associated keys. This is useful when you are primarily concerned with analyzing or processing the data itself rather than the attribute names. For example, if you want to inspect all device details retrieved from NAPALM without needing to know their corresponding keys, you can iterate over `.values()` as a quick way to access the raw data.

[Example 7-9](#) uses a **for** loop to iterate over the dictionary values and print each one. In this snippet of code from [Example 7-4](#), the variable **value** is used, but again, any valid variable name could be used.

Example 7-9 *Using a **for** Loop and `.values()`*

k, **v**, **attribute**, or **data**, but as we saw earlier, **key** and **value** are commonly used for clarity.

Example 7-10 Using a **for** Loop and **.items()**

```
for key, value in device_facts.items():  
    print(f'{key}: {value}')
```



```
fqdn: Router-R1.SSH-KEY.com  
hostname: Router-R1  
interface_list: ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1'  
'GigabitEthernet0/0/2', 'GigabitEthernet0']  
model: ISR4331/K9  
os_version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Ver:  
RELEASE SOFTWARE (fc8)  
serial_number: FLM2229W1R6  
uptime: 2820.0  
vendor: Cisco
```

During each iteration, the loop processes one key/value pair at a time. On the first iteration, **key** holds '**fqdn**', and **value** holds '**Router-R1.SSH-KEY.com**'. On the next iteration, **key** becomes '**hostname**', and **value** becomes '**Router-R1**'. This continues until every key/value pair in the dictionary has been processed.

Summarizing the Three Dictionary Methods in a for Loop

To summarize, a **for** loop can be used with **.keys()** and **.values()**, but these methods only return the keys or values, respectively, without pairing them together. A **for** loop using **.keys()** has access only to the keys and not the

values, whereas a **for** loop using **.values()** has access only to the values and not the keys. These methods are typically used for specific use cases. The **.items()** method is the most practical method for iterating through structured network data as it allows access to both the keys and their associated values.

[Table 7-3](#) provides a comparison of the three dictionary methods.

Table 7-3 *Comparison of the Three Dictionary Methods*

Method	Usage in NAPALM	Common Scenarios
.keys()	Less common	If you only need to check for specific attributes, such as whether 'os_version' is in fact .keys()
.values()	Less common	If you're searching for a specific value but don't need the keys
.items()	Most common	When iterating over facts, configurations, or interface data

[Example 7-11](#) shows the complete output from the code in [Example 7-4](#).

Example 7-11 *Complete Output from [Example 7-4](#)*

```
MyPrompt% python3 ex7-4_dictionary_methods.py

Keys in the dictionary:
dict_keys(['fqdn', 'hostname', 'interface_list', 'model', 'os_version', 'serial_number', 'uptime', 'vendor'])

Keys in the dictionary using a for loop:
Key: fqdn
Key: hostname
Key: interface_list
Key: model
Key: os_version
Key: serial_number
```

Key: uptime

Key: vendor

Values in the dictionary:

```
dict_values(['Router-R1.SSH-KEY.com', 'Router-R1', ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0/0/3'], 'ISR4331/K9', 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 12.2(4)T2, RELEASE SOFTWARE (fc8)', 'FLM2229W1R6', 2820.0, 'Cisco'])
```

Values in the dictionary using a for loop:

Value: Router-R1.SSH-KEY.com

Value: Router-R1

Value: ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0/0/3']

Value: ISR4331/K9

Value: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 12.2(4)T2, RELEASE SOFTWARE (fc8)

Value: FLM2229W1R6

Value: 2820.0

Value: Cisco

Key-Value pairs in the dictionary:

```
dict_items([('fqdn', 'Router-R1.SSH-KEY.com'), ('hostname', 'Router-R1'), ('interface_list', ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0/0/3']), ('model', 'ISR4331/K9'), ('os_version', 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 12.2(4)T2, RELEASE SOFTWARE (fc8)'), ('serial_number', 'FLM2229W1R6'), ('uptime', 2820.0), ('vendor', 'Cisco')])
```

Key-Value pairs in the dictionary using a for loop:

fqdn: Router-R1.SSH-KEY.com

hostname: Router-R1

interface_list: ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0/0/3']

```
'GigabitEthernet0/0/2', 'GigabitEthernet0']  
model: ISR4331/K9  
os_version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Ver:  
RELEASE SOFTWARE (fc8)  
serial_number: FLM2229W1R6  
uptime: 2820.0  
vendor: Cisco  
MyPrompt%
```

Determining the Type of Value

Determining the type of a dictionary value is important because different data types often require different formatting or handling when displayed. For example, strings and numbers can be printed directly, whereas lists and nested dictionaries may need to be iterated through or presented line by line for readability.

The type of value associated with a key in a dictionary may vary. In the previous example, the **get_facts()** method returned three types of key/value pairs, as shown in [Table 7-4](#).

Table 7-4 *Type of Values for **get_facts()***

Type	Key	Value
String	fqdn	Router-R1.SSH-KEY.com
	hostname	Router-R1
	model	ISR4331/K9
	os_version	ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.6.3, RELEASE SOFTWARE (fc8)
	serial_number	FLM2229W1R6
	vendor	Cisco
Float	uptime	2820.0
List	interface_list	['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0']

In addition to strings, floats, and lists, dictionary values may also include nested dictionaries, integers, Booleans, or other valid Python data types. As you will see, the way you access and display each value depends on its type. Before processing a value, it is important to know its type to ensure that you extract all relevant data correctly and display it in a readable format.

Python provides two common functions to determine the type of a variable or dictionary value:

- **type(value):** Returns the type of an object, such as **str** (string), **list**, **float**, **bool** (Boolean), or **int** (integer).
- **isinstance(value, type):** Returns **True** or **False**, depending on whether the object is an instance of the specified type.

Using the type() Function (Less Preferred Approach)

It is possible to modify the code by using either [Example 7-1](#) or [Example 7-2](#), depending on whether you wish to use live or simulated data. [Example 7-12](#) is a modification of

[Example 7-1](#) that includes a **for** loop using the **.items()** function to evaluate each value and determine its type.

Example 7-12 ex7-12_type_function.py

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)

device.open()

device_facts = device.get_facts()

for key, value in device_facts.items():
    print(f'{key} is of type {type(value)}')

device.close()
```

[Example 7-13](#) shows the output, which displays the type of each value in the dictionary.

Note

Dictionary iteration follows insertion order in Python 3.7 and later. NAPALM may build its fact dictionary in different key-insertion orders, depending on the network operating system, driver, or NAPALM

version. Therefore, the printed order of keys can vary. You can use the **sorted()** function—for example, **sorted(device_facts.items())**—when you need the output to be in alphabetical order.

Example 7-13 Output from [Example 7-12](#)

```
MyPrompt % python3 ex7-12_type_function.py
fqdn is of type <class 'str'>
hostname is of type <class 'str'>
interface_list is of type <class 'list'>
model is of type <class 'str'>
os_version is of type <class 'str'>
serial_number is of type <class 'str'>
uptime is of type <class 'float'>
vendor is of type <class 'str'>
MyPrompt %
```

You might wonder why each type is displayed as **<class 'type'>**. It's because Python treats everything as an object, including dictionary values. The **type()** function returns the class of the object, which is why it explicitly states **<class 'str'>** (for string), **<class 'float'>**, or **<class 'list'>** instead of just **'str'**, **'float'**, or **'list'**.

[Example 7-14](#) shows how you can use the **type()** function to compare the dictionary value associated with a key such as **interface_list**, **device_facts['interface_list']**, with each type. **condition == list:** checks against the type object, much like what is displayed in [Example 7-13](#). The **if** statement evaluates the type of the **interface_list** value and returns either **True** or **False**. If **True**, the indented statement is executed; if **False**, the program moves to the next condition. Because the value of **interface_list** is of type **list**, the output would be **'interface_list is a list'**.

Example 7-14 *ex7-14_interface_list_type.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)

device.open()

device_facts = device.get_facts()

if type(device_facts['interface_list']) == list:
    print('interface_list is a list')
elif type(device_facts['interface_list']) == dict:
    print('interface_list is a dictionary')
elif type(device_facts['interface_list']) == float:
    print('interface_list is a float')
elif type(device_facts['interface_list']) == int:
    print('interface_list is an integer')
elif type(device_facts['interface_list']) == bool:
    print('interface_list is a boolean')
elif type(device_facts['interface_list']) == str:
    print('interface_list is a string')
else:
    print('Unknown data type')
```

```
device.close()
```

Note

Remember that in Python, dictionary values are accessed using the syntax ***dictionary_name['key']***. In this case, ***device_facts['interface_list']*** retrieves the value associated with the key '***interface_list***', allowing you to check its type by using the ***type()*** method.

[Example 7-15](#) shows the output you get after executing the code in [Example 7-14](#).

Example 7-15 *Output from [Example 7-14](#)*

```
MyPrompt % python3 ex7-14_interface_list_type.py
interface_list is a list
MyPrompt %
```

[Figure 7-1](#) illustrates this process. As each value is evaluated, the program checks its type and follows a logical path to determine how the data should be handled or displayed.

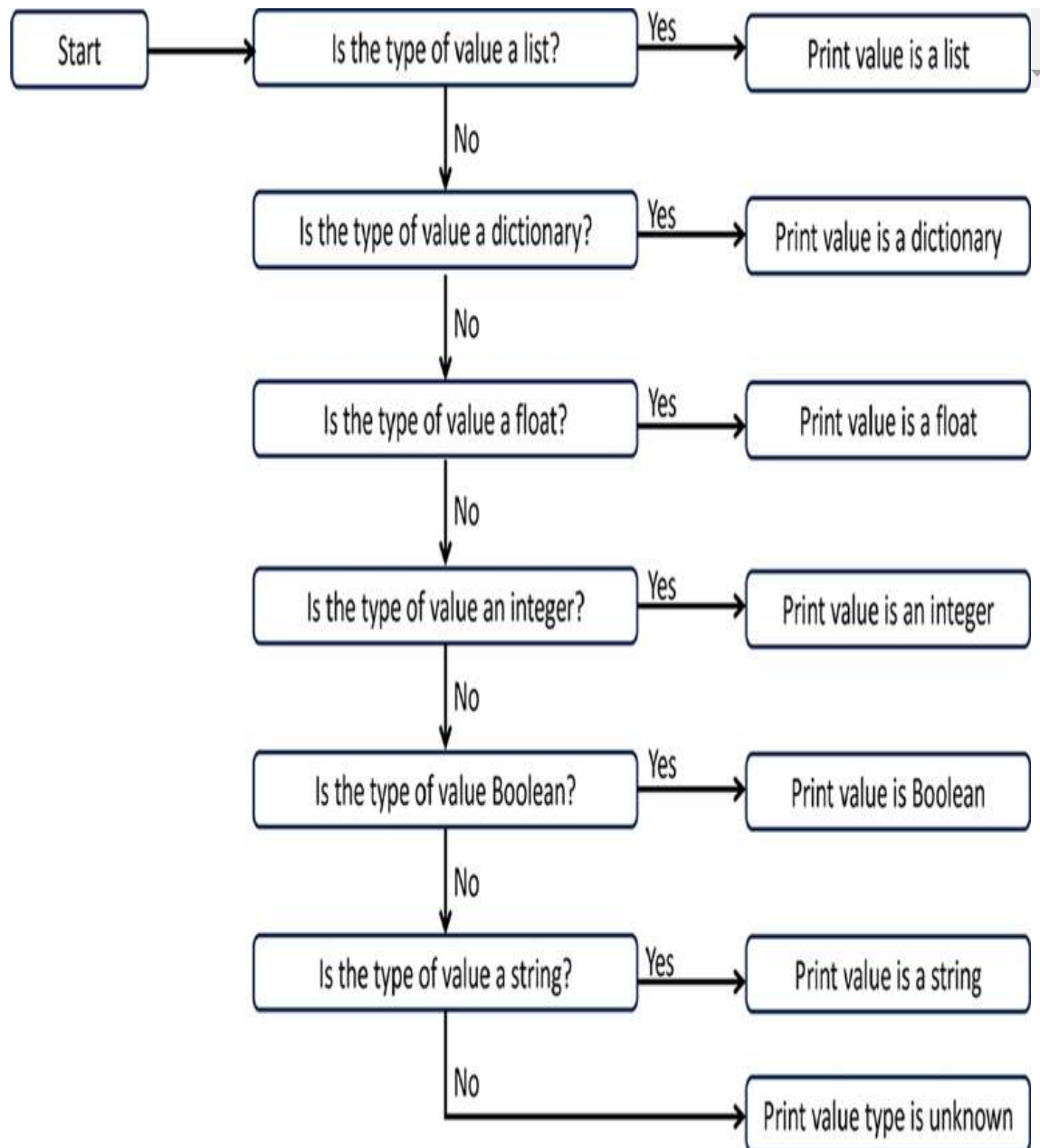


Figure 7-1 *The value decision process*

Using the `isinstance()` Function (Preferred Approach)

While the `type()` function works for checking value types, the preferred approach in Python is to use `isinstance()`.

Unlike the **type()** function, which returns the type of an object (such as **<class 'str'>** or **<class 'list'>**), the **isinstance()** function returns a Boolean value (**True** or **False**), based on whether the object matches the specified type in the function call. This makes **isinstance()** useful for conditional checks where you need to verify a value's type before processing it.

This is the syntax for the **isinstance()** function:

```
isinstance(object, type)
```

It returns **True** if the object is an instance of the specified type and **False** otherwise. Here is an example:

```
isinstance(device_facts['interface_list'], list)
```

This function checks whether the value associated with the key **'interface_list'** is a list and returns **True**. [Example 7-16](#) shows other types of values you can check for.

Example 7-16 *The **isinstance()** Function Returns **True** or **False***

```
print(isinstance(device_facts['interface_list'], list)) # Retu
print(isinstance(device_facts['interface_list'], dict)) # Retu
print(isinstance(device_facts['interface_list'], float)) # Retu
print(isinstance(device_facts['interface_list'], int)) # Retu
print(isinstance(device_facts['interface_list'], bool)) # Retu
print(isinstance(device_facts['interface_list'], str)) # Retu
```

[Example 7-17](#) demonstrates how **isinstance()** can be used to check whether the dictionary value **device_facts['interface_list']** matches a specific type.

Example 7-17 *ex7-17_isinstance_function.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)

device.open()

device_facts = device.get_facts()

if device_facts['interface_list'] is None:
    print('interface_list has no data (None)')
elif isinstance(device_facts['interface_list'], list):
    print('interface_list is a list')
elif isinstance(device_facts['interface_list'], dict):
    print('interface_list is a dictionary')
elif isinstance(device_facts['interface_list'], float):
    print('interface_list is a float')
elif isinstance(device_facts['interface_list'], int):
    print('interface_list is an integer')
elif isinstance(device_facts['interface_list'], bool):
    print('interface_list is a boolean')
elif isinstance(device_facts['interface_list'], str):
    print('interface_list is a string')
else:
    print('Unknown data type')
```

```
device.close()
```

[Example 7-18](#) shows that the output of [Example 7-17](#) would be the same, **interface_list is a list**.

Example 7-18 *Output from [Example 7-17](#)*

```
MyPrompt % python3 ex7-17_isinstance_function.py
interface_list is a list
MyPrompt %
```

The **isinstance()** function is more flexible than **type()** because it allows you to check against multiple types and works with subclass relationships, making it more robust for evaluating dictionary values. However, for a beginning Python user, multiple types and subclasses are usually not relevant, and **isinstance()** is preferred over **type()** because it makes code easier to read, avoids unnecessary complexity when checking for **None**, and works well with both built-in and user-defined objects, ensuring better compatibility as coding skills grow.

Note

In network automation, **None** often appears (without quotes) when a device does not provide certain data. This can happen if an interface is down, a configuration field is unset, or the API response omits a value. Checking for **None** ensures that missing or uninitialized data is handled properly, preventing errors in further processing.

Looping Through Key/Value Pairs with `.items()` and Processing Values with `isinstance()`

In this section, we bring together everything we have discussed so far in this chapter. The dictionaries in our examples—specifically those returned by the **`get_facts()`** method (and later the **`get_environment()`** method—are likely to contain various data types. These dictionaries are often *heterogeneous* (that is, containing different data types) and form the main dictionary, as discussed in [Chapter 6](#). Later in this chapter, we will examine how to iterate through other types of dictionaries as well.

When using a **`for`** loop with the **`.items()`** method to iterate through a dictionary, you may want to process both the key and its associated value. If the value is a simple data type—such as a string, integer, float, or Boolean—you can print it directly by using the **`print()`** function. Here is an example:

```
for key, value in device_facts.items():  
    print(f'{key}: {value}')
```

This would result in simply printing the key followed by the value, as shown here:

```
'hostname': 'Router-R1',  
'vendor': 'Cisco',  
'uptime': 3900.0
```

While this approach works well for simple data types, some values—such as lists and nested dictionaries—typically require further iteration using another **`for`** loop.

Iterating through a NAPALM dictionary involves the following steps:

Step 1. Iterate through each key/value pair by using a **for** loop and the **.items()** method.

Step 2. Evaluate each key by using **if-elif** and the **isinstance()** function to determine the key's type (string, integer, float, list, or dictionary).

- If the key is of type string, integer, float, or Boolean, print the key/value pair or perform other processing.
- If the key is of type list or dictionary, perform additional iteration and processing (with a **for** loop).

When the Dictionary Value Is a List

As you saw at the beginning of this chapter, in [Example 7-1](#), you can use the **get_facts()** method to display device information. This program uses the following commands to represent the dictionary as a variable, **device_facts**, and displays the contents of this variable by using **pprint()**:

```
device_facts = device.get_facts()
pprint(device_facts)
```

The output from [Example 7-3](#) is displayed again in [Example 7-19](#), as a reminder of the key/value pairs associated with the **get_facts()** method.

Example 7-19 Output from [Examples 7-1](#) and [7-2](#)

```
{'fqdn': 'Router-R1.SSH-KEY.com',
 'hostname': 'Router-R1',
 'interface_list': ['GigabitEthernet0/0/0',
                    'GigabitEthernet0/0/1',
                    'GigabitEthernet0/0/2',
```

```
        'GigabitEthernet0'],  
'model': 'ISR4331/K9',  
'os_version': 'ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),  
               '16.6.3, RELEASE SOFTWARE (fc8)',  
'serial_number': 'FD02302A08A',  
'uptime': 3900.0,  
'vendor': 'Cisco'}
```

[Example 7-20](#) shows how you can modify the original program in [Example 7-1 \(ex7-1_get_facts.py\)](#) to retrieve device facts from a network device using NAPALM and process the output using a **for** loop. The program connects to a Cisco router, collects system details using the same **get_facts()** method, and then iterates through the resulting dictionary. The key/value pairs are displayed in a structured format, with special handling for lists, such as **interface_list**, which contains multiple network interfaces. Instead of printing the entire list as a single line, the program prints each interface on a separate line for better readability.

Note

Later in this chapter you will continue to modify this part of the program to include dictionaries and strings.

Example 7-20 *ex7-20_for_loop_dictionary_list.py*

```
import napalm  
from pprint import pprint  
  
driver = napalm.get_network_driver('ios')  
  
device = driver(  

```

```

        hostname='192.168.1.1',
        username='admin',
        password='cisco',
        optional_args={'secret':'spot'}
    )

device.open()

device_facts = device.get_facts()

### **Manually Traversing get_facts() Dictionary**
print('\nKey-Value pairs in the get_facts() dictionary:')

for key, value in device_facts.items():

    # Check if value is a list (e.g., interface_list)
    if isinstance(value, list):
        print(f'{key}:')
        for item in value:
            print(f'    - {item}')
    else:
        # Just print value if string, integer, float, or Boolean
        print(f'{key}: {value}')

device.close()

```

The output of this program, shown in [Example 7-21](#), presents the retrieved device information in an easy-to-read format. It displays standard attributes such as hostname, model, serial number, and uptime. **interface_list** is printed in a structured format, ensuring that each interface is listed individually. This method makes it easier to interpret network device data in an automation workflow.

Example 7-21 Output from [Example 7-20](#)

```
MyPrompt% python3 ex7-20_for_loop_dictionary_list.py

Key-Value pairs in the get_facts() dictionary:
fqdn: Router-R1.SSH-KEY.com
hostname: Router-R1
interface_list:
    - GigabitEthernet0/0/0
    - GigabitEthernet0/0/1
    - GigabitEthernet0/0/2
    - GigabitEthernet0
model: ISR4331/K9
os_version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Ver:
RELEASE SOFTWARE (fc8)
serial_number: FLM2229W1R6
uptime: 2820.0
vendor: Cisco
MyPrompt%
```

A Step-by-Step Explanation

Let's examine this process step by step.

Step 1: Iterate Through Key/Value Pairs

[Example 7-20](#) retrieves device information by using the NAPALM **get_facts()** method, which returns a dictionary containing various details about the network device. Because dictionaries store data as key/value pairs, we use a **for** loop with the **.items()** method to iterate through the contents of **device_facts** and process each key/value pair:

```
for key, value in device_facts.items():
```

This loop extracts each key (such as **'hostname'** or **'interface_list'**) and its associated value, which could be a string, a number, a list, or another dictionary. The goal is to properly format and display these values in a readable way.

Step 2: Handle Lists

One of the keys that **get_facts()** returns is **'interface_list'**, which contains a value that is a *list* of network interfaces. If you print this value directly, Python displays it as a list inside square brackets, like this: **['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', ...]**.

Instead, you can format this output so that each interface appears on a separate line. To do this, you check whether the value is a list by using the **isinstance()** function:

```
if isinstance(value, list):  
    print(f'{key}:')  
    for item in value:  
        print(f'    - {item}')
```

If the value is a list, the program first prints the key (for example, **'interface_list:'**), and then it iterates through each item in the list by using a nested **for** loop. For example, if the list contains **['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1', 'GigabitEthernet0/0/2', 'GigabitEthernet0']**, each item in the list represents an interface name, which is printed on a new line, prefixed by a dash (-) to improve readability.

For example, given the following dictionary entry:

```
'interface_list': ['GigabitEthernet0/0/0', 'GigabitEthernet0/0/1',  
'GigabitEthernet0/0/2', 'GigabitEthernet0']
```

the output of this section would be:

```
interface_list:
    - GigabitEthernet0/0/0
    - GigabitEthernet0/0/1
    - GigabitEthernet0/0/2
    - GigabitEthernet0
```

This formatting makes the output easier to read compared to printing the entire list on a single line.

Step 3: Handle Other Data Types

For each of the other keys in **device_facts**, the value is a string or a float (number). These values do not require additional iteration, so they can be printed directly:

```
else:
    print(f'{key}: {value}')
```

For example, given the following dictionary entries:

```
'hostname': 'Router-R1',
'vendor': 'Cisco',
'uptime': 3900.0
```

the loop produces:

```
hostname: Router-R1
vendor: Cisco
uptime: 3900.0
```

Each key/value pair is printed on a single line, making this output easy to interpret.

Summary of Loop Behavior

The **for** loop in [Example 7-20](#) ensures that every key/value pair in the dictionary is processed correctly:

- If the value is a list, it is printed with each item on a separate line.
- If the value is *not* a list, it is printed as a simple key/value pair.

When the Dictionary Value Is a Another Dictionary

When the dictionary value is another dictionary, the process of iterating through the data becomes a little more involved—but it's not difficult!

This section walks through the process carefully, breaking down the code step by step so you can see exactly what is happening at each level. By the end of this section, you'll be able to recognize and understand nested dictionaries.

Note

A nested dictionary is a dictionary where one or more of the values is itself another dictionary, creating multiple levels of key/value pairs.

[Example 7-22](#) uses the **get_environment()** method instead of **get_facts()**. This method returns structured data about system resources, such as CPU usage, memory, power status, and temperature readings. The returned output, stored in the variable **device_environment**, consists of nested dictionaries, which demonstrate how to traverse this type of data structure.

Note

For both those with Python experience and beginners, it's worth noting that you can write a single recursive function that handles dictionaries of *any* depth, not just two levels. This approach allows you to dynamically process arbitrarily nested structures without hardcoding the number of loops. [Appendix F, “Using a Recursive Function to Handle Nested Dictionaries of Any Depth,”](#) provides an example of this code.

Example 7-22 `ex7-22_for_loop_nested_dictionary.py`

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)

device.open()

device_environment = device.get_environment()

print('\nOutput: pprint(device_environment)')
pprint(device_environment)

### **Manually Traversing dictionary with nested dictionaries and lists
print('\nKey-Value pairs with nested dictionaries and lists:')
```

```

# For loop - first level key and first level value pairs
for first_level_key, first_level_value in device_environment.items():
    # Print first-level key
    print('\nFirst-level key:')
    print(f'{first_level_key}:')

    # Check if value is a list (e.g., interface_list)
    if isinstance(first_level_value, list):
        for item in first_level_value:
            print(f'    - {item}')

    # Nested 2 levels (one for loop for each level)
    # If value is another dictionary, go deeper
    elif isinstance(first_level_value, dict):

        # For loop - second level key and second level value pairs
        for second_level_key, second_level_value in first_level_value.items():
            # Print second-level key
            print('\n    Second-level key:')
            print(f'        {second_level_key}:')

            # If this is also a dictionary, go one more level
            if isinstance(second_level_value, dict):

                # For loop - third level key and third level value pairs
                for third_level_key, third_level_value in \
                    second_level_value.items():
                    # Print third-level key and third-level value
                    print('\n        Third-level key and value:')
                    print(f'            {third_level_key}: {third_level_value}')
            else:
                # Second level Value is not a list or dictionary
                print('\n        Second-level Value:')

```

```

        print(f'        {second_level_value}')
    else:

        # Print single value (assuming not a list or dictionary)
        # If first-level value is not a list or dictionary, print
        print(f'    {first_level_value}')

device.close()

```

[Example 7-23](#) shows the results of running the code in [Example 7-22](#). First, it displays the complete dictionary returned by the **get_environment()** method using **pprint()**, and then it shows the manually formatted traversal through the dictionary.

The output may seem detailed at first glance, but don't worry: We will break it down step by step in the next sections to help you clearly understand how the code generates this structure.

Example 7-23 *Output from [Example 7-22](#)*

```

MyPrompt % python3 ex7-22_for_loop_nested_dictionary.py

Output: pprint(device_environment)
{'cpu': {0: {'%usage': 1.0}},
 'fans': {'invalid': {'status': True}},
 'memory': {'available_ram': 1845474744, 'used_ram': 345593948},
 'power': {'invalid': {'capacity': -1.0, 'output': -1.0, 'status': -1.0}},
 'temperature': {'invalid': {'is_alert': False,
                             'is_critical': False,
                             'temperature': -1.0}}}

Key-Value pairs with nested dictionaries and lists:

```

First-level key:

cpu:

Second-level key:

0:

Third-level key and value:

%usage: 1.0

First-level key:

memory:

Second-level key:

used_ram:

Second-level Value:

345593948

Second-level key:

available_ram:

Second-level Value:

1845474744

First-level key:

temperature:

Second-level key:

invalid:

Third-level key and value:

is_alert: False

Third-level key and value:
is_critical: False

Third-level key and value:
temperature: -1.0

First-level key:
power:

Second-level key:
invalid:

Third-level key and value:
status: True

Third-level key and value:
output: -1.0

Third-level key and value:
capacity: -1.0

First-level key:
fans:

Second-level key:
invalid:

Third-level key and value:
status: True

MyPrompt%

Note

The **pprint()** function displays the key/value pairs in alphabetical order (by key), whereas NAPALM displays them in the order in which they are stored.

The output reveals that several first-level keys ('**cpu**', '**fans**', '**memory**', '**power**', and '**temperature**') each contain a least one nested dictionary.

Working with a Three-Level Nested Dictionary

When working with a three-level nested dictionary (such as **device_environment**), you follow these structured steps:

Step 1. Iterate through the first-level dictionary by using **.items()** to retrieve each *first-level* key and its associated first-level value.

Step 2. Print the first-level key.

Step 3. Check whether the first-level value is a list (for example, a list of interfaces).

- If it is, iterate through the list and print each item.

Step 4. Check whether the first-level value is a dictionary (a nested structure).

- If it is, iterate through the second-level keys and second-level values.

Step 5. Check whether the second-level value is also a dictionary.

- If it is, iterate more deeply through the third-level keys and third-level values, printing each one.
- If it is not, print the second-level value directly.

Step 6. If the first-level value is neither a list nor a dictionary, print the first-level value directly.

This process is shown in [Figure 7-2](#), which illustrates the logic used to process and print values from a three-level nested dictionary structure, such as those returned by NAPALM's **get_environment()** method.

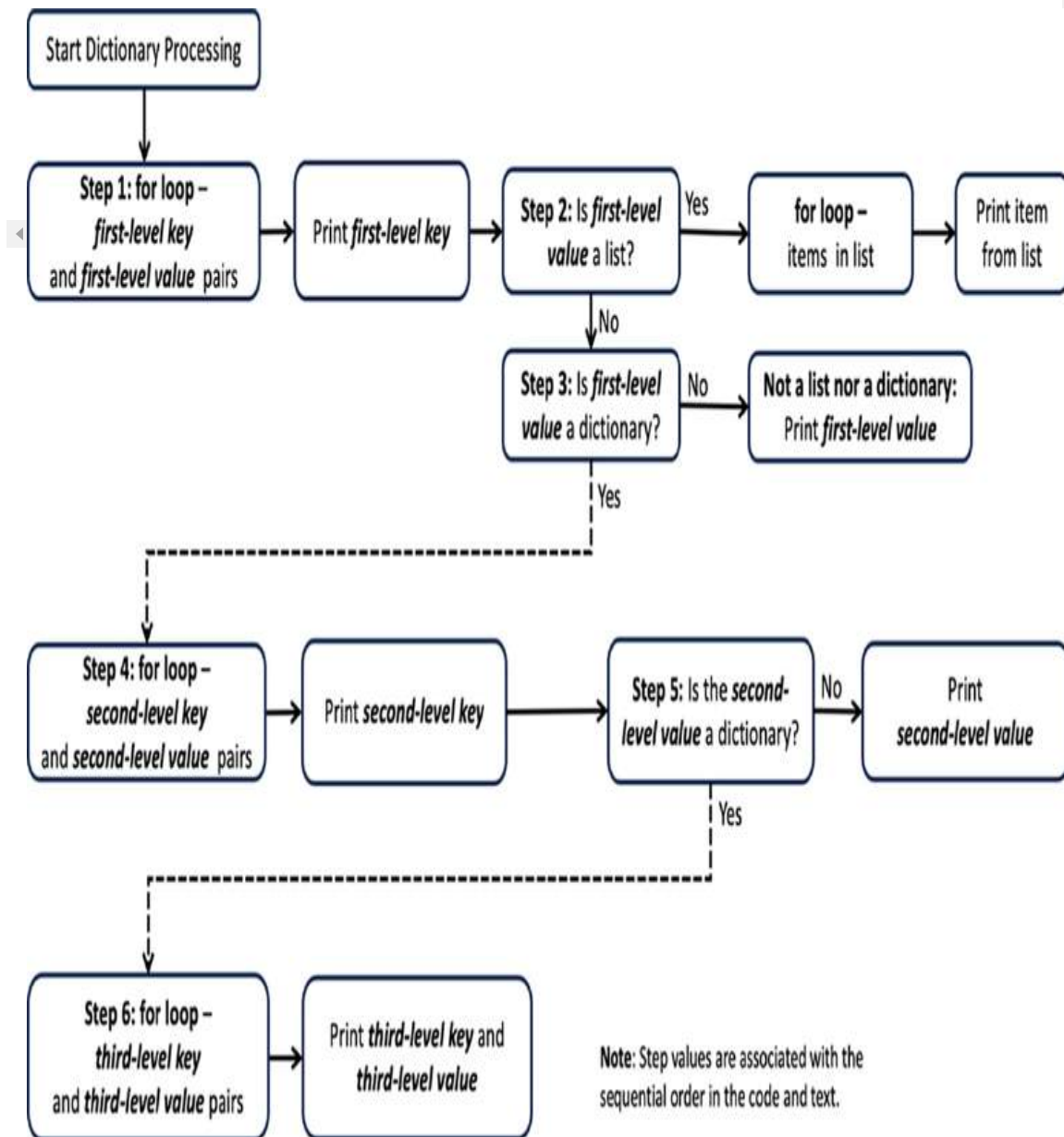


Figure 7-2 Process logic for a three-level dictionary using a series of **if-elif** statements

Step-by-Step Explanation: Three-Level Nested Dictionary

[Example 7-24](#) shows partial output from running the code in [Example 7-22](#). Let's walk through the process step by step, using just two of the first-level keys, **memory** and **temperature**.

Example 7-24 Selected Nested Dictionary Output

```
First-level key:
```

```
memory:
```

```
    Second-level key:
```

```
    used_ram:
```

```
        Second-level Value:
```

```
        345593948
```

```
    Second-level key:
```

```
    available_ram:
```

```
        Second-level Value:
```

```
        1845474744
```

```
First-level key:
```

```
temperature:
```

```
    Second-level key:
```

```
    invalid:
```

```
        Third-level key and value:
```

```
        is_alert: False
```



```
Third-level key and value:  
is_critical: False
```

```
Third-level key and value:  
temperature: -1.0
```

Step 1: Iterate Through the First-Level Keys and First-Level Values

You begin by using a **for** loop with **.items()** to retrieve each first-level key and first-level value from the dictionary:

```
for first_level_key, first_level_value in device_environment.items:  
    print('\nFirst-level key:')  
    print(f'{first_level_key}:')
```

At this point, you have the first-level key (such as **'memory'** or **'temperature'**) and its associated first-level value.

It is important to note that the first-level key is examined and iterates through all the nested dictionaries before the second first-level key is processed.

Print the First-Level Key

Immediately after retrieving the key and value, you print the first-level key:

```
for first_level_key, first_level_value in device_environment.items:  
    print('\nFirst-level key:')  
    print(f'{first_level_key}:')
```

This allows you to clearly separate and label each section of data.

Step 2: Check Whether the First-Level Value Is a List

After printing the first-level key, the next step is to check if the first-level value is a list:

```
if isinstance(first_level_value, list):  
    for item in first_level_value:  
        print(f'    - {item}')
```

If the first-level value is a list (such as a list of interfaces), you iterate through each item in the list and print it.

As you can see in [Example 7-24](#), with our examples (**memory** and **temperature**), the first-level values are *not* lists, so we don't iterate through the list and print each item.

Step 3: Check Whether the First-Level Value Is a Dictionary

If the first-level value is *not* a list, we check whether it is a dictionary:

```
elif isinstance(first_level_value, dict):
```

Step 4: Iterate Through Second-Level Keys and Second-Level Values

If the first-level value is a dictionary, you iterate through the second-level keys and second-level values and print the second-level keys. As you can see in [Example 7-24](#), the values of both **memory** and **temperature** are dictionaries:

```
    for second_level_key, second_level_value in first_level_value.  
        print('\n        Second-level key:')
```

```
print(f'    {second_level_key}:')
```

For '**memory**', the first-level value is a dictionary with two second-level keys:

- '**used_ram**'
- '**available_ram**'

For '**temperature**', the first-level value is a dictionary with one second-level key:

- '**invalid**'

Step 5: Check Whether the Second-Level Value Is Also a Dictionary

While iterating through the second-level key/value pairs, you check whether the second-level value is itself a dictionary:

```
if isinstance(second_level_value, dict):
    for third_level_key, third_level_value in second_level_value.items():
        print('\n        Third-level key and value:')
        print(f'            {third_level_key}: {third_level_value}')
else:
    print('\n        Second-level Value:')
    print(f'            {second_level_value}')
```

If the second-level value is a dictionary, you perform another iteration through third-level keys and third-level values, printing each:

```
if isinstance(second_level_value, dict):
    <code above omitted>
```

If the second-level value is a simple type (integer, string, and so on), you print the second-level value directly:

```
if isinstance(second_level_value, dict):
    <code above omitted>
else:
    print('\n          Second-level Value:')
    print(f'          {second_level_value}')
```

In [Example 7-24](#), notice that **memory** has two second-level keys, **available_ram** and **used_ram**.

The second-level values of both of these second-level keys are simple integers. Therefore, you print the second-level values directly.

Step 6: Iterate Through Third-Level Keys and Third-Level Values

In [Example 7-24](#), notice that **temperature** has a second-level key, **invalid**. The second-level key **invalid** contains a second-level value that is another dictionary. This dictionary contains three third-level keys, **is_alert**, **is_critical**, and **temperature**.

The **for** loop iterates through these three third-level key/value pairs and prints each one:

```
for third_level_key, third_level_value in second_level_value.items():
    print('\n          Third-level key and value:')
    print(f'          {third_level_key}: {third_level_value}')
```

If the First-Level Value Is Neither a List Nor a Dictionary, Print the First-Level Value Directly

Finally, if the first-level value was neither a list nor a dictionary (for example, if the value is of type string, float, or Boolean), you print it directly:

```
else:  
    print(f'    {first_level_value}')
```

This ensures that if a first-level value is a simple type (such as an integer or a string), it will still be handled properly. Earlier examples don't show this because the first-level value is a dictionary.

Accessing Specific Values in a Nested Dictionary

So far, you've seen how to iterate through a dictionary dynamically, handling different levels of nesting. However, if you already know the structure of the dictionary and the exact keys you want to access, you can directly reference values by using their keys.

The syntax for accessing these values follows this format, where each key leads to the next level of the dictionary:

```
dictionary_variable['key1']['key2']['key3'],
```

This forms a sequence (or chain) of keys, drilling down into the nested structure to retrieve the desired value. For example, the statements in [Example 7-25](#) extract specific data from **device_environment**.

Example 7-25 *Accessing Specific Dictionary Values*

```
# Retrieves the entire 'cpu' dictionary
print(device_environment['cpu'])

# Retrieves the dictionary nested under key 0 inside 'cpu'
print(device_environment['cpu'][0])

# Retrieves the '%usage' value from the nested dictionary
print(device_environment['cpu'][0]['%usage'])

# Retrieves the entire 'temperature' dictionary
print(device_environment['temperature'])

# Retrieves the nested dictionary under 'invalid'
print(device_environment['temperature']['invalid'])

# Retrieves the 'is_alert' value
print(device_environment['temperature']['invalid']['is_alert'])

# Retrieves the 'is_critical' value
print(device_environment['temperature']['invalid']['is_critical'])

# Retrieves the 'temperature' value
print(device_environment['temperature']['invalid']['temperature'])
```

This method is useful when working with known, structured data, as it allows you to extract specific values without looping through the dictionary.

Iterating Through a Dictionary of Dictionaries

Now let's look at another type of dictionary: a dictionary of dictionaries. This section shows how to use the NAPALM method **get_interfaces()** to retrieve the device's interface, which structures the information as a dictionary of dictionaries. The good news is that you can use the same code you used in [Example 7-22](#) and simply change the method to **get_interfaces()**.

In earlier chapters, you saw that a dictionary of dictionaries is similar to a table in a spreadsheet or database. Each first-level key represents a row identifier (such as an interface name), and its associated value is another dictionary that contains key/value pairs that act as columns (such as interface attributes like **mac_address**, **speed**, and **is_enabled**). This structure makes it easy to store and access structured data where each entry follows the same format. Because each interface has the same set of attributes, this structure is known as a *homogeneous dictionary*.

[Example 7-26](#) shows the code to retrieve interface information from a device by using the **get_interfaces()** method and assign it to the **device_interfaces** dictionary. The only difference between this code and the code in [Example 7-22](#) is that [Example 7-26](#) uses a different method (**get_interfaces()**) and assigns the output to a different variable (**device_interfaces**). You might notice that even though the method has changed, the logic of how to iterate through the dictionary remains exactly the same.

Because this is a dictionary that contains nested dictionaries, you first use **pprint(device_interfaces)** to display its structure in a readable format. [Example 7-27](#) shows the output to help you understand the organization of the data before you iterate through it.

Example 7-26 *ex7-26_for_loop_dictionaries.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
```

```

        hostname='192.168.1.1',
        username='admin',
        password='cisco',
        optional_args={'secret':'spot'}
    )

device.open()

device_interfaces = device.get_interfaces()

print('\nOutput: pprint(device_interfaces)')
pprint(device_interfaces)

### **Manually Traversing dictionary with nested dictionaries and lists
print('\nKey-Value pairs with nested dictionaries and lists:')

# For loop - first level key and first level value pairs
for first_level_key, first_level_value in device_interfaces.items():
    # Print first-level key
    print('\nFirst-level key:')
    print(f'{first_level_key}:')

    # Check if value is a list (e.g., interface_list)
    if isinstance(first_level_value, list):
        for item in first_level_value:
            print(f'    - {item}')

    # Nested 2 levels (one for loop for each level)
    # If value is another dictionary, go deeper
    elif isinstance(first_level_value, dict):

        # For loop - second level key and second level value pairs
        for second_level_key, second_level_value in first_level_value.items():

```



```

        # Print second-level key
        print('\n    Second-level key:')
        print(f'        {second_level_key}:')

        # If this is also a dictionary, go one more level
        if isinstance(second_level_value, dict):

            # For loop - third level key and third level value
            for third_level_key, third_level_value in \
                second_level_value.items():
                # Print third-level key and third-level value
                print('\n        Third-level key and value:')
                print(f'            {third_level_key}: {third_level_value}')
            else:
                # Second level Value is not a list or dictionary
                print('\n        Second-level Value:')
                print(f'            {second_level_value}')
        else:

            # Print single value (assuming not a list or dictionary)
            # If first-level value is not a list or dictionary, print it
            print(f'    {first_level_value}')

device.close()

```

Example 7-27 shows the output generated from iterating through **device_interfaces**. First, it displays the complete dictionary returned by **get_interfaces()** using **pprint()**, followed by the manually formatted traversal through the dictionary.

Each interface name (such as GigabitEthernet0/0/0) is a top-level key, and its corresponding attributes (such as

description, **is_enabled**, **mac_address**, and **mtu**) are nested key/value pairs. The iteration process demonstrates how each attribute can be displayed in a structured and readable format.

Example 7-27 Output from [Example 7-26](#)

```
MyPrompt% python3 ex7-26_for_loop_dictionaries.py

Output: pprint(device_interfaces)

{'GigabitEthernet0': {'description': '',
                      'is_enabled': False,
                      'is_up': False,
                      'last_flapped': -1.0,
                      'mac_address': 'DC:F7:19:94:A8:BF',
                      'mtu': 1500,
                      'speed': 1000.0},
 'GigabitEthernet0/0/0': {'description': '',
                          'is_enabled': True,
                          'is_up': True,
                          'last_flapped': -1.0,
                          'mac_address': 'DC:F7:19:94:A8:30',
                          'mtu': 1500,
                          'speed': 100.0},
 'GigabitEthernet0/0/1': {'description': '',
                          'is_enabled': True,
                          'is_up': True,
                          'last_flapped': -1.0,
                          'mac_address': 'DC:F7:19:94:A8:31',
                          'mtu': 1500,
                          'speed': 1000.0},
 'GigabitEthernet0/0/2': {'description': '',
                          'is_enabled': False,
```

```
'is_up': False,  
'last_flapped': -1.0,  
'mac_address': 'DC:F7:19:94:A8:32',  
'mtu': 1500,  
'speed': 1000.0}}
```

First-level key:
GigabitEthernet0/0/0:

Second-level key:
is_enabled:

Second-level Value:
True

Second-level key:
is_up:

Second-level Value:
True

Second-level key:
description:

Second-level Value:

Second-level key:
mac_address:

Second-level Value:
DC:F7:19:94:A8:30

Second-level key:

```
last_flapped:

    Second-level Value:
    -1.0

    Second-level key:
    mtu:

        Second-level Value:
        1500

    Second-level key:
    speed:

        Second-level Value:
        100.0

First-level key:
GigabitEthernet0/0/1:

<output omitted for brevity>
```

The iteration process for this dictionary follows the same approach used earlier, with **get_environment()**, where certain values were nested dictionaries requiring additional loops to access their key/value pairs. However, there is a key difference:

- In **get_environment()**, nested dictionaries contain different structures, depending on the key. For example, **'cpu'** and **'temperature'** have different fields.
- In **get_interfaces()**, every nested dictionary has the same structure. This consistency allows you to reuse the

same iteration logic without needing extra checks for varying dictionary formats.

Because all nested dictionaries contain the same fields, the same code can be used. Because there are no lists or other types of data, two parts of the code will not actually be used for this dictionary type. Even though these parts of the code are not needed for this specific output, you can leave them in place to maintain a flexible structure that works across many types of NAPALM dictionaries

This is the code to iterate through a list that will not be used:

```
# Check if value is a list (e.g., interface_list)
if isinstance(first_level_value, list):
    for item in first_level_value:
        print(f'    - {item}')
```

This is the code to print a value that is not a list nor a dictionary that will also not be used:

```
else:
    # Print single value (assuming not a list or dictionary)
    # If first-level value is not a list or dictionary, print it d
    print(f'    {first_level_value}')
```

Although some of this code is not required, this more general-purpose program can be used to iterate through different types of dictionaries. This means the same code works across multiple structures, making it more flexible when dealing with various types of data.

Note

The output generated in [Example 7-27](#) provides a structured view of the interface data, but it is not in a format that is suited for quick analysis. You might prefer to see this information in a tabular format, similar to a spreadsheet table, where each row represents an interface, and each column corresponds to a specific attribute (key), such as **mac_address**, **mtu**, **speed**, and **is_up**. Although Python code for this type of program is beyond the scope of this book, we provide an example in [Appendix C, “Using Python Dictionaries as NAPALM Outputs.”](#)

At some point, you might want to access the value of a key directly. The syntax for accessing such values follows the same format described previously:

```
dictionary_variable['key1']['key2']['key3'],
```

In this case, **device_interfaces** is a dictionary where each key represents an interface (for example, **'GigabitEthernet0/0/0'**), and the value is another dictionary that contains attributes for that interface.

[Example 7-28](#) shows how to access dictionary values by using the top-level key **'GigabitEthernet0/0/0'** and its corresponding second-level keys, **'is_enabled'**, **'mac_address'**, **'mtu'**, and **'speed'**. This method allows you to directly access specific details without iterating through the entire structure.

Example 7-28 *Accessing Specific Dictionary Values*

```
print(device_interfaces['GigabitEthernet0/0/0']['is_enabled'])
print(device_interfaces['GigabitEthernet0/0/0']['mac_address'])
```

```
print(device_interfaces['GigabitEthernet0/0/0']['mtu'])
print(device_interfaces['GigabitEthernet0/0/0']['speed'])
```

Iterating Through a List of Dictionaries

As you saw in [Chapter 6](#), a list of dictionaries is similar to a table in a spreadsheet:

- Each dictionary in the list represents a row.
- Each key/value pair within a dictionary represents a column and its associated data.

In a list of dictionaries, the keys are identical for each dictionary (just like consistent column headers), and the values represent the data for each entry (row). This structure is commonly used to organize information like ARP tables, routing tables, and other lists of structured data retrieved from a device.

[Example 7-29](#) shows how NAPALM uses the **get_arp_table()** method to retrieve the ARP table information from a device—in this case, a Cisco router. [Example 7-30](#) shows the output of this code.

Example 7-29 *ex7-29_for_loop_list_dict.py*

```
import napalm
from pprint import pprint

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
```

```

        username='admin',
        password='cisco',
        optional_args={'secret':'spot'}
    )

device.open()

device_arp = device.get_arp_table()

print('\nOutput: pprint(device_arp)')
pprint(device_arp)

### **Manually Traversing list of dictionaries**
print('\nEntries in the ARP table:')

# For loop - each item in the list
for entry in device_arp:
    print('\nNew ARP Entry:')

    # Each entry is a dictionary
    for key, value in entry.items():
        print(f'    {key}: {value}')

device.close()

```

Example 7-29 uses two **for** loops to iterate through the ARP table data:

- The first **for** loop uses a list.
- Using the variable **entry**, the **for** loop iterates through each entry in the list **device_arp**.

- Each entry is a separate dictionary that contains the same key/value pairs about a specific ARP record ('**age**', '**interface**', '**ip**', '**mac**'). For example, the first item in the list **entry** is assigned the first value:

```
entry = {'age': -1.0,  
        'interface': 'GigabitEthernet0/0/0',  
        'ip': '192.168.1.1',  
        'mac': 'DC:F7:19:94:A8:30'},
```

- The code prints a label (**New ARP Entry:**) to separate the ARP entries in the output.
- The second **for** loop (inside the first one) uses a dictionary.
- Using **entry.items()**, this **for** loop iterates through each of the key/value pairs within the current dictionary.
- For each key ('**age**', '**interface**', '**ip**', and '**mac**') and its corresponding value, the code prints both the key and the value. For example, this is the **for** loop through the **.items()** method of the first entry:

```
key, value in entry.items() = age: -1.0  
key, value in entry.items() = interface: GigabitEthernet0/0/0  
key, value in entry.items() = ip: 192.168.1.10  
key, value in entry.items() = mac: DC:F7:19:94:A8:30
```

Note

Recall that the **pprint()** function displays the key/value pairs in alphabetical order (by key), whereas NAPALM displays them in the order they are stored.

Example 7-30 Output from [Example 7-29](#)

MyPrompt% **python3 ex7-29_for_loop_list_dict.py**

Output: pprint(device_arp)

```
[{'age': -1.0,  
  'interface': 'GigabitEthernet0/0/0',  
  'ip': '192.168.1.1',  
  'mac': 'DC:F7:19:94:A8:30'},  
{ 'age': 1.0,  
  'interface': 'GigabitEthernet0/0/0',  
  'ip': '192.168.1.10',  
  'mac': '00:E0:4C:68:05:B6'},  
{ 'age': -1.0,  
  'interface': 'GigabitEthernet0/0/1',  
  'ip': '192.168.2.1',  
  'mac': 'DC:F7:19:94:A8:31'},  
{ 'age': 47.0,  
  'interface': 'GigabitEthernet0/0/1',  
  'ip': '192.168.2.2',  
  'mac': '04:62:73:49:4B:E1'}]
```

Entries in the ARP table:

New ARP Entry:

```
interface: GigabitEthernet0/0/0  
mac: DC:F7:19:94:A8:30  
ip: 192.168.1.1  
age: -1.0
```

New ARP Entry:

```
interface: GigabitEthernet0/0/0  
mac: 00:E0:4C:68:05:B6  
ip: 192.168.1.10  
age: -1.0
```

```
New ARP Entry:
```

```
interface: GigabitEthernet0/0/1  
mac: DC:F7:19:94:A8:31  
ip: 192.168.2.1  
age: -1.0
```

```
New ARP Entry:
```

```
interface: GigabitEthernet0/0/1  
mac: 04:62:73:49:4B:E1  
ip: 192.168.2.2  
age: 47.0
```

In this chapter, you have seen how to iterate through different types of structured data—dictionaries, nested dictionaries, and lists of dictionaries—and you can see how flexible Python’s **for** loops and the **isinstance()** function are when working with network device data.

Now that you have learned how to retrieve and navigate structured information from a device, you are ready to take the next step. [Chapter 8](#) moves beyond gathering data and begins using NAPALM to configure devices programmatically.

Summary

This chapter has built on your knowledge of Python dictionaries and applied it to the structured data returned by NAPALM.

You have learned how to use three important dictionary methods:

- **.keys():** This method returns all the keys in the dictionary (for example, **'hostname'**, **'uptime'**, **'vendor'**), allowing you to see what kinds of information are available.
- **.values():** This method returns all the values stored in the dictionary, which can be helpful when you're interested only in the data and not the attribute names.
- **.items():** This method returns both the keys and their corresponding values as key/value pairs (tuples), and it is the most useful method for looping through NAPALM dictionaries in a **for** loop.

In this chapter, you have practiced using these methods with **for** loops to iterate through different types of NAPALM dictionaries:

- A single heterogeneous dictionary like the one returned by **get_facts()**, which contains different data types, such as strings, floats, and lists
- A dictionary of dictionaries from **get_interfaces()**, where each key (such as an interface name) maps to another dictionary of attributes, such as MAC address, speed, and status
- A list of dictionaries, as in **get_arp_table()**, where each item in the list is a dictionary representing an ARP entry, with consistent keys like **'ip'** and **'mac'**

You have also learned how to use the **type()** and **isinstance()** functions to check the type of a dictionary value so that you can decide how to process or display it. This is especially important when handling lists (like **interface_list**) or nested dictionaries (like those returned by **get_environment()**), where extra loops may be needed to extract all the details.

In this chapter you have also seen how to format structured data in a more readable way, such as by printing list items line by line or thinking about how tabular views (such as spreadsheets) can make data easier to understand. (The code for the tabular format is provided in [Appendix G](#). “[Using Tabular Output](#).”)

By understanding how to iterate through and interpret these dictionary structures, you now have a strong foundation for retrieving and working with structured data from network devices using NAPALM. This is a key step toward building more powerful automation programs that can analyze, report on, and configure your network.

Chapter 8. Configuring Devices with NAPALM

In this chapter, we will explore how to safely and programmatically configure Cisco devices using NAPALM.

With manual CLI configuration, changes are applied immediately. In contrast, NAPALM provides a structured and flexible workflow that allows administrators to stage, preview, apply, or cancel configuration changes. Although NAPALM manages the workflow, the actual configuration still uses the device's native operating system commands, such as those used in Cisco IOS.

Whether you are modifying the current running configuration or replacing it entirely, NAPALM does not apply changes immediately. Instead, it holds changes in a candidate configuration until they are explicitly committed. This process—supported by built-in comparison tools and automatic backups—makes it easier to automate changes while minimizing the risk of disruption.

You can use NAPALM methods such as **load_merge_candidate()**, **load_replace_candidate()**, **compare_config()**,

commit_config(), **discard_config()**, and **rollback()** to manage device configuration in a safe, structured, and repeatable way (see [Figure 8-1](#)).

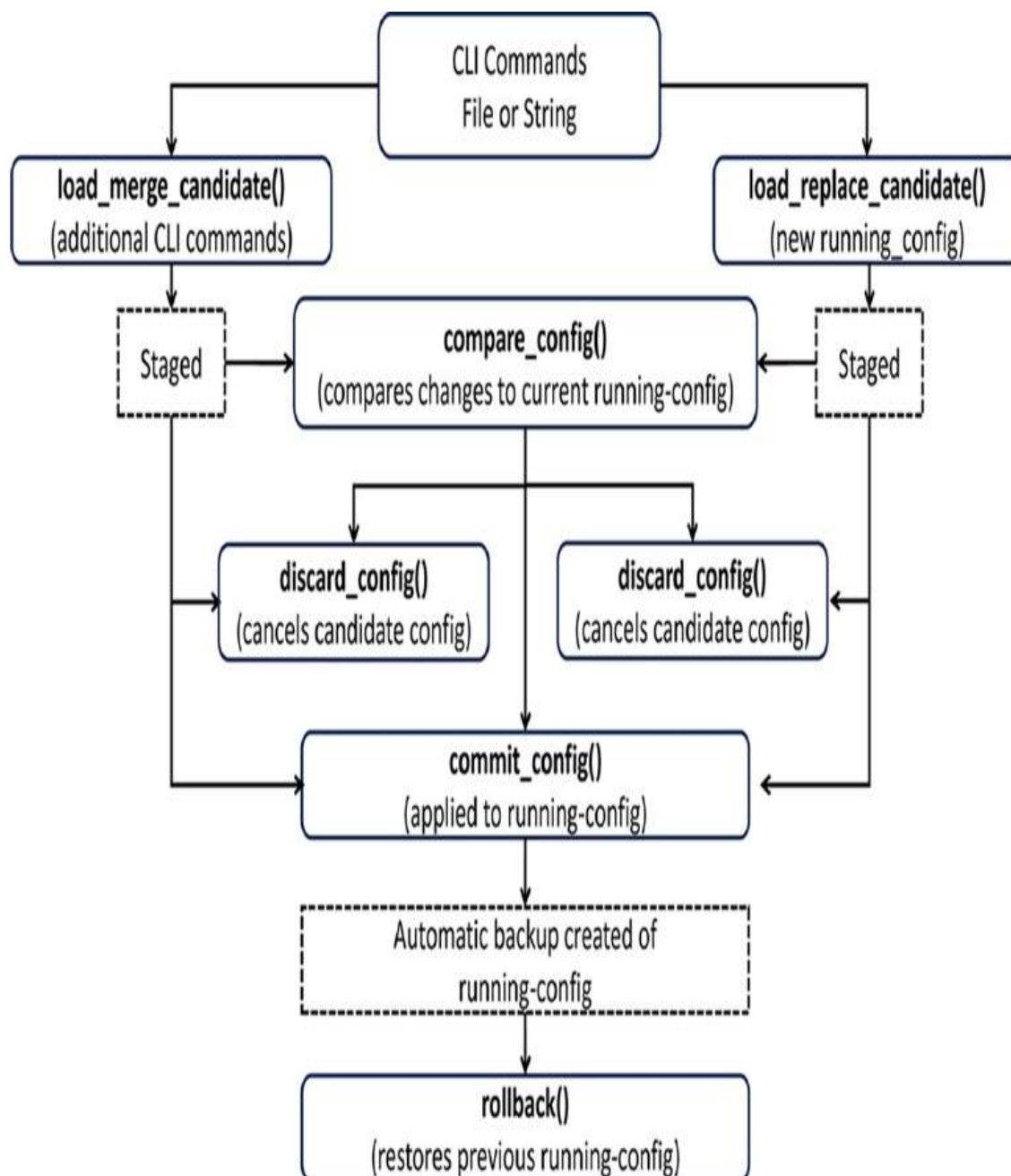


Figure 8-1 NAPALM configuration workflow

A Quick Overview of NAPALM Methods

Let's briefly talk about the methods shown in [Figure 8-1](#). Understanding these methods will help you see how NAPALM safely stages, applies, and manages configuration changes on a network device. Each method plays a specific role in the configuration workflow, from loading and previewing changes to committing or rolling back if needed.

Here is a brief explanation of the methods shown in [Figure 8-1](#):

- **load_merge_candidate(config=None):** Loads configuration commands that will merge with the existing running configuration. This is similar to entering CLI commands manually without removing existing settings.
- **load_replace_candidate(config=None):** Loads an entire configuration that is intended to completely replace the current running configuration. This is like copying a full configuration file onto the device.
- **compare_config():** Compares the currently staged candidate configuration to the device's existing running configuration and displays the differences.
- **discard_config():** Cancels the candidate configuration and discards any changes that were staged but not yet committed.
- **commit_config():** Applies the candidate configuration changes to the running configuration. Until this method is called, no changes are actually made to the device.
- **rollback():** Reverts the running configuration to the backup copy that was automatically created before the

last commit. This provides a quick way to undo changes if needed.

Introducing Our Example Scenario

In the sections that follow, we will walk through a NAPALM configuration workflow using a simple but practical example: adding IPv6 addressing to a Cisco router. Using the topology in [Figure 8-2](#), we will use SSH to connect to the device at 192.168.1.1. Once connected, we will use NAPALM to load and stage IPv6 interface configuration commands, preview the changes, and then either commit or discard them as needed.

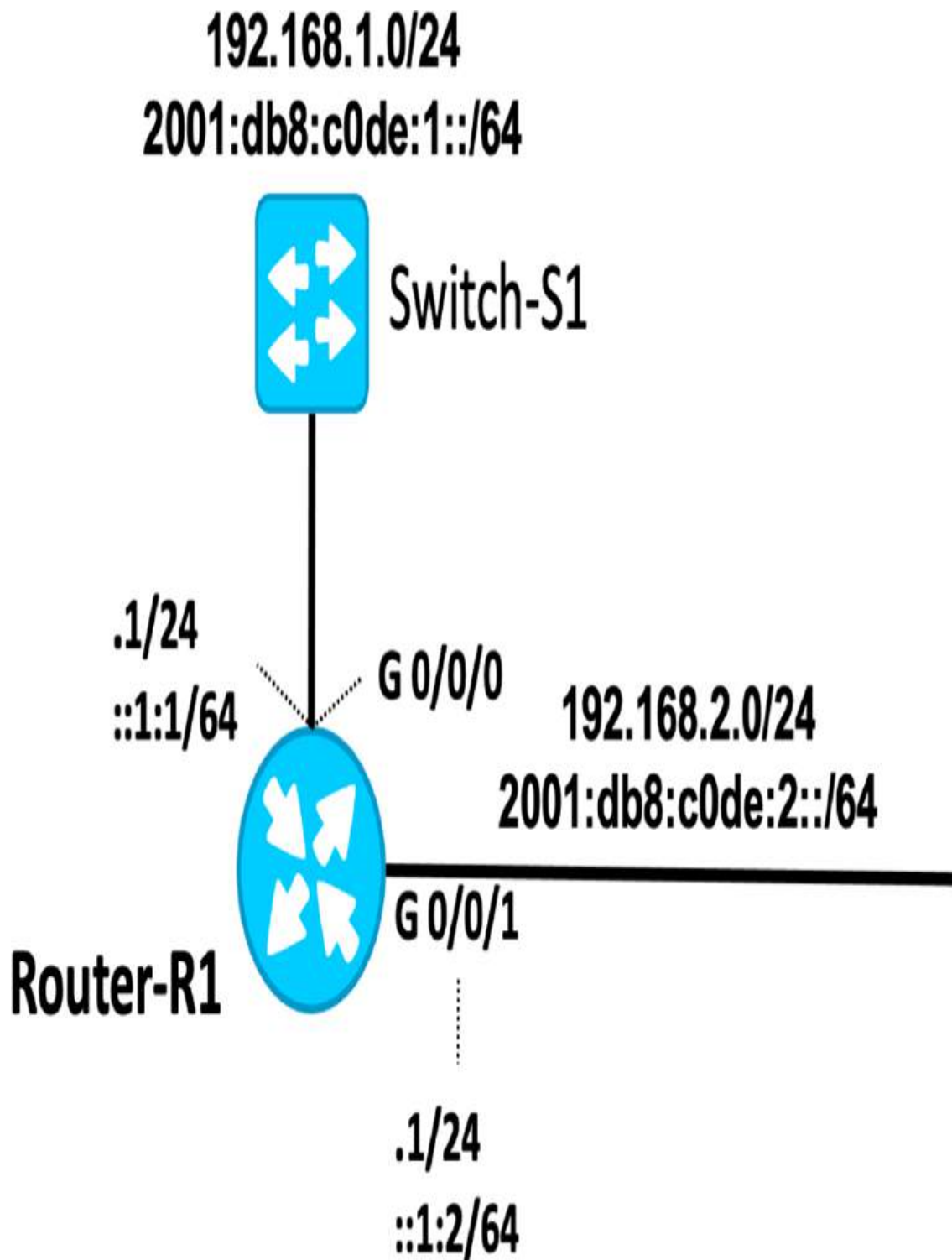


Figure 8-2 *Router-R1*

In this example, you will learn how to safely introduce new configuration lines—such as IPv6 address commands—using

the merge method, which gives you full control over the changes before they are applied to the running configuration.

In order to use NAPALM's configuration methods on a device, the device must allow file transfers over Secure Copy Protocol (SCP) and provide an authorized user account with sufficient privileges.

Two IOS configuration commands are required before you can use NAPALM configuration methods:

```
Router-R1(config)# ip scp server enable  
Router-R1(config)# username admin privilege 15 password 0 cisco
```

The **ip scp server enable** command enables the SCP server functionality on the router or switch. NAPALM often uses SCP (or sometimes SFTP) to **upload configuration files** to the device.

The **username admin privilege 15 password 0 cisco** command creates a user account named **admin** with privilege level **15** (the highest administrative level) and password **cisco**. NAPALM needs a fully privileged user because it performs configuration operations that require access to global configuration mode (configure terminal) and possibly to privileged EXEC mode (enable).

NAPALM's configuration methods are designed to stage, compare, and safely apply configuration changes. NAPALM often uploads temporary configuration files to devices using SCP or SFTP. If you don't enable the SCP server and provide a full administrative user account, NAPALM cannot transfer configurations or enter configuration mode, and operations like **load_merge_candidate()** will fail.

Configuring a Device with `load_merge_candidate()`

[Example 8-1](#) uses the `load_merge_candidate()` method to safely configure IPv6 addressing on two router interfaces: **GigabitEthernet0/0/0** and **GigabitEthernet0/0/1**. Instead of applying changes immediately, the program stages the configuration commands, displays the proposed changes using `compare_config()`, and prompts the user to either commit or discard the changes. This staged approach allows administrators to verify changes before making them active on a device.

Example 8-1 *ex8-1_configure_ipv6.py*

```
import napalm
from pprint import pprint

# Load Cisco IOS driver
driver = napalm.get_network_driver('ios')

# Define device
device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret': 'spot'}
)

# Open connection to device
device.open()

# View interface configuration before changes
print('\nBefore changes:')
```

```
pprint(device.get_interfaces_ip())

# Define IPv6 config as list of IOS CLI commands
ipv6_config = '''

ipv6 unicast-routing

interface GigabitEthernet0/0/0
ipv6 address 2001:db8:c0de:1::1/64
ipv6 address fe80::1 link-local
exit

interface GigabitEthernet0/0/1
ipv6 address 2001:db8:c0de:2::1/64
ipv6 address fe80::1 link-local
exit
'''

# Load configuration into candidate
device.load_merge_candidate(config=ipv6_config)

# Show the differences between running config and candidate config
print('\nStaged changes:')
print(device.compare_config())

# Ask user if they want to commit the changes
user_input = input("\nDo you want to commit these changes? (yes, no) ")

if user_input.lower() == 'yes':
    device.commit_config()
    print('\nChanges committed.')
else:
    device.discard_config()
```

```
print('\nChanges discarded.')

# Verify interface configuration after changes
print('\nAfter changes:')
pprint(device.get_interfaces_ip())

# Close connection
device.close()
```

[Example 8-2](#) shows the IP addressing before the configuration changes, the staged IPv6 addressing changes using **compare_config()**, and the final result after committing the changes.

Example 8-2 *Output from [Example 8-1](#)*

```
MyPrompt% python3 ex8-1_configure_ipv6.py

Before changes:
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length': 24}},
  'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length': 24}}}}

Staged changes:
+ipv6 unicast-routing
+
+interface g0/0/1
+ipv6 address 2001:db8:c0de:1::1/64
+ipv6 address fe80::1 link-local
+exit
+
+interface g0/0/0
+ipv6 address 2001:db8:c0de:2::1/64
+ipv6 address fe80::1 link-local
```

```
+exit
```

```
Do you want to commit these changes? (yes/no): yes
```

```
Changes committed.
```

```
After changes:
```

```
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length':  
                                'ipv6': {'2001:db8:c0de:1::1': {'prefix_length':  
                                'fe80::1': {'prefix_length':  
'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length':  
                                'ipv6': {'2001:db8:c0de:2::1': {'prefix_length':  
                                'fe80::1': {'prefix_length':
```

```
MyPrompt%
```

Note

Your **compare_config()** diff (short for *difference*, a common term in Linux and programming in general) process may include or omit context lines like **interface g0/0/1**, and interface names may appear abbreviated (for example, **g0/0/1**). Output varies by IOS/IOS XE version and device. The presence or absence of these lines does not change what will be committed.

After establishing the connection to the router using NAPALM, you first retrieve and display the current IP addressing information on the device by using **get_interfaces_ip()**. This allows you to verify the interface status before making any changes. Notice that there is currently no IPv6 addressing information on the interfaces.

Next, you define the new IPv6 addressing configuration in a multiline string stored in the **ipv6_config** variable. Using

triple quotes (""" """) makes it easy to format multiple CLI commands across several lines in a clear and organized way.

The new configuration is then staged using the **load_merge_candidate()** method. At this point, no changes have been applied to the running configuration. To preview the proposed changes, you use the **compare_config()** method, which displays the differences between the current running configuration and the staged candidate configuration.

The output of **compare_config()** displays the differences between the current running configuration and the candidate configuration that has been staged. Each line is marked with a symbol:

- A plus sign (+) means the line will be *added to* the running configuration if the changes are committed.
- A minus sign (-) indicates that a line will be *removed from* the running configuration if the changes are committed.

In [Example 8-2](#), all lines are marked with a plus sign (+), showing that new IPv6 address configurations are being added to the router interfaces.

Before proceeding, the program prompts the user to either commit or discard the staged changes. If the user enters **yes**, the **commit_config()** method is called to apply the changes to the running configuration. If the user enters **no**, the **discard_config()** method is called to cancel the changes, and the device remains unchanged.

Finally, you retrieve and display the updated interface addressing by using **get_interfaces_ip()** again. This time, you see that the IPv6 addressing has been configured and

applied to the running configuration. Finally, you close the connection to the device by using **device.close()**.

Understanding the Relationships Between Configuration Methods

Now that we've seen NAPALM methods in action, let's take a closer look at how they work together during the configuration workflow.

Changes loaded with **load_merge_candidate()** or **load_replace_candidate()** are staged in a temporary candidate configuration. The device's actual running configuration remains unchanged until **commit_config()** is executed.

commit_config() finalizes the candidate configuration. If you used **load_merge_candidate()**, the changes are merged into the running configuration. If you used **load_replace_candidate()**, the entire running configuration is replaced with the candidate.

NAPALM handles this very safely: It creates a backup of the current configuration before committing, so you can use **rollback()** if something goes wrong.

The **rollback()** method must be called *before* **device.close()** ends the session. After the connection is closed, the automatic backup created during **commit_config()** is no longer available, and a rollback cannot be performed.

It is important to note that the **commit_config()** method does not merge these same commands into the startup configuration file. Saving the running configuration file to the startup configuration file is covered in the next section.

Saving the Configuration

While NAPALM supports committing configuration changes to the running configuration using **commit_config()**, not all platforms support saving those changes to the startup configuration using **save_config()**. Where supported, the **save_config()** method performs the equivalent of **copy running-config startup-config**. Always check platform compatibility before relying on this feature.

When working with NAPALM methods like **save_config()**, there are two possible issues to prepare for:

- If the method *exists but is not supported*, NAPALM raises a **NotImplementedError** exception.
- If the method *does not exist at all* (which is common for some platforms, like Cisco IOS), Python raises an **AttributeError** exception.

To safely handle both cases, always wrap **save_config()** in a **try...except** block and catch **AttributeError**. On unsupported platforms, you can fall back to using the **cli()** method to manually issue the IOS command **copy running-config startup-config**.

[Example 8-3](#) extends our earlier program to include saving the running configuration to the startup configuration. It uses a **try...except** block to handle platforms where **save_config()** is not available. When this method is not available, the program falls back to manually issuing the **copy running-config startup-config** command using the **cli()** method.

Example 8-3 *Including the Commands to Save the Running Configuration to the Startup Configuration*

```
<previous code from ex8-1_configure_ipv6.py omitted for brevity>

# Verify interface configuration after changes
print('\nAfter changes:')
pprint(device.get_interfaces_ip())

# After committing changes, try to save the configuration (if supported)
print('\nSaving running-config to startup-config')
try:
    device.save_config()
    print('\nUsing NAPALM save.config()')
    print('\nConfiguration saved to startup-config.')
except AttributeError:
    # Some platforms (like Cisco IOS) do not support save_config()
    print('\nUsing IOS copy running-config startup-config')
    output = device.cli(['copy running-config startup-config'])
    print(output['copy running-config startup-config'])
    print('\nConfiguration saved to startup-config.')

# Close connection
device.close()
```

Note

In addition to being used for configuration management, NAPALM's **cli()** method can also be used to send operational commands to a device and retrieve their output in a structured format. In the next section, we will explore how to use the **cli()** method to run common **show** commands and work with the results in Python.

Example 8-4 shows the output that results from running the program with the changes we just discussed.

Example 8-4 Output from Example 8-3

```
MyPrompt% python3 ex8-3_configure_save.py

Before changes:
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length': 24}, 'ipv6': {'2001:db8:c0de:1::1': {'prefix_length': 64}, 'fe80::1': {'prefix_length': 64}}, 'name': 'GigabitEthernet0/0/0'}, {'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length': 24}, 'ipv6': {'2001:db8:c0de:2::1': {'prefix_length': 64}, 'fe80::1': {'prefix_length': 64}}, 'name': 'GigabitEthernet0/0/1'}}

Staged changes:
+ipv6 unicast-routing
+
+interface g0/0/1
+ipv6 address 2001:db8:c0de:1::1/64
+ipv6 address fe80::1 link-local
+exit
+
+interface g0/0/0
+ipv6 address 2001:db8:c0de:2::1/64
+ipv6 address fe80::1 link-local
+exit

Do you want to commit these changes? (yes/no): yes

Changes committed.

After changes:
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length': 24}, 'ipv6': {'2001:db8:c0de:2::1': {'prefix_length': 64}, 'fe80::1': {'prefix_length': 64}}, 'name': 'GigabitEthernet0/0/0'}, {'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length': 24}, 'ipv6': {'2001:db8:c0de:1::1': {'prefix_length': 64}, 'fe80::1': {'prefix_length': 64}}, 'name': 'GigabitEthernet0/0/1'}}}
```

```
        'ipv6': {'2001:db8:c0de:1::1': {'pref:
            'fe80::1': {'prefix_length':
'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_leng
        'ipv6': {'2001:db8:c0de:2::1': {'pref:
            'fe80::1': {'prefix_length':

Saving running-config to startup-config

Using IOS copy running-config startup-config
Building configuration...

[OK]

Configuration saved to startup-config.
MyPrompt%
```

Using the cli() method

While NAPALM provides structured methods for retrieving common device information, sometimes you need the flexibility to run specific CLI commands directly. The **cli()** method allows you to send operational commands—such as **show** commands—to a device and retrieve their output in a structured dictionary format. This is especially useful when a particular command is not covered by NAPALM’s predefined **get** methods or when you want to quickly inspect device status, interface information, or routing tables. In this section, you will learn how to use the **cli()** method to retrieve and work with device data in Python.

The program shown in [Example 8-5](#) uses the **cli()** method to send the **show ipv6 interface brief** command to the router. It demonstrates two ways of working with the output. First, it shows how to pretty-print the entire returned

dictionary, which is useful when you want to see "live" information (that is, current information at the time the command is run). Second, it shows how to display the output without the surrounding dictionary structure, which is especially useful when you want to store the command results to use later in the code. [Example 8-6](#) shows the resulting output.

Example 8-5 *ex8-5_cli_ipv6_brief.py*

```
import napalm
from pprint import pprint

# Load Cisco IOS driver
driver = napalm.get_network_driver('ios')

# Define device
device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret': 'spot'}
)

# Open connection to device
device.open()

# Send the "show ipv6 interface brief" command and pretty-print
dictionary output
print('\nOutput from: device.cli(["show ipv6 interface brief"])')
pprint(device.cli(['show ipv6 interface brief']))

# Send the same command again and store the dictionary output in
output_v6 = device.cli(['show ipv6 interface brief'])
```

```
# Print only the command output text (accessing the value inside
print('\nOutput from: output_v6["show ipv6 interface brief"]')
print(output_v6['show ipv6 interface brief'])

# Close connection
device.close()
```

Example 8-6 *Output from [Example 8-5](#)*

```
MyPrompt% python3 ex8-5_cli_method_ipv6_brief.py

Output from: device.cli(["show ipv6 interface brief"]
{'show ipv6 interface brief': 'GigabitEthernet0/0/0    [up/up]\n
                               '    FE80::1\n'
                               '    2001:DB8:C0DE:1::1\n'
                               'GigabitEthernet0/0/1    [down/down]
                               '    FE80::1\n'
                               '    2001:DB8:C0DE:2::1\n'
                               'GigabitEthernet0/0/2    [administrat
                               'down/down]\n'
                               '    unassigned\n'
                               'GigabitEthernet0          [administrat
                               'down/down]\n'
                               '    unassigned'}
```

```
Output from: output_v6["show ipv6 interface brief"]
GigabitEthernet0/0/0    [up/up]
    FE80::1
    2001:DB8:C0DE:1::1
GigabitEthernet0/0/1    [down/down]
    FE80::1
    2001:DB8:C0DE:2::1
```

```
GigabitEthernet0/0/2    [administratively down/down]
    unassigned
GigabitEthernet0        [administratively down/down]
    unassigned
MyPrompt%
```

Understanding the Code and Output

[Example 8-5](#) uses the `cli()` method to send the command **show ipv6 interface brief** to the device and retrieve the output. The following sections take a closer look at the three commands that are boldfaced in [Example 8-5](#):

```
pprint(device.cli(['show ipv6 interface brief'])),  
output_v6 = device.cli(['show ipv6 interface brief']),  
and print(output_v6['show ipv6 interface brief']).
```

Using the pprint Command

This line sends the **show ipv6 interface brief** command to the device and immediately pretty-prints the entire dictionary returned by the `cli()` method:

```
pprint(device.cli(['show ipv6 interface brief']))
```

The output is shown in dictionary format, where the key is the exact command string, and the value is the output text from the device. This line can be used for quick inspection or debugging and does *not* store the result for later use.

The key is the exact command string that was sent:

```
'show ipv6 interface brief':
```

The value is the full text output from the device after running that command:


```
'GigabitEthernet0/0/0    [up/up]\n'  
'    FE80::1\n'  
'    2001:DB8:C0DE:1::1\n'  
'GigabitEthernet0/0/1    [down/down]\n'  
'    FE80::1\n'  
'    2001:DB8:C0DE:2::1\n'  
'GigabitEthernet0/0/2    [administratively down/down]\n'  
'    unassigned\n'  
'GigabitEthernet0        [administratively down/down]\n'  
'    unassigned'
```

All the pieces inside single quotes (") are part of the same value. Python splits the long text output into multiple lines for readability when printing it, but in memory, it is still one long string. Each `\n` is a newline character that shows where a line break occurs in the device's output.

Storing the Output to a Variable

This line again sends the same command we just discussed, but instead of printing the output immediately, it stores the returned dictionary into a variable named **output_v6**:

```
output_v6 = device.cli(['show ipv6 interface brief'])
```

This allows you to work with the result later in the program.

Accessing the Value of the Key "show ip interface brief"

This line accesses just the value associated with the "**show ip interface brief**" key inside the **output_v6** dictionary:

```
print(output_v6['show ip interface brief'])
```

It prints the command output text retrieved from the device without displaying the entire dictionary structure.

The following output shows the value associated with the **'show ipv6 interface brief'** key in the **output_v6** dictionary:

```
GigabitEthernet0/0/0    [up/up]
    FE80::1
    2001:DB8:C0DE:1::1
GigabitEthernet0/0/1    [down/down]
    FE80::1
    2001:DB8:C0DE:2::1
GigabitEthernet0/0/2    [administratively down/down]
    unassigned
GigabitEthernet0        [administratively down/down]
    unassigned
```

Accessing and printing only the value from the CLI output keeps your program cleaner and makes it easier to further process or parse the device output later. This approach is especially useful when you want to work with the command results without the surrounding dictionary structure.

Accessing the Key “show ip interface brief”

You can also print a command if you wish. In this example, the command is known in advance, so it can be printed directly as a string rather than being retrieved from the dictionary.

```
print('show ipv6 interface brief')
```

However, if you want to programmatically print the actual command or commands that were sent and stored as keys inside the **output_v6** dictionary, you can use the **.keys()** method to retrieve them:

```
for command in output_v6.keys():  
    print(command)
```

The Read-Only cli() Method

The **cli()** method in NAPALM is intended for read-only operational commands such as **show ipv6 interface brief** and cannot be used to configure devices. Instead, configuration should be done using NAPALM's structured methods, like **load_merge_candidate()**.

In the previous section, you saw the **cli()** method used to manually issue the **copy running-config startup-config** command:

```
output = device.cli(["copy running-config startup-config"])  
print(output["copy running-config startup-config"])
```

The variable **output** is a dictionary where the key is "**copy running-config startup-config**" and the value is the output text returned from the device after executing the command, such as:

```
Building configuration...  
[OK]
```

When you use the **cli()** method, the key used to retrieve the output *must exactly match the command string* you sent, including any spaces and capitalization. Even a small mismatch will cause a **KeyError** exception.

For example, a space between **startup-config** and the closing **"**, as shown here, will cause an error:

```
print(output["copy running-config startup-config "]) # Extra space
```

Using `load_replace_candidate()` to Replace the Configuration

In addition to merging small configuration changes, NAPALM allows you to completely replace a device's running configuration by using the **`load_replace_candidate()`** method. This approach stages a full replacement configuration that, when committed, overwrites the existing running configuration with the new one.

However, it is important to use this method with caution and careful attention to detail. Any configuration commands not included in the replacement file, such as SSH access, IP addresses, and passwords, will be removed when the configuration is applied. This will result in the SSH connection being terminated, and you may need to use a console connection to reconnect to the device.

In this section, we will walk through how to safely stage, preview, and commit a full configuration replacement using NAPALM. The only change we will be making is to modify the hostname from **Router-R1** to **Router-R1-NEW**.

Cisco IOS Archive Feature

Before using **`load_replace_candidate()`** on Cisco IOS devices, it is necessary to configure the **archive** feature. NAPALM relies on Cisco's **archive config replace** mechanism to safely replace the running configuration with a staged candidate configuration. At a minimum, **archive** must be enabled with a valid path to save backup copies of the configuration, for example:

```
Router-R1(config)# archive
Router-R1(config-archive)# path flash:backup-config
```

If **archive** is not properly configured, the device will reject the configuration replacement attempt, and NAPALM will raise an error.

Note

The **write-memory** archive command can be included to automatically save the running configuration to the startup configuration any time changes are made to the running configuration. However, some newer IOS versions auto-save the running configuration to startup configuration after a successful replacement, even without **write-memory**.

Using a New Configuration File to Change the Hostname

[Example 8-7](#) shows the command used to SSH into the router to verify the current hostname.

Example 8-7 *Verifying the Hostname Router-R1*

```
MyPrompt% ssh admin@192.168.1.1
(admin@192.168.1.1) Password:

Router-R1#
Router-R1#
```

[Example 8-8](#) shows the code used to replace the running configuration file, which will modify the hostname.

Example 8-8 *ex8-8_load_replace_candidate.py*

```
import napalm
from pprint import pprint

# Load Cisco IOS driver
driver = napalm.get_network_driver('ios')

# Define device
device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret': 'spot'}
)

# Open connection to device
device.open()

# Define full replacement configuration as a multi-line string
full_replacement_config = ''
version 16.6
service timestamps debug datetime msec
service timestamps log datetime msec
platform qfp utilization monitor load 80
no platform punt-keepalive disable-kernel-core
!
hostname Router-R1-NEW
!
boot-start-marker
boot-end-marker
!
!
vrf definition Mgmt-intf
!
```

```
address-family ipv4
exit-address-family
!
address-family ipv6
exit-address-family
!
enable secret 5 $1$tGx0$PAdygny/8T5W2.xTctqlu0
!
no aaa new-model
!
no ip domain lookup
ip domain name SSH-KEY.com
!
subscriber templating
ipv6 unicast-routing
!
!
multilink bundle-name authenticated
!
license udi pid ISR4331/K9 sn FD022512ULU
file prompt quiet
diagnostic bootup level minimal
spanning-tree extend system-id
archive
  path flash:backup-config
!
username admin privilege 15 password 0 cisco
!
redundancy
  mode none
!
interface GigabitEthernet0/0/0
  description Updated LAN interface using Netmiko
```

```
ip address 192.168.1.1 255.255.255.0
negotiation auto
ipv6 address FE80::1 link-local
ipv6 address 2001:DB8:C0DE:1::1/64
!
interface GigabitEthernet0/0/1
description Updated interface to R2 using Netmiko
ip address 192.168.2.1 255.255.255.0
negotiation auto
ipv6 address FE80::1 link-local
ipv6 address 2001:DB8:C0DE:2::1/64
!
interface GigabitEthernet0/0/2
no ip address
shutdown
negotiation auto
!
interface GigabitEthernet0
vrf forwarding Mgmt-intf
no ip address
shutdown
negotiation auto
!
ip forward-protocol nd
ip http server
ip http authentication local
ip http secure-server
ip tftp source-interface GigabitEthernet0
ip route 0.0.0.0 0.0.0.0 192.168.2.2
!
ip scp server enable
!
control-plane
```



```
!  
line con 0  
    exec-timeout 0 0  
    logging synchronous  
    transport input none  
    stopbits 1  
line aux 0  
    stopbits 1  
line vty 0 4  
    login local  
    transport input ssh  
line vty 5 15  
    login local  
    transport input ssh  
!  
wsma agent exec  
!  
wsma agent config  
!  
wsma agent fileSYS  
!  
wsma agent notify  
!  
end  
...  
  
# Stage the replacement configuration  
device.load_replace_candidate(config=full_replacement_config)  
  
# Compare the staged configuration to the current running-config  
print('\nStaged differences:')  
print(device.compare_config())
```

```
# Prompt user to commit or discard
user_input = input('\nDo you want to commit this full configura-
(yes/no): ')

if user_input.lower() == 'yes':
    device.commit_config()
    print('\nNew configuration committed.')
else:
    device.discard_config()
    print('\nConfiguration changes discarded.')

# Close connection
device.close()
```

After staging a replacement configuration, it's important to understand that **full_replacement_config** must include everything from the device's current running configuration. To prepare a complete and valid replacement, it is best to retrieve the full configuration by using the **show running-config** command. While cosmetic comment lines (!) can be safely removed, all actual configuration commands—starting from the **version** line—must be preserved and properly ordered. Missing key elements, such as version information, service settings, or boot parameters, will cause the replacement to fail when you attempt to commit the configuration.

Note

You do not need to worry about the **crypto key generate rsa** command. This command is not part of the configuration file and cannot be included in a configuration file loaded by NAPALM. SSH keys must be generated manually at the device CLI before

using SSH access. Once the keys are created, they are saved in the device's memory and survive reloads unless they are manually deleted. In other words, the SSH keys (the RSA key pair) are not part of the running configuration or startup configuration. They are stored separately in the device's filesystem (usually in NVRAM or flash memory).

[Example 8-9](#) shows the output from running the code in [Example 8-8](#).

Example 8-9 *Output from [Example 8-8](#)*

```
MyPrompt% python3 ex8-8_load_replace_candidate.py

Staged differences:
+hostname Router-R1-NEW
-hostname Router-R1

Do you want to commit this full configuration replacement? (yes,

New configuration committed.
MyPrompt%
```

You can verify the change of the hostname by pressing Enter to get a new prompt and then reentering the command **ssh admin@192.168.1.1**, as shown in [Example 8-10](#).

Example 8-10 *Verifying the Hostname Router-R1-NEW*

```
MyPrompt % ssh admin@192.168.1.1
(admin@192.168.1.1) Password:
```

```
Router-R1#
Router-R1#
Router-R1-NEW#
Router-R1-NEW#
Router-R1-NEW#show startup-config
Using 2023 out of 33554432 bytes
!
! Last configuration change at 22:08:26 UTC Sun Apr 27 2025 by a
!
version 16.6
service timestamps debug datetime msec
service timestamps log datetime msec
platform qfp utilization monitor load 80
no platform punt-keepalive disable-kernel-core
!
hostname Router-R1-NEW
!
<output omitted for brevity>
```

[Example 8-10](#) also shows the **show startup-config** command being issued. Notice that **startup-config** is automatically updated with the new hostname as well. Why? When using **load_replace_candidate()** with Cisco IOS devices, NAPALM replaces the running configuration by leveraging the router's built-in **archive config replace** feature. This operation updates the running configuration in place but does not automatically save it to the startup configuration.

If the router's archive feature is configured with the **write-memory** option, it automatically creates a backup copy whenever the running configuration is manually saved (for example, using **copy running-config startup-config** or

NAPALM's **save_config()** method). The archive system itself never saves to NVRAM; after a replacement, the new configuration exists only in the running configuration until you explicitly save it. To ensure that the configuration persists across reloads, you can include a manual save or a call to **device.save_config()** in your program.

Displaying Running and Startup Configuration Files

In addition to staging and applying configuration changes, NAPALM also provides an easy way to retrieve and display the full contents of a device's running configuration and startup configuration. By using the **get_config()** method, you can access these configurations as Python dictionary values and print them for inspection. This allows you to quickly verify the current operational state of the device, check whether changes have been saved to the startup configuration, or simply review the device's configuration files directly from Python.

[Example 8-11](#) shows how to use the **get_config()** method to retrieve and display both the running configuration and startup configuration. The program includes a simple check to verify whether a startup configuration is available before attempting to display it, ensuring compatibility across different devices and situations.

Example 8-11 *ex8-11_get_config.py*

```
import napalm

# Load Cisco IOS driver
driver = napalm.get_network_driver('ios')
```

```
# Define device
device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret': 'spot'}
)

# Open connection to device
device.open()

configs = device.get_config()

# Display running-config
print("\nRunning Config:")
print(configs["running"])

# Display startup-config (if available)
if "startup" in configs:
    print("\nStartup Config:")
    print(configs["startup"])
else:
    print("\nStartup config not available on this platform.")

# Close connection
device.close()
```

The **get_config()** method retrieves the configuration files from the device and stores them in a Python dictionary. When you assign the output of **device.get_config()** to the variable **configs**, this dictionary contains different configuration sections as key/value pairs. The two most

common keys are "**running**" for the running configuration and "**startup**" for the startup configuration.

The command **print(configs["running"])** accesses and prints only the value associated with the "**running**" key, which is the full text of the device's current running configuration. Similarly, **print(configs["startup"])** accesses and prints the value associated with the "**startup**" key and displays the saved startup configuration, if it is available. This allows you to easily inspect both configurations directly from your Python program.

Next Step: Nornir

Now that you have seen how Netmiko and NAPALM allow you to programmatically connect to devices, retrieve information, and apply configurations, you are ready to take the next step and learn how to manage multiple devices more efficiently. Whereas Netmiko and NAPALM excel at interacting with one device at a time, Nornir is designed from the ground up to handle network automation at scale.

Importantly, Nornir does not replace Netmiko and NAPALM; it actually uses them behind the scenes to connect to devices and perform tasks. The knowledge and skills you have developed so far in the book will continue to be valuable as you move into learning about Nornir and scaling your network automation workflows.

[Chapter 9, "Introducing NORNIR: A Pythonic Framework for Network Orchestration,"](#) introduces Nornir, a powerful Python automation framework that helps organize devices, run tasks in parallel, and coordinate complex workflows across your entire network—all without leaving the familiar Python environment.

Summary

In this chapter, you have learned how to safely and programmatically configure Cisco devices using NAPALM. Unlike manual CLI configuration, NAPALM provides a structured workflow where changes are staged, compared, and applied only when explicitly committed. This minimizes the risk of mistakes and increases operational safety.

In this chapter, you have learned about the key NAPALM configuration methods:

- **load_merge_candidate()** to add new commands
- **load_replace_candidate()** to fully replace the running configuration
- **compare_config()** to preview changes
- **commit_config()** to apply changes
- **discard_config()** to cancel staged changes
- **rollback()** to undo changes before the session ends

With the help of practical examples, you have learned how to stage and commit IPv6 address configurations, save the running configuration to the startup configuration, and handle cases where **save_config()** is unsupported. You have also used the **cli()** method to send operational commands like **show ipv6 interface brief** and retrieve results in a Python dictionary.

You have learned about the importance of enabling SCP and creating an admin privilege 15 account for NAPALM configuration methods, and you have learned about configuring Cisco's **archive** feature to allow full configuration replacement with **load_replace_candidate()**.

Finally, you have seen how to retrieve and display the full running configuration and startup configuration by using the **get_config()** method, which allows for easy programmatic inspection of device state.

Throughout this chapter, we have emphasized the importance of careful staging, previewing, and verification when automating network changes to ensure reliability and stability in your network automation workflows.

Part 3: Nornir

Chapter 9. Introducing Nornir: A Pythonic Framework for Network Orchestration

Nornir is a Python-based automation framework designed specifically for working with network infrastructure. Its name comes from Norse mythology, where the Norns were said to control the fate of gods and humans. In a similar way, Nornir gives network engineers more control over the future of their automation projects. Some of the same developers who helped create Netmiko and NAPALM have also contributed to Nornir, so you'll notice many familiar ideas.

Note

Nornir was originally created by David Barroso, the same engineer behind the NAPALM automation framework, and is maintained by a community of network automation professionals. Kirk Byers, the creator of Netmiko, has also contributed to the broader Nornir ecosystem and helped popularize its use in network automation. Together, Nornir, Netmiko, and NAPALM form a complementary open-source toolkit for automating network devices using Python. Nornir remains an actively maintained open-

source project hosted on GitHub and documented at nornir.readthedocs.io.

In this book, you have used Netmiko to connect to devices using SSH and send CLI commands through Python. You have also explored NAPALM, which makes it easier to gather structured, consistent data from different network vendors. Now you'll learn about Nornir, a more advanced tool that helps you manage tasks across many devices at once.

Note

If you're new to Python, some of the examples in this chapter might be a bit challenging at first. Don't worry: We'll walk you through each step of the code and explain what it does. You don't need to be a Python expert to follow along and understand what this code is accomplishing. By the end of [Part 3](#), “[Nornir](#),” you'll see how Nornir can help you automate tasks across multiple devices with just a few lines of code—and why it's such a powerful tool for network automation.

Other tools, like Ansible and Terraform, also automate infrastructure, but they use their own languages. Ansible uses YAML (Yet Another Markup Language), a text-based format for configuration files, and Terraform uses HCL (HashiCorp Configuration Language). These tools are great, but they require you to learn extra syntax on top of your automation logic. In contrast, Nornir lets you write everything directly in Python, which is a big advantage if you already know Python—as you do by now.

One of Nornir's biggest strengths is that it runs tasks across many devices at the same time. This capability, called *parallel execution*, is powered by a technique called *multi-threading*, in which multiple actions happen at once. Instead

of working with one device at a time, Nornir handles many devices in parallel. This makes your automation run faster and more efficiently, and it works just as well whether you're managing 5 devices or 500.

Nornir is also highly customizable. It's built using small, modular pieces called *plugins* that handle specific parts of your automation, such as device connections, task execution, and logging. You can even create your own plugins if needed, but for now, we'll stick to the built-in plugins.

Nornir includes its own *inventory system*, which is how it keeps track of devices and their details. You can define your inventory by using simple YAML files or connect to external systems like NetBox. These inventory files contain information about hosts (devices), device groups (such as switches or routers), credentials, and other variables. Devices can be grouped based on location, function, or any other category that's useful for your automation. Once your inventory is in place, Nornir automatically connects to all the devices in the group without needing **for** loops. It handles the connections for you, using its built-in multi-threaded engine behind the scenes.

Starting in this chapter, you'll see how Nornir works together with Netmiko and NAPALM using special plugins. Thanks to its modular design, Nornir can use these libraries directly, combining the power of structured data, CLI access, and large-scale task orchestration all in one place.

What Is Orchestration?

In network automation, *orchestration* means coordinating and managing many tasks across multiple devices in a structured and automated way. Think of it like conducting an orchestra, where each musician (or network device) plays a

part, and the conductor (your automation framework) ensures that all the musicians (network devices) play in sync. With Nornir, orchestration allows you to define what needs to happen (such as pushing a configuration or checking interface status) and ensures that it happens across all your devices in a consistent and organized manner—often in parallel. It makes automation powerful at scale.

Here are some common tasks that Nornir can help automate:

- **Configuration management:** Nornir can push configuration changes to one or many devices at once.
- **Provisioning:** It can bring new devices online quickly, with consistent settings.
- **Network testing and monitoring:** It can check device health, uptime, and connectivity status.
- **Backups:** Nornir can save running or startup configurations for disaster recovery.
- **Device validation:** You can use Nornir to confirm that device settings match what you expect.
- **Compliance checks:** Nornir allows you to ensure that devices follow security or policy requirements.

How Does Nornir Compare to Netmiko and NAPALM?

In this book, we've chosen to focus on three essential Python libraries for network automation: Netmiko, NAPALM, and Nornir. Each offers a different layer of capability, and together they provide a natural learning progression for building automation skills.

We began with Netmiko, which allows you to automate command-line interactions with network devices over SSH—essentially using Python to replicate what a network engineer would do manually. It's a great starting point because it shows how you can use Python to automate common tasks on real devices.

Next, we explored NAPALM, which introduced the idea of structured data and vendor-agnostic automation. With NAPALM, you do not have to parse raw CLI output. Instead, you can work with clean, consistent data formats such as dictionaries, which make it easy to gather and apply configurations across different vendors.

Now we've arrived at Nornir, which brings everything together into a flexible automation framework. Nornir doesn't replace Netmiko or NAPALM; it *uses them*. It acts as a controller, allowing you to organize tasks, run the tasks across many devices at once, and reuse code you've already written. What makes Nornir especially powerful is that it handles device connections for you automatically and runs tasks in parallel, so you don't need to manually create loops or manage timing.

In short:

- Netmiko is great for learning how to connect and run commands.
- NAPALM introduces structured, vendor-neutral data handling.
- Nornir adds orchestration, letting you automate at scale by using libraries like Netmiko and NAPALM together in a more organized, efficient way.

If you've followed along with the book so far, you're ready for this next step. Nornir builds on what you already know and gives you more power and flexibility.

How Nornir Uses Netmiko and NAPALM

One of the most important things to know about Nornir is that it doesn't try to replace tools like Netmiko or NAPALM; it works with them. Nornir acts as a controller or manager, using Netmiko and NAPALM as plugins to talk to network devices. With Netmiko, NAPALM, and Nornir, you get the best of all worlds: the task coordination and speed of Nornir, combined with the connection and configuration features of Netmiko and NAPALM.

A *plugin* is an optional add-on that extends a program's capabilities without modifying its core code. It "plugs in" to the main application to provide additional features or specialized functions. [Figure 9-1](#) shows how Nornir uses Netmiko and NAPALM as plugins. Nornir provides orchestration and parallel execution, while Netmiko and NAPALM handle the device-level communication.

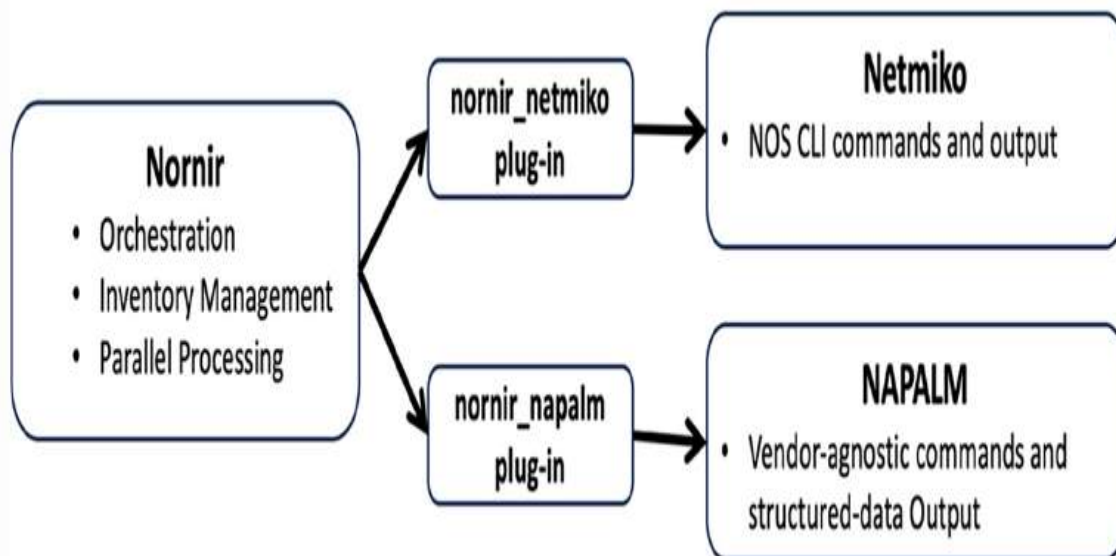


Figure 9-1 *Nornir uses Netmiko and NAPALM for device management*

Nornir uses plugins to integrate with different network automation libraries:

- **nornir_netmiko:** This library lets Nornir use the Netmiko library to send CLI commands and configurations over SSH to network devices.
- **nornir_napalm:** This library lets Nornir use the NAPALM library to retrieve and manage device information through structured APIs (for example, getting facts, getting interfaces, or comparing configurations).

When Nornir runs a task, it can use Netmiko to connect to devices over SSH and send NOS (network operating system) CLI commands—just as you’ve done in earlier chapters. Or it can use NAPALM to gather structured data and apply configurations in a vendor-neutral way. Instead of choosing one or the other, you can mix and match based on what a particular task requires.

You can use Nornir for orchestration and parallel execution and let Netmiko and NAPALM do the device-level work. Nornir handles the *how* (such as which devices to connect to, in what order, and how many at a time), and Netmiko and NAPALM handle the *what* (the specific commands or data you want to retrieve or configure). This modular approach makes your automation cleaner, faster, and easier to scale.

Installing Nornir

At the time of this writing, the latest version of Nornir is 3.5, and it requires Python 3.9 or higher. You can install Nornir by using pip, the standard Python package manager:

```
pip install nornir
```



Since version 3.0, Nornir has not included built-in support for any plugins. However, you can use plugins to extend Nornir's functionality—for example, to allow it to work with libraries like Netmiko and NAPALM. You can install plugins separately when needed.

Note: In this chapter you will not be using any plugins. In [Chapter 10, “Using Nornir with Netmiko,”](#) you will connect to routers and use Nornir with the `nornir_netmiko` plugin to execute commands using Netmiko.

Basic Nornir Framework: Python and YAML

[Figure 9-2](#) shows the three-router topology used throughout this chapter. The routers (R1, R2, and R3) are connected through point-to-point links, with R1 and R3 providing access to their local LANs. All the Nornir examples in this chapter use the same topology, with Nornir connecting to each router over SSH to run commands and gather information.

However, this chapter does not show connections to the actual devices. Instead, you will see how Nornir reads the information stored in YAML files—and the YAML files will refer to the routers in the topology shown in [Figure 9-2](#).

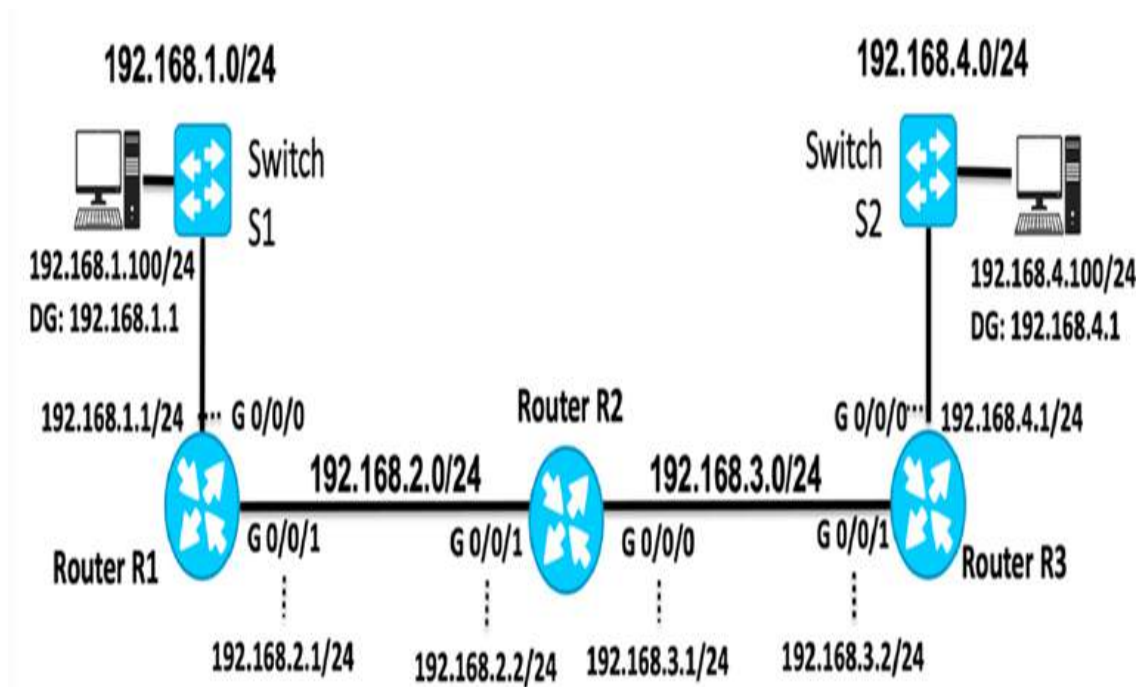


Figure 9-2 *Topology for the examples in this chapter*

Before looking at the Nornir code, let's first discuss how Nornir organizes its inventory. Nornir separates device information into several YAML files—one for hosts, one for groups, and one for default settings, all referenced by a central configuration file. [Figure 9-3](#) shows how these files relate to one another and to a Python program.

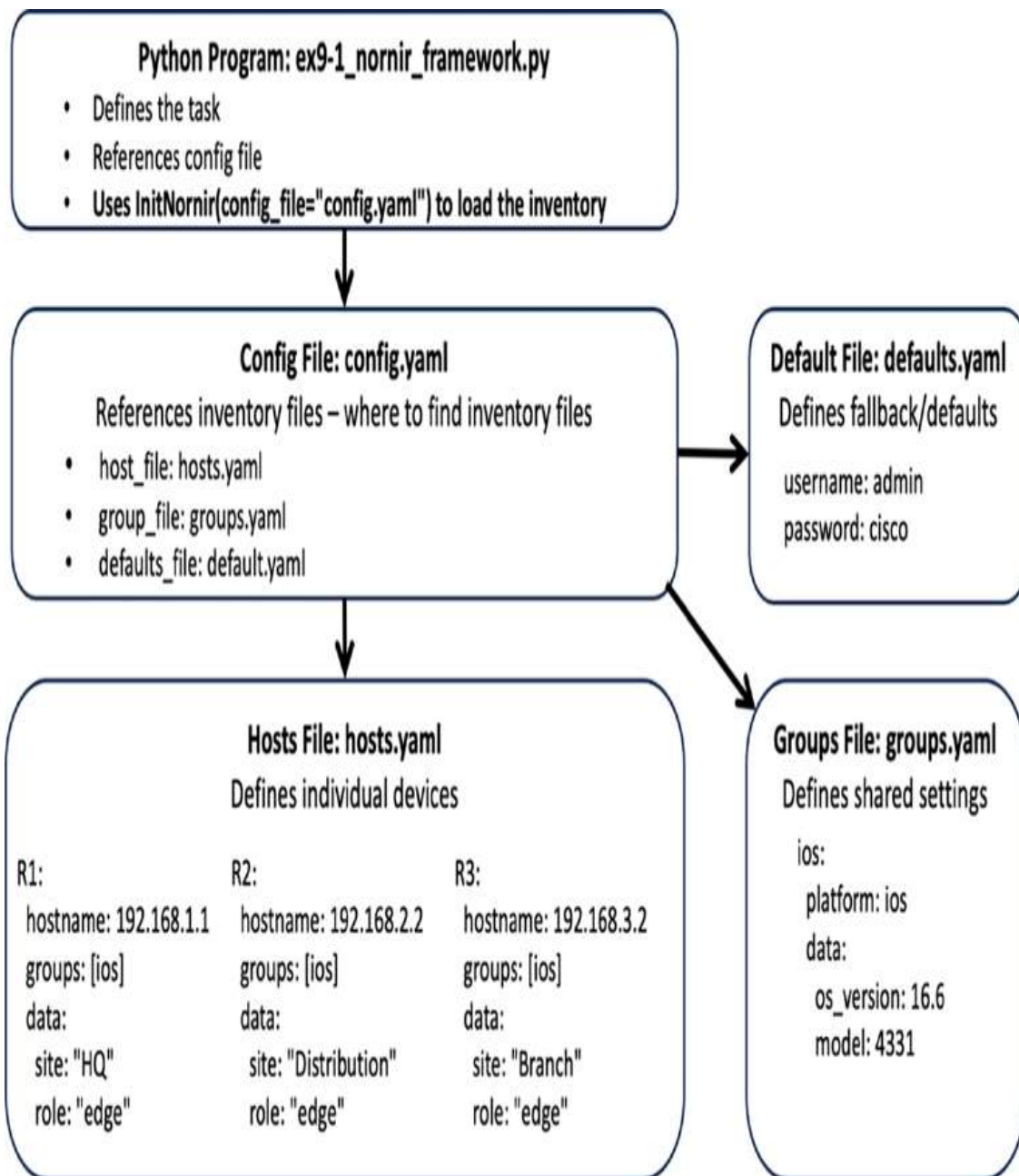


Figure 9-3 *How Nornir uses configuration and inventory files to generate task output*

The config.yaml file tells Nornir where to find the inventory files. The inventory is divided into three parts: hosts.yaml, groups.yaml, and defaults.yaml (all discussed in the next section).

To run any Nornir program, you need at least one required file, `config.yaml`. This file points to three optional—but almost always used—inventory files:

- **hosts.yaml:** Lists the individual devices in the network, with one entry per router or switch.
- **groups.yaml:** Defines settings shared among similar devices, such as the operating system or device type.
- **defaults.yaml:** Provides fallback information such as usernames and passwords used when a value isn't specified elsewhere.

When you initialize Nornir in a Python program, it reads all these files, combines their information, and builds an internal inventory of your devices. Once the inventory is loaded, Nornir can connect to each device and begin executing tasks in parallel.

Your First Nornir Program and YAML Files

This section starts with a simple Nornir program that confirms that your configuration and inventory files are working correctly. This first example does not use Netmiko or NAPALM and does not make SSH connections to the routers. Instead, it tests and demonstrates how Nornir reads the information stored in the YAML files `config.yaml`, `hosts.yaml`, `groups.yaml`, and `defaults.yaml`.

This is an important first step. If Nornir can successfully load and display the inventory, then you know that your file paths and structure are correct. Once you've confirmed that, you can move on to using IOS commands to view and configure the actual routers in [Chapter 10](#).

The Python File

With the YAML files in place, the next step is to write a small Python program that uses Nornir to read that inventory.

[Example 9-1](#) shows a simple Nornir program that loads the config.yaml file and prints a summary of each device so you can confirm that everything is wired together correctly.

Example 9-1 *ex9-1_nornir_inventory_check.py*

```
# 1. Import Nornir module InitNornir
from nornir import InitNornir

# 2. Initialize Nornir using the configuration file
nr = InitNornir(config_file='config.yaml')

# 3. Loop through each device in the inventory and print its de
for name, host in nr.inventory.hosts.items():
    print(f'Host: {name}')
    print(f'  Hostname (IP/DNS): {host.hostname}')
    print(f'  Platform: {host.platform}')
    print(f'  Username: {host.username}')
    print('-' * 40)
```

This short program helps verify that Nornir can correctly load and interpret the information from your YAML files. Nornir is not communicating with the routers; it's just displaying the structured data it has read. Let's go through it step by step:

Step 1. Import Nornir: The program imports only one function (sometimes referred to as a *module*): **InitNornir**. This function is responsible for initializing Nornir and loading your inventory and

configuration files. No extra libraries or SSH connections are needed yet.

```
from nornir import InitNornir
```

Step 2. Initialize Nornir: This line tells Nornir to load its configuration and inventory files:

```
nr = InitNornir(config_file="config.yaml")
```

The **config.yaml** file specifies which inventory plugin to use and where to find the host, group, and default files.

Step 3. Print host details: The **for** loop walks through all devices in the inventory and prints key information: device name, platform, username, and hostname (IP address). These details come entirely from the local YAML files and are explained later in this chapter.

Note

We will examine the code used to initialize Nornir and print the host details later in this chapter, after we have discussed the YAML files.

The config.yaml File

In order to display information about your devices, Nornir needs to know where those devices are and what their settings are. This is where the config.yaml file comes in. This file can reference three additional inventory files—hosts.yaml, groups.yaml, and defaults.yaml—which are almost always used in practice. Together, these files describe your network devices, shared attributes, and default login information.

Example 9-2 shows the config.yaml file.

Example 9-2 *config.yaml*

```
inventory:
  plugin: SimpleInventory
  options:
    host_file: "hosts.yaml"
    group_file: "groups.yaml"
    defaults_file: "defaults.yaml"
runner:
  plugin: threaded
  options:
    num_workers: 5
```

Note

If you are familiar with YAML, you may notice that there is no `---` at the start of the file. The `---` is optional. It's technically a document start marker that is used when:

- A file contains **multiple YAML documents** separated by `---`, or
- You want to make the file explicitly YAML (which is useful in certain systems or pipelines).

The config.yaml file is the main entry point for Nornir. It tells Nornir how to build its inventory and how to run tasks. The config.yaml file has two main sections—**inventory** and **runner**—each of which contains key/value pairs that define how Nornir behaves (for example, **plugin: SimpleInventory**, **host_file: "hosts.yaml"**, and **num_workers: 5**).

The **inventory** section defines which inventory plugin Nornir should use and where to find the inventory files. It includes the following key/value pair and subsection:

- **plugin: SimpleInventory:** This pair tells Nornir to use the built-in SimpleInventory plugin, which loads information from local YAML files.
- **options:** This subsection specifies the locations of the inventory files:
 - **host_file:** Points to the file that lists the individual devices (**hosts.yaml**).
 - **group_file:** Points to the file that defines shared group settings (**groups.yaml**).
 - **defaults_file:** Points to the file **that** contains default values (**defaults.yaml**).

The **runner** section controls how Nornir executes tasks. It includes the following key/value pair and subsection:

- **plugin: threaded:** This pair tells Nornir to use a threaded execution model, which allows tasks to run on multiple devices at the same time (parallel processing).
- **options:** This subsection provides additional settings for the runner, including the following:
 - **num_workers: 5:** Indicates that Nornir can run up to five tasks in parallel, which is usually plenty for lab environments or small topologies.

The **plugin** lines in the config.yaml file are required; they tell Nornir which internal component or plugin to use. If there is no plugin specified, Nornir does not know how to load the inventory or execute tasks. The **options:** sections are optional, but strongly recommended.

The **runner** section in the config.yaml file is optional. If it's omitted, Nornir uses its default serial runner, which executes tasks on one device at a time.

In summary, config.yaml tells Nornir, "Use the SimpleInventory plugin, read my device information from these YAML files, and run tasks on up to five devices at once."

The hosts.yaml File

The hosts.yaml file, as shown in [Example 9-3](#), lists each individual device in the network. You can see that each device or host is a top-level section (**R1**, **R2**, and **R3**), with key/value pairs and subsections. Each section or entry defines the device's IP address (or DNS name), the group it belongs to, and any additional custom data you want to associate with it.

Example 9-3 *hosts.yaml*

```
R1:
  hostname: 192.168.1.1
  groups: [ios]
  data:
    site: "HQ"
    role: "edge"

R2:
  hostname: 192.168.2.2
  groups: [ios]
  data:
    site: "Distribution"
    role: "core"
```

```
R3:
  hostname: 192.168.3.2
  groups: [ios]
  data:
    site: "Branch"
    role: "edge"
```

Let's look at each part:

- **R1, R2, and R3 (device names):** Each of these sections includes the key/value pairs for each device. Each device is identified by a unique name that you use when referring to it in your Nornir programs. These names do not need to match the device's actual hostname, but using consistent names helps keep your inventory organized.
- **hostname:** This is the IP address or DNS name of the device—the value Nornir (and later Netmiko) uses to connect to it. In larger or enterprise networks, it may instead be a DNS hostname (for example, `r1.lab.example.com`) if name resolution is configured. Either way, Nornir uses this value to establish the SSH connection to the device.
- **groups:** This specifies which group or groups the device belongs to. In this example, all devices belong to the **ios** group defined in `groups.yaml`. Group membership allows devices to share common attributes such as platform type or OS version. This entry directly references the **ios** group defined in the `groups.yaml` file discussed next, allowing the device to inherit that group's shared settings automatically. Devices can belong to multiple groups separated by a comma—for example, **[ios, core]**. The square brackets (**[]**) indicate that the value is a YAML list (even if it contains only one

item). The list tells Nornir that this host belongs to one or more groups defined in groups.yaml. (The groups.yaml file is discussed next.)

- **data:** The **data** subsection is optional and can include any custom information you want to associate with the device. In this example, each router includes two **data** fields (keys):
 - **site:** Identifies where the device is located (HQ, Distribution, or Branch).
 - **role:** Describes its function in the network (edge or core).

These additional key/value pairs can be used later in your programs for filtering, labeling, or reporting.

In the hosts.yaml file, each device must include at least one required field—**hostname**, which specifies the device's IP address or DNS name. All other fields, such as **groups** and **data**, are optional but strongly recommended because they make your inventory more organized and reusable across multiple automation tasks.

In short, hosts.yaml defines the *who* and *where* of your network—the specific devices Nornir will manage and key details about each one.

Note

The inventory files (hosts.yaml, groups.yaml, and defaults.yaml) must use reserved keys where required, such as **hostname** or **groups** in hosts.yaml. However, in the **data:** section, you can create any custom keys your program needs.

The groups.yaml File

The groups.yaml file, as you can see in [Example 9-4](#), defines shared attributes for devices that belong to the same group. Groups are useful when you have multiple devices with similar characteristics, such as the same operating system or hardware model. Rather than repeat those details for every host, you define them once in a group and assign devices to that group in hosts.yaml.

Example 9-4 groups.yaml

```
ios:
  platform: ios
  data:
    os_version: 16.6
    model: 4331
```

Let's look at each part:

- **ios - Group name:** This section identifies a collection of devices with shared settings. In this example, all routers are assigned to the **ios** group. This was the item in the **groups** YAML list we discussed for the hosts.yaml file. You can have multiple groups (for example, **ios**, **core**), each one as its own top-level section with key/value pairs.
- **platform:** This section defines the operating system or driver type Nornir will use for this group. Here, the value **ios** indicates that these devices use Cisco IOS. This setting becomes important when using plugins such as Netmiko or NAPALM, which need to know the platform to select the right connection method.
- **data:** As in hosts.yaml, the **data** subsection contains optional custom information that applies to all devices

in this group. In this example:

- **os_version:** Specifies the operating system version shared by all routers in the group
- **model:** Defines the router hardware model

Any host that belongs to the **ios** group automatically inherits these settings unless it overrides them in its own hosts.yaml entry.

In short, groups.yaml helps eliminate repetition and keeps your inventory organized by defining settings once for all similar devices.

The **groups** key in hosts.yaml links a device to shared attributes in groups.yaml, as shown in [Figure 9-4](#).

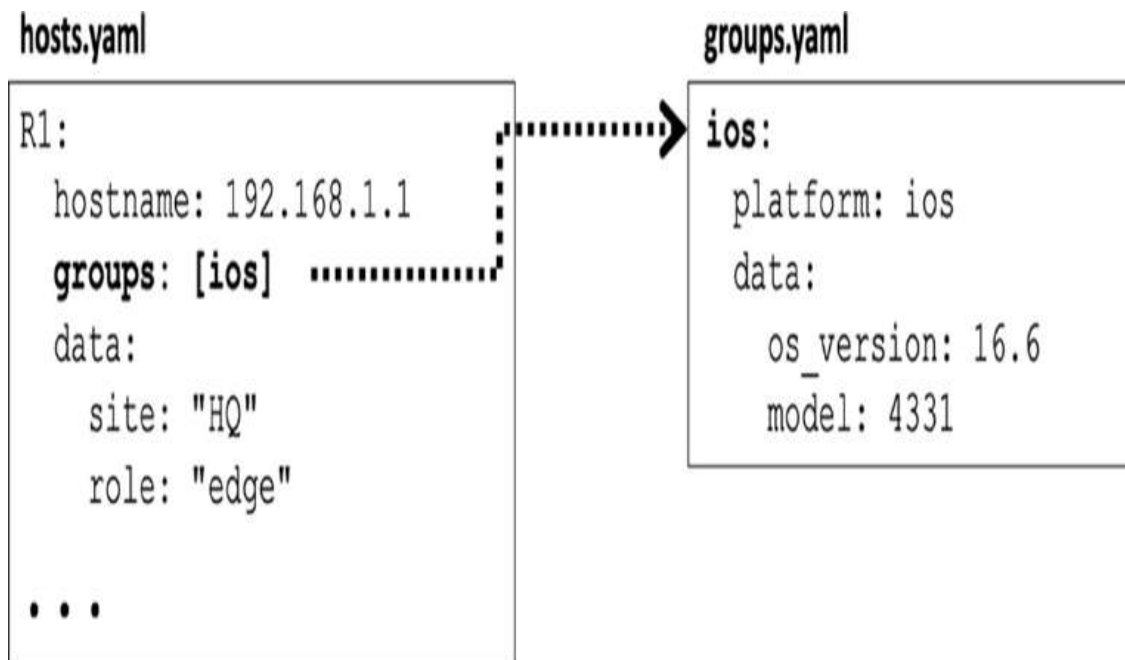


Figure 9-4 *The relationship between hosts.yaml and groups.yaml*

When Nornir loads the inventory, each host inherits all group-level settings, which can be overridden by host-

specific values defined directly in that host's entry within `hosts.yaml`.

For example, if `groups.yaml` specifies this:

```
ios:
  platform: ios
  data:
    model: 4331
```

but the `hosts.yaml` entry for R1 includes this:

```
R1:
  hostname: 192.168.2.1
  groups: [ios]
  data:
    model: 4451
```

then Nornir will use **model 4451** for R1 because host-level values always take precedence over group-level ones.

In the `groups.yaml` file, each group must have a top-level section or group name (such as **ios:**). While no other fields are strictly required, it's best practice to include a **platform** setting so Nornir and its plugins know how to communicate with those devices. The optional **data** section can store shared values, such as model or software version, that apply to every device in the group.

The defaults.yaml File

[Example 9-5](#) shows the `defaults.yaml` file, which defines the fallback values that apply to all devices in the inventory unless a specific host or group overrides them. Defaults are especially useful for storing settings that are common to every device, such as login credentials.

Example 9-5 *defaults.yaml*

```
username: admin
password: cisco
```

Let's look at each part of the file:

- **username:** Specifies the default username that Nornir (and any connection plugin, such as Netmiko or NAPALM) will use when connecting to devices. In this example, all routers use the same username: **admin**.
- **password:** Specifies the default password for all devices. Again, this can be overridden in an individual host entry if a specific device uses different credentials. In this example, all routers use the same password: **cisco**.

The **username** and **password** values in the defaults.yaml file are used for SSH authentication when Nornir connects to devices through plugins such as Netmiko or NAPALM. These credentials serve as the default login information for all devices unless a specific host or group defines its own credentials in the inventory.

Both the groups.yaml and hosts.yaml files can also contain username and password key/value pairs. When Nornir initializes, it automatically merges these defaults with the data from groups.yaml and hosts.yaml.

If the username and password are defined in the groups.yaml file, this will take precedence over the defaults.yaml file. If the username and password are defined in the hosts.yaml file, this will take precedence over both the groups.yaml and defaults.yaml files.

The defaults.yaml file is completely optional, but it's highly recommended because it defines values that apply to all devices when not specified elsewhere. No fields are

required, but in most cases, you'll include at least the username and password so Nornir and its connection plugins (like Netmiko or NAPALM) know how to log in. Any host or group that defines its own credentials will override these default values.

In short, `defaults.yaml` provides a convenient way to define settings that most or all devices share—keeping your inventory clean, consistent, and easy to maintain.

Note

For simplicity, this example stores the username and password directly in `defaults.yaml`. However, this is not recommended for production environments. In real deployments, credentials should be stored securely—such as in environment variables, in a secrets manager (for example, HashiCorp Vault, Ansible Vault), or in a separate file that is excluded from version control—to protect sensitive information and maintain good security practices.

How These YAML Files Work Together

Together, the four YAML files form the foundation of Nornir's inventory system:

- The `config.yaml` file is the entry point. It tells Nornir which inventory plugin to use and where to find the other files.
- The `hosts.yaml` file lists each individual device.
- The `groups.yaml` file defines shared characteristics for devices that have something in common, such as platform type or operating system.

- Finally, `defaults.yaml` provides fallback values, such as usernames and passwords, that apply to all devices unless overridden.

You initialize Nornir with this command:

```
nr = InitNornir(config_file="config.yaml")
```

Nornir reads and merges all this information into a single internal structure, **nr**. At this point, Nornir knows every device in your network, what platform it runs, and how to log in—all before you run any automation tasks.

The variable **nr** is a Python object that represents Nornir's internal data structure. This object contains:

- The full inventory (all hosts, groups, and defaults merged together)
- The runner settings (such as the number of worker threads)
- The methods to execute tasks (like **nr.run()**) (You'll learn about executing tasks and **nr.run()** in [Chapter 10](#).)

The Output

[Example 9-6](#) shows the output that results from running the code in [Example 9-1](#). Here are the steps that occur when the program runs:

Step 1. `InitNornir(config_file="config.yaml")` tells Nornir to load the inventory using the `SimpleInventory` plugin and read from the three YAML files listed.

Step 2. `hosts.yaml` provides each device's name (R1, R2, R3), IP address, and group membership (**ios**).

Step 3.groups.yaml supplies shared attributes for the **ios** group, including the platform type (**ios**), which each host inherits.

Step 4.defaults.yaml provides fallback credentials (**username: admin, password: cisco**) used by all hosts.

Step 5.The program prints a summary of what Nornir loaded for each device:

- The hostname (from hosts.yaml)
- The platform (from groups.yaml)
- The username (from defaults.yaml)
- The device IP address (from hosts.yaml)

Example 9-6 Output from [Example 9-1](#)

```
MyPrompt% python3 ex9-1_nornir_inventory_check.py
Host: R1
  Hostname (IP/DNS): 192.168.1.1
  Platform: ios
  Username: admin
-----
Host: R2
  Hostname (IP/DNS): 192.168.2.2
  Platform: ios
  Username: admin
-----
Host: R3
  Hostname (IP/DNS): 192.168.3.2
  Platform: ios
  Username: admin
```

MyPrompt%

How did the following Python code in **ex9-1_nornir_inventory_check.py** produce the output shown in [Example 9-6](#)?

```
for name, host in nr.inventory.hosts.items():
    print(f'Host: {name}')
    print(f'  Hostname (IP/DNS): {host.hostname}')
    print(f'  Platform: {host.platform}')
    print(f'  Username: {host.username}')
    print('-' * 40)
```

When Nornir initializes, it merges information from the three inventory files (`hosts.yaml`, `groups.yaml`, and `defaults.yaml`) into a single in-memory data structure. Nornir creates an inventory object, which contains a dictionary of host objects.

Each host object stores all of the merged information for one device. You can access its individual attributes by using dot notation (for example, **`host.hostname`**, **`host.platform`**, or **`host.username`**). The variable **`name`** comes from the top-level section of each device in the `hosts.yaml` file (for example, `R1`, `R2`, `R3`). The variable **`host`** refers to the corresponding host object that contains all of that device's merged inventory data, such as **`hostname`**, **`platform`**, and **`username`**, combined from `hosts.yaml`, `groups.yaml`, and `defaults.yaml`.

[Figure 9-5](#) shows which file each printed value originates from. As mentioned previously, a key in the `host.yaml` file—for example, `R1`—is referred to as the device name.

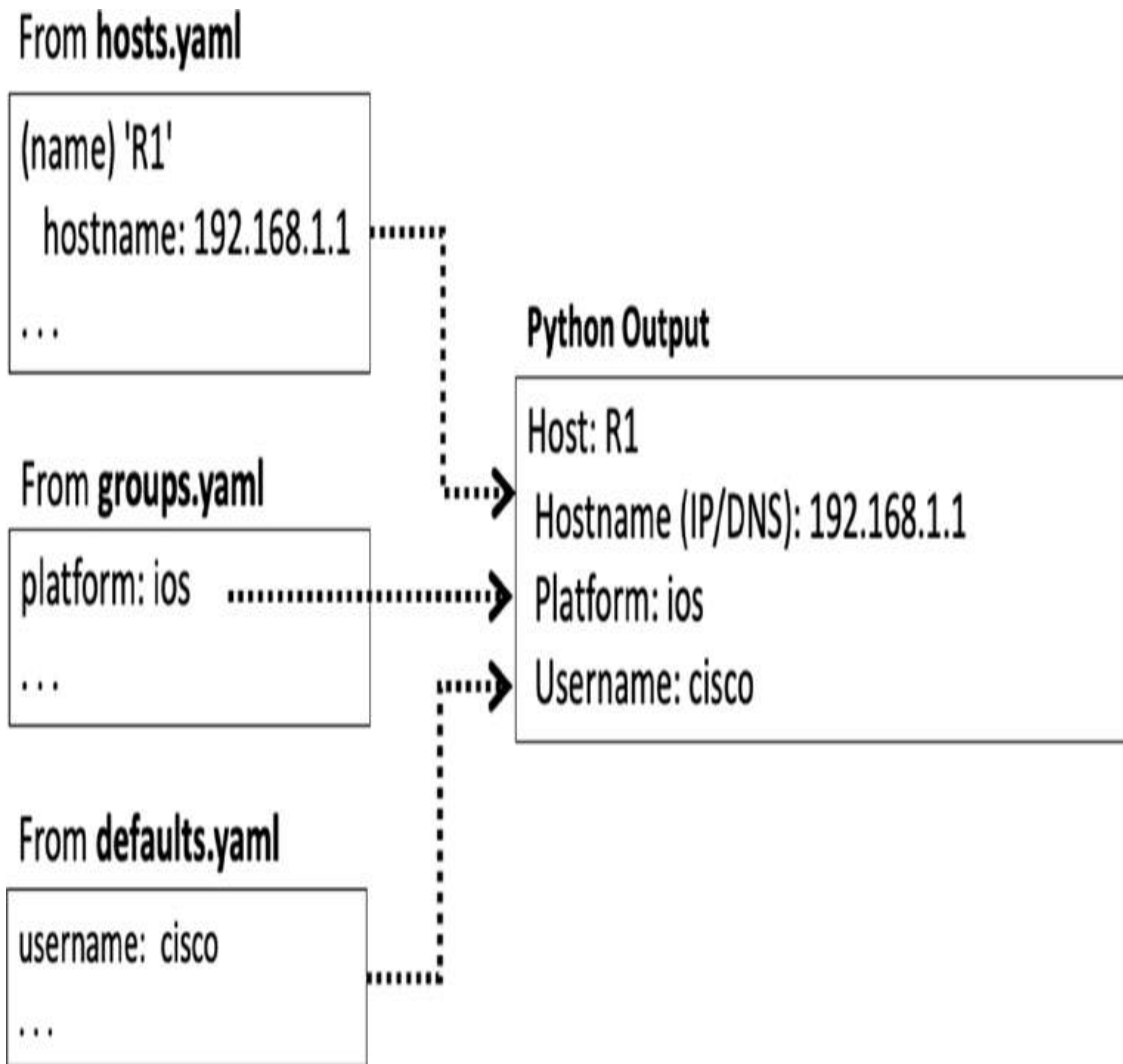


Figure 9-5 *The relationship between `host.yaml` and `groups.yaml`*

The output confirms that Nornir correctly read and merged data from all three inventory files. *No SSH connections have occurred yet.* This is purely inventory data, not device output.

What's Next?

If you've made it this far, you've already done the heavy lifting. You've seen how Nornir is structured, how it uses a configuration file to load inventory data, and how that data

is organized using hosts, groups, and defaults. If any of this still feels a little overwhelming, don't worry. Nornir is a more advanced tool than Netmiko and NAPALM, and with practice, you will find it much easier to work with. Just like learning to configure a new network device, using Netmiko, NAPALM, and Nornir takes a bit of repetition and hands-on experience to help you feel comfortable.

The next chapters show how to use the full power of Nornir by integrating it with the tools you've already learned—Netmiko and NAPALM. Nornir doesn't replace them; rather, it works with them. Think of Nornir as the conductor and Netmiko and NAPALM as the musicians: Nornir coordinates the work across many devices, while Netmiko and NAPALM do the actual device interaction—sending commands, gathering outputs, and making configuration changes.

In the next chapter, you'll use this same inventory to actually connect to routers and execute commands using Netmiko.

Summary

In this chapter, you were introduced to Nornir, a Python-based network automation framework that brings together everything you have learned about Netmiko and NAPALM. Unlike those individual libraries, Nornir acts as a controller and orchestrator, managing how and when automation tasks run across many devices in parallel.

You have learned that Nornir's foundation is its inventory system, which is built from four YAML files:

- **config.yaml:** Tells Nornir which plugin to use and where to find the inventory.
- **hosts.yaml:** Lists individual devices and their key details, such as IP address, group membership, and role.

- **groups.yaml:** Defines shared attributes (like platform type or OS version) for similar devices, reducing repetition.
- **defaults.yaml:** Provides fallback values, such as common SSH credentials, that are used unless they are overridden by a host or group.

When you initialize Nornir with

InitNornir(config_file="config.yaml"), it merges all three inventory files into a single in-memory structure. Each device became a host object that is accessible through **nr.inventory.hosts**, and you can reference details like **host.hostname** or **host.platform** directly in Python. You can use the sample program in this chapter to confirm that your inventory has loaded correctly—before you make any SSH connections or executing commands—so you can verify that your Nornir environment is ready for automation.

This first step is essential: It ensures that your file paths, plugins, and inventory hierarchy work as expected. The next chapter builds on this foundation by introducing Netmiko as a Nornir plugin, using the same inventory to connect to real routers, send commands, and collect live output from your network devices.

Chapter 10. Using Nornir with Netmiko

In [Chapter 9, “Introducing Nornir: A Pythonic Framework for Network Orchestration,”](#) you learned how to initialize and run Nornir, a Python-based framework that is designed to orchestrate automation tasks across multiple network devices in parallel. You also saw how Nornir organizes inventory, manages connections, and executes tasks efficiently through its plugin-based architecture.

This chapter builds on that foundation by exploring `nornir_netmiko`, a plugin that integrates Nornir with the Netmiko library. In [Part 1, “Netmiko,”](#) you used Netmiko to establish SSH connections and send commands directly to Cisco IOS devices. Now, you’ll see how to combine that CLI flexibility with Nornir’s orchestration and multi-threading capabilities in order to run Netmiko commands across many devices simultaneously.

Installing Support for Netmiko: `nornir_netmiko`

`nornir_netmiko` is a plugin library that extends Nornir’s capabilities by allowing it to use Netmiko under the hood for

connecting to network devices and running commands. At the time of this writing, the latest release of this plugin is version 1.0.1, which requires Python 3.8 or later.

To install the plugin, run the following command:

```
pip install nornir_netmiko
```

This command installs the `nornir_netmiko` plugin, which allows Nornir to use Netmiko for tasks such as **`netmiko_send_command`** and **`netmiko_send_config`**, which you will be using in this chapter. If they are not already installed, it also installs Nornir, the core automation framework, and Netmiko, the SSH library that connects to network devices and executes NOS (network operating system) commands.

Once the `nornir_netmiko` plugin is installed, Nornir can use Netmiko's SSH engine as a task plugin to connect to devices, send commands, and collect results—while also taking advantage of Nornir's parallel task execution and inventory management.

[Figure 10-1](#) illustrates the relationship between Nornir and plugins for Netmiko and Nornir.

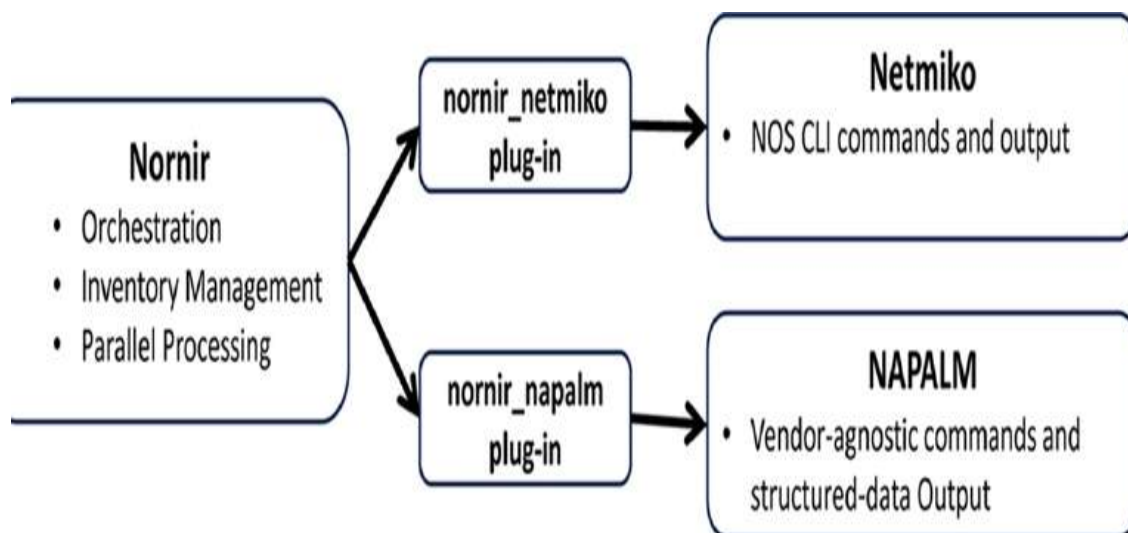


Figure 10-1 *Nornir uses Netmiko and NAPALM for device management*

Once `nornir_netmiko` is installed, you can begin using it to connect to real network devices.

In the next section, you'll see how to build a simple Nornir program that uses the **`netmiko_send_command`** task to send CLI commands—such as **`show ip interface brief`**—to multiple devices in parallel. This example demonstrates how Nornir and Netmiko work together to automate operations efficiently across an entire inventory.

Using Nornir and `netmiko_send_command()`

[Figure 10-2](#) shows the same three-router topology we have been using throughout this book and that we will continue to use in this chapter. In [Chapter 9](#), you created a Nornir program to display device information directly from the inventory, but you did not actually connect to any routers or switches.

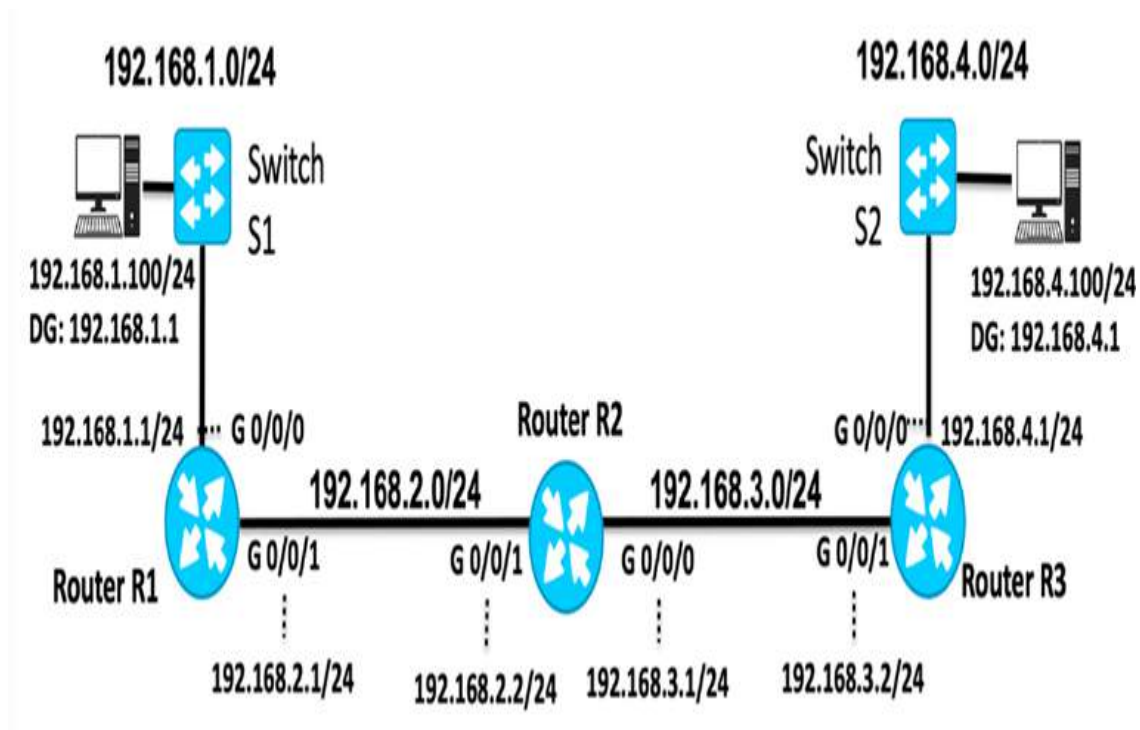


Figure 10-2 *Topology for the examples in this chapter*

Now, you're ready to take the next step and use `nornir_netmiko` to establish SSH connections and send commands to multiple devices at once. We'll begin with one of the most familiar IOS commands, **`show ip interface brief`**, to verify interface status on all devices in the inventory.

The program in [Example 10-1](#) builds on everything you've learned so far about Nornir's configuration and inventory files. In this example, Nornir uses the `nornir_netmiko` plugin to connect to multiple devices, run a command in parallel, and return the results. We'll go through each part of this program in detail later in this chapter.

Example 10-1 `ex10-1_nornir_netmiko_send_command.py`

```
# 1. Import required modules
from nornir import InitNornir
```

```
from nornir_netmiko.tasks import netmiko_send_command

# 2. Initialize Nornir with the configuration file
nr = InitNornir(config_file='config.yaml')

# 3. Run the task across all hosts
result = nr.run(
    task=netmiko_send_command,
    command_string='show ip interface brief'
)

# 4. Display the output for each device
for host, multi_result in result.items():
    print(f'\n\n-----{host}-----')
    print(multi_result[0].result)
```

The config.yaml file (see [Example 10-2](#)) along with the three inventory files—hosts.yaml (see [Example 10-3](#)), groups.yaml (see [Example 10-4](#)), and defaults.yaml (see [Example 10-5](#))—are the same files we used in [Chapter 9](#).

Example 10-2 *config.yaml*

```
inventory:
  plugin: SimpleInventory
  options:
    host_file: "hosts.yaml"
    group_file: "groups.yaml"
    defaults_file: "defaults.yaml"
runner:
  plugin: threaded
  options:
    num_workers: 5
```

Example 10-3 *hosts.yaml*

```
R1:
  hostname: 192.168.1.1
  groups: [ios]
  data:
    site: "HQ"
    role: "edge"

R2:
  hostname: 192.168.2.2
  groups: [ios]
  data:
    site: "Distribution"
    role: "core"

R3:
  hostname: 192.168.3.2
  groups: [ios]
  data:
    site: "Branch"
    role: "edge"
```

Example 10-4 *groups.yaml*

```
ios:
  platform: ios
  data:
    os_version: 16.6
    model: 4331
```

Example 10-5 *defaults.yaml*

```
username: admin
password: cisco
```

Parallel Execution with Nornir

[Example 10-6](#) displays the output from running the code in [Example 10-1](#). Although this output looks similar to what you might see when using Netmiko alone, there is an important difference in how it was produced. In this example, the **show ip interface brief** command was executed on all three routers *simultaneously*. Nornir's threaded **runner** in the config.yaml file handled the connections in parallel, collecting results as each device responded:

```
runner:
  plugin: threaded
  options:
    num_workers: 5
```

This capability is especially powerful when you're working with dozens or even hundreds of devices, where executing commands sequentially would take significantly longer.

Example 10-6 Output from [Example 10-1](#)

```
MyPrompt% python3 ex10-1_nornir_netmiko_send_command.py

-----R1-----
Interface                IP-Address      OK? Method Status
GigabitEthernet0/0/0     192.168.1.1     YES NVRAM  up
GigabitEthernet0/0/1     192.168.2.1     YES NVRAM  up
GigabitEthernet0/0/2     unassigned      YES NVRAM  administrative
GigabitEthernet0         unassigned      YES NVRAM  administrative
```

-----R2-----

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.3.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.2	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

-----R3-----

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.4.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.3.2	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

MyPrompt%

While Nornir can scale to hundreds or even thousands of devices, the number of tasks that can run truly in parallel depends on system resources such as CPU cores, available memory, and SSH session limits.

Understanding Workers in Nornir

The **num_workers** option in the config.yaml file controls how many concurrent threads, or “workers,” Nornir uses to run tasks concurrently. Here is an example of **num_workers** in the config.yaml file:

```
runner:
  plugin: threaded
```

```
options:  
    num_workers: 5
```

Setting the worker count too high can lead to connection timeouts or resource exhaustion, especially when SSH negotiations take time. The configuration currently processes up to 5 devices concurrently; any additional devices are queued until a worker is free. Increasing the number of workers can improve throughput but also raises CPU, memory, and SSH session usage. In most lab or enterprise environments, using 10 to 50 workers offer a practical balance between speed and reliability; larger automation servers may support higher values, and smaller setups may require fewer.

Note

The configuration in this chapter allows up to five devices to be processed at the same time. If your inventory contains more devices, Nornir will queue them and run the next ones as workers become available.

The Python Code: A Closer Look

[Example 10-1](#) brings together everything you learned in [Chapter 9](#) and applies it to real network devices. Some of the information in this section is also provided in [Chapter 9](#), but we include it here for a review and continuity.

Note

Some of the code in this chapter is more advanced than what you have seen previously, including the user of a user-defined function. If you are still new to Python, don't get bogged down in the details of the code. Instead, focus on the result or basic idea of

what the code is doing. In other words, don't worry about it if you don't understand the details of the code! Just focus on how to use these snippets of code to accomplish your task. If you are new to user-defined functions, near the end of this chapter there is a section, Understanding the User-Defined Function.

Let's walk through how the program **ex10-1_nornir_netmiko_send_command.py** works, step by step:

Step 1.Import modules: The program imports two key modules:

- **from nornir import InitNornir:** This imports the **InitNornir** function, which initializes the Nornir framework by loading its configuration and inventory files. It's the first step in setting up Nornir so that it knows which devices to manage and how to run tasks.
- **from nornir_netmiko.tasks import netmiko_send_command:** This line imports the **netmiko_send_command** task from the **nornir_netmiko** plugin. It allows Nornir to use Netmiko under the hood to send CLI commands—such as **show** commands—over SSH to multiple devices in parallel.

Step 2.Initialize Nornir: The line **nr = InitNornir(config_file='config.yaml')** loads the inventory information from the YAML files (**hosts.yaml**, **groups.yaml**, and **defaults.yaml**) and sets up the threading configuration.

- The **inventory** section tells Nornir which devices to connect to, and the **runner** section in

config.yaml tells it how many tasks to run in parallel.

- The variable **nr** represents an instance of the Nornir framework. When you call **InitNornir(config_file='config.yaml')**, Nornir reads the configuration and inventory files, and it builds an internal data structure that knows which devices exist, how to connect to them, and how many worker threads to use. You can think of **nr** as your active Nornir environment—a controller that keeps track of all managed devices and coordinates task execution.
- **nr** is the Nornir object that holds your inventory, configuration, and connection details. It's the controller that actually runs your automation tasks when you call **nr.run()**.

Step 3. Run the task: The line **result = nr.run(task=netmiko_send_command, command_string='show ip interface brief')** tells Nornir to connect to all devices in the inventory and run the same command, **show ip interface brief**, on each one. At this point, the **nr** object (the Nornir environment created earlier) already knows which devices to contact and how to reach them. The **nr.run()** function uses that information to:

1. Open an SSH connection to every device in parallel,
2. Send the **show ip interface brief** command to each one, and
3. Collect all the results into a single structure called an **AggregatedResult** object, which we will discuss in more detail later in the chapter.

You don't have to write any **for** loops, connection logic, or threading code yourself. Nornir handles all of that automatically.

Step 4. Display the output: The details of this section require a little more explanation and are explained in the next section. For now, you just need to know that this **for** loop displays the command output for each device that Nornir connected to:

```
for host, multi_result in result.items():  
  
    print(f'\n\n-----{host}-----')  
    print(multi_result[0].result)
```

After **netmiko_send_command** runs and collects results from all routers and switches, Nornir stores that information in a structured object. The loop then goes through those stored results and prints each device's name, followed by its output. In other words, this part of the program is the final step. It doesn't perform any network actions itself but presents the results from the earlier steps in a readable way.

The bottom line is that **multi_result[0].result** holds the actual command output returned by the **netmiko_send_command** task for the specific device. Each host has its own separate result, but Nornir collects all of the results together inside the **AggregatedResult** object so you can easily access every device's output in one place. (The next section provides the details of this **for** loop.)

If this feels like too much detail right now, don't worry—you can still use this code effectively. Here's the big picture of what it does, step by step:

Step 1. The program imports Nornir and Netmiko so it can connect to routers and send commands.

Step 2.The program starts Nornir by using the config file, which tells it which devices to manage and how to connect to them.

Step 3.Nornir runs the **show ip interface brief** command on all the routers at the same time.

Step 4.The program prints each device's name and the command output so you can see the results.

Understanding How Nornir Displays the Output

The final part of the program in [Example 10-1](#) displays the command output for each device. It's worth taking a little time to understand how this final loop works because it shows how Nornir organizes and returns the results of your automation tasks. When you understand this structure, you'll know exactly where your command outputs come from—and how to access, display, or process them.

Note

If the details in this section are more than you need right now, don't worry about it. Just focus on the end result of this code: It prints the results of **show ip interface brief** from **netmiko_send_command** for each host.

Once again, here is the **for** loop we are referring to:

```
for host, multi_result in result.items():  
    print(f'\n\n-----{host}-----')  
    print(multi_result[0].result)
```

When the Nornir task finishes, the variable `result` contains an **AggregatedResult** object.

What Is an AggregatedResult Object?

When Nornir runs a task, such as **netmiko_send_command**, it needs a way to store and organize the results it gets back from every device. To do that, it creates a special data structure called an **AggregatedResult** object. You can think of an **AggregatedResult** as a container (similar to a Python dictionary) that holds the results for *all the devices involved in the task*.

Each device's results are grouped under its name. Inside this top-level container, each host (for example, R1, R2, or R3) maps to another container called a **MultiResult** container. The **MultiResult** container holds one or more **Result** objects—each one representing the outcome of a specific task or subtask for that host.

In simpler terms, imagine that Nornir returns a folder called **AggregatedResult**, as shown in [Figure 10-3](#). Inside that folder, there's one subfolder for each device (for example, R1, R2, and R3). Each subfolder contains one or more files that record the output of each command or task that ran on that device.

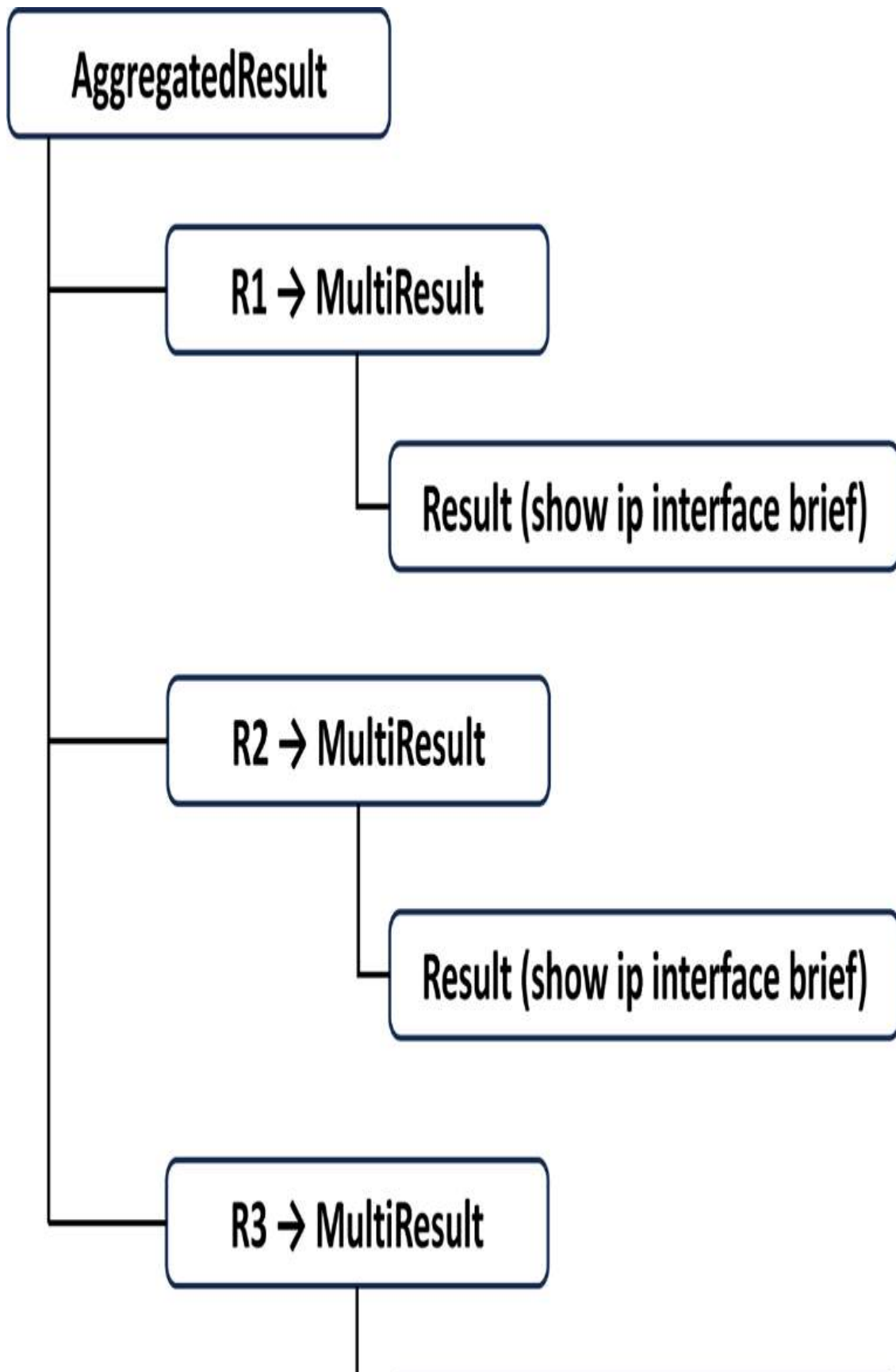


Figure 10-3 *Aggregated result structure*

Result (show in interface brief)

Nornir uses the **AggregatedResult** object because it makes it easy for you to access the results for one specific device, handle multiple results per host (if your program ran multiple tasks), and collect or analyze results across all devices at once. It's how Nornir keeps all task outputs neatly organized, even when running many commands on many devices in parallel.

Note

Keep in mind this key point: An **AggregatedResult** object is Nornir's way of packaging all the results from your task into one structured, easy-to-access container. It lets you retrieve exactly what you need—from all devices or just one—with simple Python indexing and dot notation.

Displaying the Results

After Nornir finishes running the task, the variable `result` contains an **AggregatedResult** object. This object, as explained earlier, organizes all device outputs in a dictionary-like structure, with each host mapped to its own **MultiResult** container.

The **for** loop in the code extracts and prints each device's output:

```
for host, multi_result in result.items():
    print(f'\n\n-----{host}-----')
    print(multi_result[0].result)
```

The following sections look at what happens here.

in `result.items()`

When the task completes, Nornir stores everything it collected in the variable **result**, which holds an **AggregatedResult** object:

```
result = nr.run(
    task=netmiko_send_command,
    command_string='show ip interface brief'
)
```

You can think of **result** as a container that maps each device name to its corresponding task results. If you had printed **result** in your output using **print(result)**, you would have seen the following:

```
AggregatedResult (netmiko_send_command): {'R1': MultiResult: [
    "netmiko_send_command"], 'R2': MultiResult: [Result: "netmiko_
    MultiResult: [Result: "netmiko_send_command"]}]
```

In the **for** loop, calling **.items()** on the **AggregatedResult** object (**result.items()**) works just as it does on a regular Python dictionary:

```
for host, multi_result in result.items():
    print(f'\n\n-----{host}-----')
    print(multi_result[0].result)
```

It returns a sequence of key/value pairs, where each key (such as **host**) is the device name (for example, **R1**, **R2**, **R3**) from `hosts.yaml` and each value (such as **multi_result**) is the **MultiResult** object that contains that host's task results (in this example, the output of the **show ip interface brief** command).

for host, multi_result in result.items()

This line tells Python to unpack each key/value pair as the loop runs:

```
for host, multi_result in result.items():  
    print(f'\n\n-----{host}-----')  
    print(multi_result[0].result)
```

This line includes the following parts:

- **host:** Holds the device name (R1, R2, or R3).
- **multi_result:** Holds all results for that device (in this case, the output of the **show ip interface brief** command).

Inside the loop, you can refer to these variables directly to print information about each host. For example, this loop would unpack the results as follows:

- On the first iteration, **host** is 'R1', and **multi_result** holds the command output (**show ip interface brief**) for device **R1**.
- On the second iteration, **host** is 'R2', and **multi_result** holds the output (**show ip interface brief**) for **R2**.
- On the third iteration, **host** is 'R3', and **multi_result** holds the output (**show ip interface brief**) for **R3**.

Although Nornir executes the commands on all devices in parallel, the **for** loop simply *iterates through the collected results* one device at a time—first (**'R1', MultiResult**), then (**'R2', MultiResult**), and then (**'R3', MultiResult**).

multi_result[0].result

Before looking at the individual attributes, it helps to see how the results are accessed and displayed in practice. The

following loop iterates through the results returned by Nornir and prints the command output for each device, showing how the **MultiResult** and **Result** objects are used together to retrieve the actual CLI output.

```
for host, multi_result in result.items():  
    print(f'\n\n-----{host}-----')  
    print(multi_result[0].result)
```

Each **MultiResult** object behaves like a list and contains one or more **Result** objects—one for each task run on that device. The **[0]** index selects the first (and only) task result in this case. The **.result** attribute accesses the actual command output returned by Netmiko's **send_command()** method.

Note

If a host ran multiple tasks or subtasks, you would see additional entries like **multi_result[1]** or **multi_result[2]**.

print(f'\n\n-----{host}-----')

This line simply prints a clear header showing which device the following output belongs to, making the results easier to read:

```
for host, multi_result in result.items():  
    print(f'\n\n-----{host}-----')  
    print(multi_result[0].result)
```

{host} inside the formatted string comes from the loop variable defined in **for host, multi_result in result.items():**. Its value is the device name from the top-level section key in `hosts.yaml` (for example, R1, R2, or R3). In other words, Nornir uses the names defined in your inventory as identifiers for each device, and **{host}** inserts

each name into the printed output so you can easily tell which router or switch the command results came from.

The Output

Refer to [Example 10-6](#), which displays the output from running the code in [Example 10-1](#). When the program runs, Nornir connects to all devices in the inventory, executes the **show ip interface brief** command in parallel, and then displays the collected results. Each section of the output begins with a header that shows the device name (from `hosts.yaml`), followed by the familiar IOS command output for that router.

Summary of How Nornir Displays the Results

You have seen how the results from each device are stored and accessed after the task completes. The following points outline the key parts of the code and how they relate to Nornir's result hierarchy:

- **result:** Contains all hosts and their outputs in an **AggregatedResult** object.
- **.items():** Lets the loop access each device and its results, one at a time.
- **multi_result[0].result:** Retrieves the command output text for the specific host.

This structure shows how Nornir organizes task results in a clear, hierarchical way. The **AggregatedResult** object acts as the top-level container that holds all device outputs, and each **MultiResult** container groups the results for a single device. Inside each **MultiResult** container, a **Result** object stores the actual command output returned by Netmiko. By looping through this structure, the program can easily display results from every device in the network, and you do

not have to manually manage connections or output handling. [Figure 10-4](#) uses folders to represent this concept.

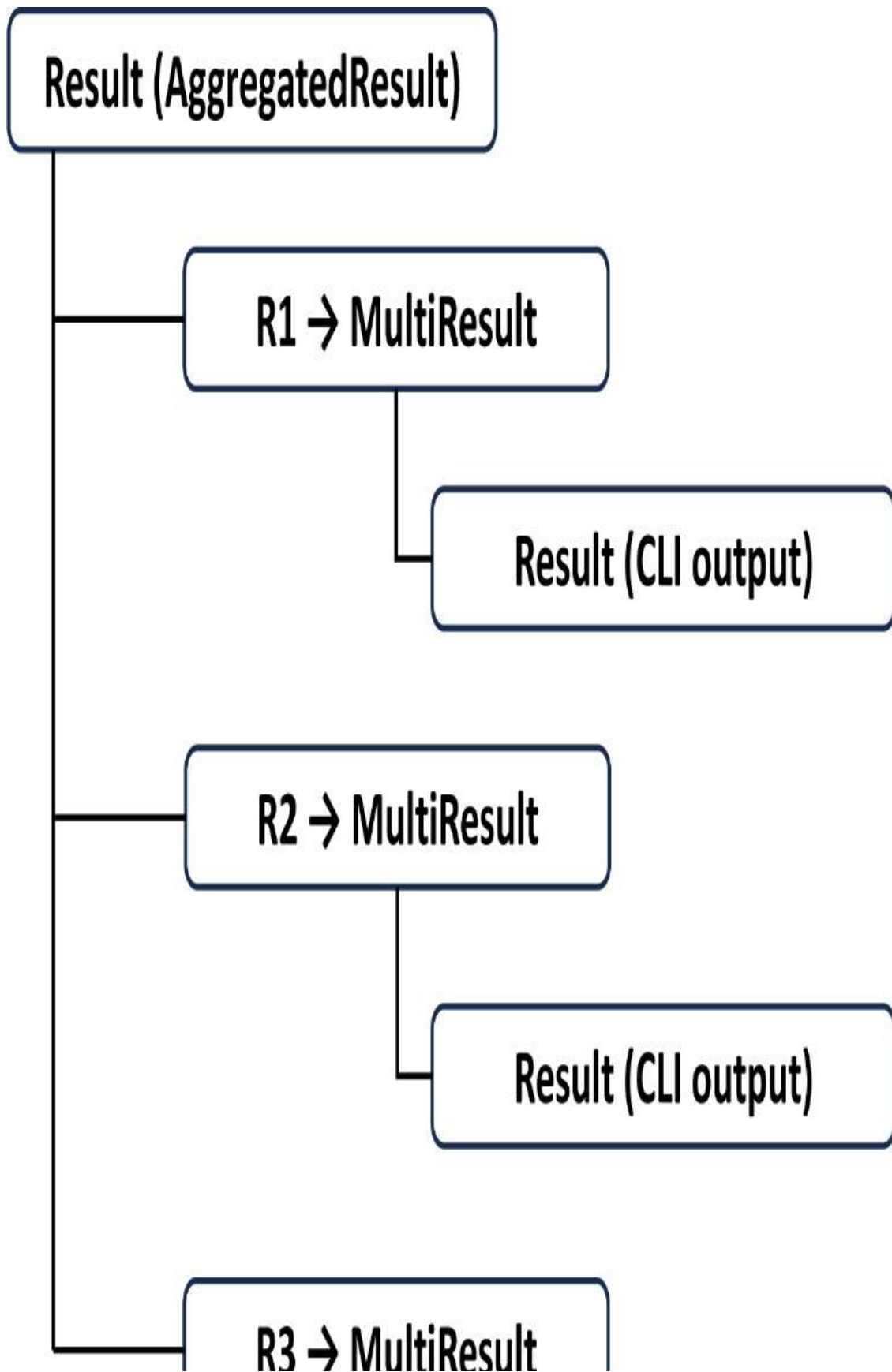


Figure 10-4 Hierarchy of Nornir results: AggregatedResult containing per-host MultiResult objects and their CLI output

It may be helpful to repeat that the bottom line is that **multi_result[0].result** holds the command output returned by the **netmiko_send_command** for the specific device. Each host has its own separate result, but Nornir collects all of the results together inside the **AggregatedResult** object so you can easily access every device's output in one place.

Result (CLI output)

Up to this point, we've focused on how Nornir, through `nornir_netmiko`, can gather information from devices by using the **netmiko_send_command** task. This task is ideal for read-only operations such as verifying interfaces, checking status, or collecting device information. In the next section, we'll move on from retrieving data to changing configurations and introduce another Nornir task: **netmiko_send_config**.

Visualizing How Nornir Executes Tasks

Now that you have seen the code in action, let's look at how Nornir and Netmiko work together behind the scenes. [Figure 10-5](#) summarizes the complete task flow—from your Python program through the plugin layer to each network device—and shows how the results are collected into the **AggregatedResult** object. The figure shows how Nornir distributes the task through the `nornir_netmiko` plugin and aggregates the results into its internal data structure.

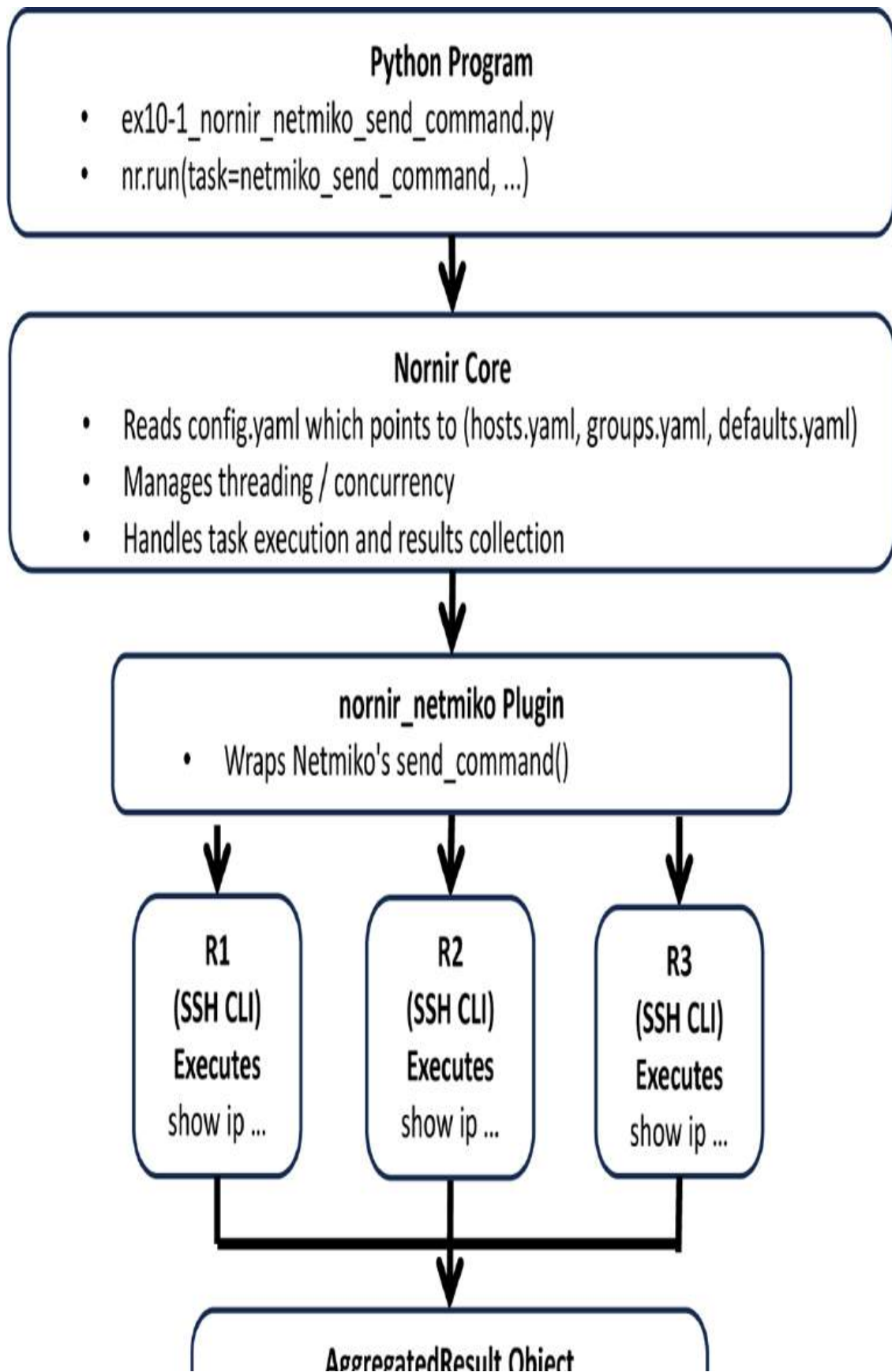


Figure 10-5 Task flow: Nornir → Netmiko → Network devices with parallel execution

Figure 10-5 shows the sequence of interactions that occur when you run a Nornir program using the `nornir_netmiko` plugin. At the top, the Python program (such as **ex10-1_nornir_netmiko_send_command.py**) calls Nornir's `run()` method to start the task—in this case, **netmiko_send_command**.

The Nornir core is the central engine of the Nornir framework that handles inventory management, threading, and task execution. It coordinates how tasks are distributed to devices, loads configuration and inventory data, and aggregates the results returned by each plugin. Once the task is triggered, the Nornir core first reads the `config.yaml` file, which specifies the inventory sources (`hosts.yaml`, `groups.yaml`, and `defaults.yaml`). It then loads these files to determine which devices to connect to and manages threading and concurrency so multiple devices can be contacted simultaneously.

Nornir then passes the task to the `nornir_netmiko` plugin, which acts as a bridge between Nornir and Netmiko. This plugin wraps Netmiko's `send_command()` method, establishes the SSH connections to each device in the inventory, and executes the specified command (for example, **show ip interface brief**).

Each device performs the command independently, returning its output to Nornir. These results are then gathered and organized into an **AggregatedResult** object, which maps each host (such as R1, R2, and R3) to a corresponding **MultiResult** container. Each **MultiResult** container holds one or more **Result** objects—each representing the output from a specific command (in this

case, **show ip interface brief**) or subtask. It is worth showing the relationships again in [Figure 10-6](#).

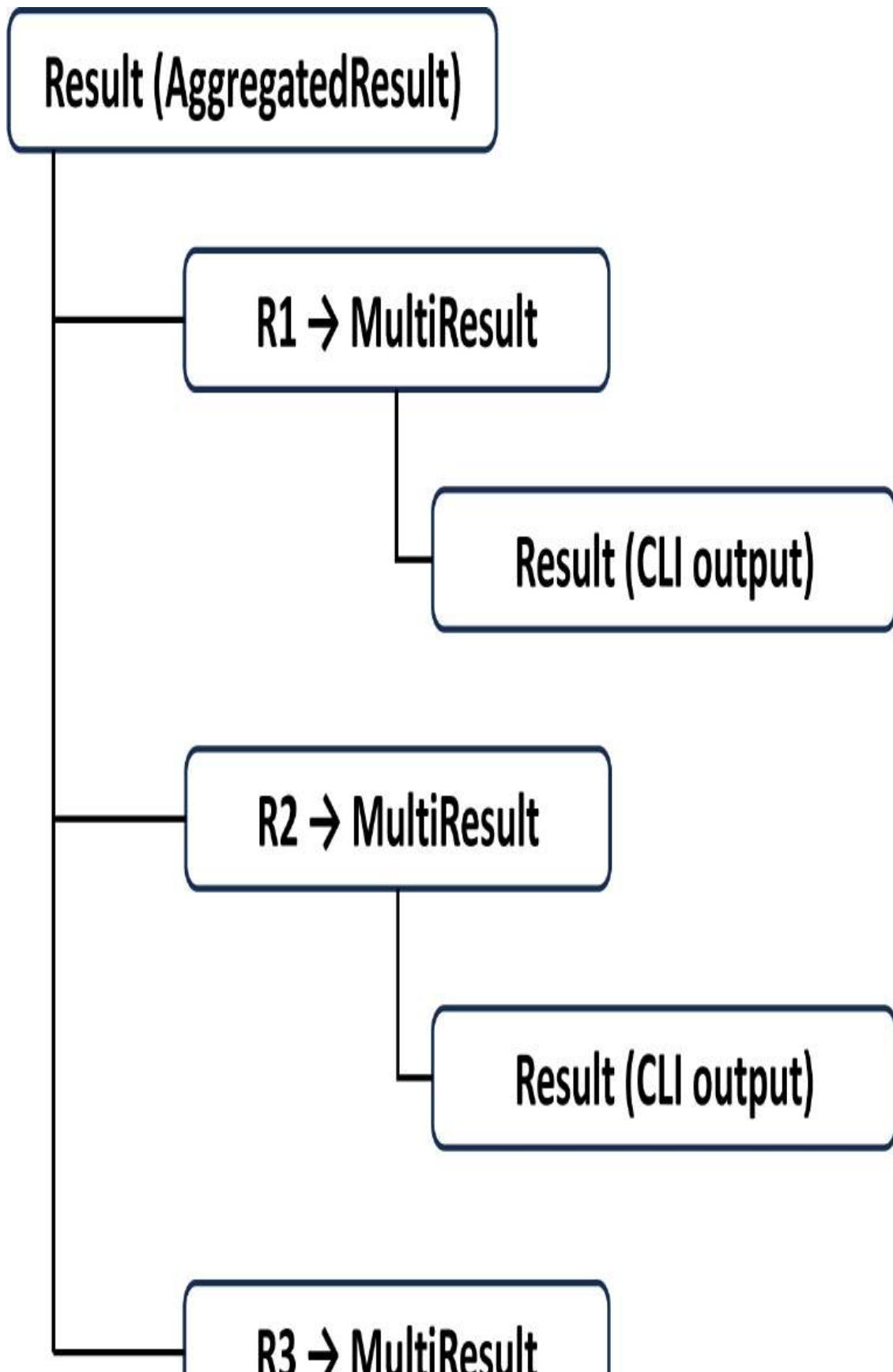


Figure 10-6 *Hierarchy of Nornir results: AggregatedResult containing per-host MultiResult objects and their CLI output*

In short, [Figure 10-5](#) shows how Nornir orchestrates the process: Your Python program defines the task, Nornir manages execution and threading, the `nornir_netmiko` plugin handles communication, and the **AggregatedResult** object neatly stores the results for easy retrieval and display.

You've already seen how the results are organized in the **AggregatedResult** → **MultiResult** → **Result** hierarchy. Each level represents a different scope—all hosts, each host, and each task, respectively—allowing Nornir to keep device outputs structured and easy to access.

If we revisit [Example 10-1](#), we are reminded that the flow looks like this:

- Step 1.Import and initialize:** Nornir loads its configuration and inventory files.
- Step 2.Run the task: `netmiko_send_command`** executes the specified CLI command (**show ip interface brief**) on every device concurrently.
- Step 3.Collect the results:** Each device's command output is stored in a **Result** object, grouped into its device's **MultiResult** container, and combined into a single **AggregatedResult** Object.
- Step 4.Display:** The **for** loop prints the output for each host, producing the console output shown in [Example 10-2](#).

Up to this point, you've seen how Nornir, through the `nornir_netmiko` plugin, can connect to multiple devices and collect operational information using

netmiko_send_command. This task is ideal for read-only operations, such as verifying interfaces, checking configurations, or gathering device facts.

In the next section, we'll move from retrieving information to making configuration changes. You'll see how Nornir uses a similar structure—still relying on the same inventory, task execution, and result-handling model—but now with the **netmiko_send_config** task, which enables you to apply configuration commands across multiple devices in parallel.

Sending Configuration Commands with **netmiko_send_config**

So far, we've used **netmiko_send_command** for read-only operations—to gather information from devices without making changes. But what if you need to configure devices instead of just viewing their status? That's where **netmiko_send_config** comes in. This task uses Netmiko's **send_config_set()** method under the hood, which automatically places a device into configuration mode, applies one or more commands, and then exits back to EXEC mode. You may recall from [Chapter 3, “Configuring Devices with Netmiko,”](#) that **send_config_set()** differs from **send_command()** in that it's designed specifically for configuration commands rather than **show** commands. [Table 10-1](#) provides a comparison between **netmiko_send_command** and **netmiko_send_config**.

Table 10-1 Comparing **netmiko_send_command** and **netmiko_send_config** Tasks

Task	Purpose	Device Mode	Input Type	Typical Use
<code>netmiko_send_command</code>	Runs operational (show) commands	EXEC mode	String (single command)	Gathering device information (for example, show ip interface brief)
<code>netmiko_send_config</code>	Sends configuration commands	CONFIG mode	List or string	Applying configuration changes (for example, interface or routing settings)

Using `netmiko_send_config` with a Single Shared List of Commands

Now that you've seen how to use **`netmiko_send_command`** to collect information from devices, let's look at how to configure devices using **`netmiko_send_config`**.

The **`netmiko_send_config`** task, which is part of the `nornir_netmiko` library, uses Netmiko's **`send_config_set()`** method under the hood. This method automatically enters configuration mode on each device, applies all the commands you specify, and then exits configuration mode when finished—just you would do if you were typing the commands manually.

[Example 10-7](#) configures IPv6 routing on all devices in the Nornir inventory by enabling **`ipv6 unicast-routing`** and **`ipv6 cef`**.

Example 10-7 *`ex10-7_nornir_netmiko_send_config.py`*

```
# 1. Import required modules
from nornir import InitNornir
from nornir_netmiko.tasks import netmiko_send_config

# 2. Initialize Nornir with the configuration file
nr = InitNornir(config_file='config.yaml')

# 3. Define the configuration commands
commands_list = [
    'ipv6 unicast-routing',
    'ipv6 cef'
]

# 4. Run the task across all hosts
result = nr.run(task=netmiko_send_config, config_commands=commands_list)

# 5. Extract the output of the configuration command for each device
for host, multi_result in result.items():
    print(f'\n\n-----{host}-----')
    print(multi_result[0].result)
```

Note

On platforms that use distributed forwarding architectures, IPv6 Cisco Express Forwarding (CEF) is enabled using the **ipv6 cef distributed** command. This command serves the same purpose as **ipv6 cef** on older, centralized platforms. On these newer platforms, IPv6 CEF is always distributed, so **ipv6 cef** without the **distributed** keyword is not available or required.

How It Works

If you look closely, the program in [Example 10-7](#) follows the same overall structure as the one in [Example 10-1](#), which uses **netmiko_send_command**. The first two steps are identical: Import Nornir and its plugin and then initialize Nornir with the config.yaml file to load the inventory and runner settings.

The main differences begin in step 3 and step 4:

- In step 3, instead of using a single **command_string**, the program defines a Python list named **commands_list**. This list contains all the configuration lines you want applied, with each item representing one CLI command, just as you would type the command manually.
- In step 4, **nr.run()** calls **netmiko_send_config** instead of **netmiko_send_command**. This version of the task automatically enters configuration mode on each device, runs all the commands in **commands_list**, exits configuration mode, and collects the results.

Because Nornir uses a threaded runner by default, multiple devices are configured in parallel—up to the number of worker threads defined in config.yaml.

Finally, the loop that prints the results works exactly as before: It iterates through the **AggregatedResult** object and displays each device's configuration feedback.

In short, the workflow is the same as before. Only the task and its input change, switching from read-only operations to configuration mode.

Changes to defaults.yaml

Because the **netmiko_send_config** task makes configuration changes on the device (unlike read-only tasks such as **netmiko_send_command**), you need to provide Netmiko with additional connection details, as shown in [Example 10-8](#). These settings are added to the defaults.yaml file so that Nornir knows how to enter privileged EXEC mode and reliably send configuration commands to each router.

Note

No changes were needed to the config.yaml, hosts.yaml, and groups.yaml files.

Example 10-8 Connection Options for Netmiko in defaults.yaml

```
username: admin
password: cisco
connection_options:
  netmiko:
    extras:
      secret: spot          # <-- enable/privileged password
      fast_cli: false       # optional; can make config pushes r
```

These settings give Netmiko additional information about how to connect and send configuration commands to the devices:

- **connection_options:** Defines connection-specific settings for a plugin—in this case, netmiko. These options tell Nornir how to connect to the devices beyond just username and password.

- **netmiko:** Specifies that the connection type is for the Netmiko plugin, which handles SSH sessions to network devices.
- **extras:** Passes additional parameters directly to Netmiko's connection handler. These options fine-tune how Netmiko behaves when interacting with the device.
- **secret: spot:** Sets the **enable** (privileged EXEC) password that Netmiko uses when entering enable (privileged EXEC) mode. Without it, configuration tasks like **netmiko_send_config** may fail if the device requires enable access.
- **fast_cli: false:** Adds slight pauses between commands, making configuration pushes more reliable, especially on slower devices or when running multiple commands. (By default, Netmiko tries to send commands quickly to speed up execution. The false setting adds the slight pause.)

As you can see from the output in [Example 10-9](#), Nornir connects to all three routers using Netmiko and applies the same configuration commands to each device.

Example 10-9 *Output of [Example 10-7](#)*

```
-----R1-----
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
Router-R1(config)#ipv6 unicast-routing
Router-R1(config)#ipv6 cef
Router-R1(config)#end
Router-R1#

-----R2-----
configure terminal
```

```
Enter configuration commands, one per line.  End with CNTL/Z.
Router-R2(config)#ipv6 unicast-routing
Router-R2(config)#ipv6 cef
Router-R2(config)#end
Router-R2#

-----R3-----
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
Router-R3(config)#ipv6 unicast-routing
Router-R3(config)#ipv6 cef
Router-R3(config)#end
Router-R3#
```

This approach works well for small common configuration requirements that you want to deploy identically across multiple devices. However, in real networks, configurations often vary between devices—different IP addresses, interface descriptions, or routing parameters.

The method used in [Example 10-7](#)—passing a single shared list of commands—doesn't easily scale when each device needs unique values. To handle that, you can store device-specific configuration data directly in the `hosts.yaml` file and have Nornir read and apply those settings automatically.

To make configurations more flexible, Nornir lets you associate unique command sets or variables with each device directly in your inventory files. Instead of using a single shared list like **commands_list**, you can define device-specific configuration lines under each host's data section in `hosts.yaml`. This approach allows you to push individualized settings—such as interface IP addresses, descriptions, or loopback numbers—while still running one centralized automation program.

In the next section, you'll see how to update the inventory to include per-device configuration data and modify the Python program so that each router applies its own customized commands automatically.

Using netmiko_send_config with hosts.yaml for per-Device Configuration

In [Example 10-7](#), every router receives the same configuration commands. That's fine for small, uniform updates, but in most real-world cases, each device needs its own unique configuration. A better way to handle this is by defining device-specific configuration commands directly in the Nornir hosts.yaml file. You do this by adding a new key (for example, **config_lines**) under each host's **data** section, as shown in [Example 10-10](#).

Example 10-10 Updated hosts.yaml File

```
R1:
  hostname: 192.168.1.1
  groups: [ios]
  data:
    site: 'HQ'
    role: 'edge'
    config_lines:
      - interface loopback0
      - description Loopback interface - used nornir_netmiko
      - ip address 1.1.1.1 255.255.255.255
      - no shutdown

R2:
  hostname: 192.168.2.2
```

```
groups: [ios]
data:
  site: 'Distribution'
  role: 'core'
  config_lines:
    - interface loopback0
    - description Loopback interface - used nornir_netmiko
    - ip address 2.2.2.2 255.255.255.255
    - no shutdown

R3:
  hostname: 192.168.3.2
  groups: [ios]
  data:
    site: 'Branch'
    role: 'edge'
    config_lines:
      - interface loopback0
      - description Loopback interface - used nornir_netmiko
      - ip address 3.3.3.3 255.255.255.255
      - no shutdown
```

Note

The groups.yaml and defaults.yaml files remain unchanged.

In this hosts.yaml file, a new key named **config_lines** has been added under each device's **data** section. The **config_lines** key is a list that contains the configuration commands that Nornir will send to that specific router. Each device now has its own **config_lines** list that contains the commands Nornir will send to that router.

In YAML, **config_lines** is the key, and its value is a list of strings, with a dash (-) indicating each list item. Each item in the list is a separate configuration command. You write these commands exactly as you would on the CLI—one per line. By storing them in hosts.yaml, each device maintains its own configuration data in a clean, organized way.

Note

In YAML, quotation marks around strings are optional. You can use either single quotes or double quotes, and you can add them if you prefer or when a command contains special characters (such as **:** or **#**) that YAML might otherwise misinterpret. Notice that the IP addresses in [Example 10-10](#) are not in quotes although they are strings. When to use quotation marks is usually a style option.

The approach using **config_lines** has two big advantages:

- You can keep all configuration data in one central file alongside your inventory.
- Each device can have its own customized configuration, but you can still deploy all of the configurations with a single Python program.

Now that the configuration commands are part of the inventory, you need a way for the Python program to read and apply them. You will see how to do this by creating a small custom task function that retrieves the list of commands from each host and passes them to the **netmiko_send_config** task. Inside the program, Nornir provides access to all variables defined for each device through a special object called **task_object.host**. You can retrieve any value from hosts.yaml—for example, **task_object.host['hostname']** or **task_object.host['config_lines']**—to use directly in your

code. [Example 10-11](#) shows how this function works and how Nornir applies the configuration lines from each device's inventory entry.

Example 10-11 *ex10-11nornir_netmiko_send_config_host.py*

```
# 1. Import required modules
from nornir import InitNornir
from nornir_netmiko.tasks import netmiko_send_config

# 2. Initialize Nornir with the configuration file
nr = InitNornir(config_file='config.yaml')

# 3. Define the task
def send_config(task_object):
    # Get device-specific config from host data
    config = task_object.host['config_lines']
    task_object.run(task=netmiko_send_config, config_commands=config)

# 4. Run the task across all hosts
result = nr.run(task=send_config)

# 5. Display the configuration results for each device
for host, multi_result in result.items():
    print(f'\n\n-----{host}-----')
    print(multi_result[1].result)
```

How It Works

The code in [Example 10-11](#) expands on the earlier program by introducing device-specific configuration stored in `hosts.yaml` and a custom task function that retrieves and applies it automatically. These are the steps:

Step 1.Import required modules: As in previous examples, the program imports **Nornir** and the **netmiko_send_config** task. These modules provide the core automation framework and the configuration-sending capability.

Step 2.Initialize Nornir: The **InitNornir(config_file='config.yaml')** line loads the inventory files (hosts.yaml, groups.yaml, and defaults.yaml) and prepares the environment. This is the same as before, but now the inventory includes per-device configuration lines under the key **config_lines**.

Step 3.Define the custom task: The new **send_config(task_object)** function is what makes this example different. **task_object** is simply the name of the parameter—a placeholder that represents the Nornir task object that will be passed in at runtime. This function uses **task_object.host['config_lines']** to retrieve the list of commands defined for each individual device in hosts.yaml, as shown in [Figure 10-7](#). It then calls **task_object.run(task=netmiko_send_config, config_commands=config)** to send those commands to the device. This allows the same Python program to push unique configurations to each host automatically. The following section, [“Understanding the User-Defined Function,”](#) describes this function.

Step 4.Run the task. The line **result = nr.run(task=send_config)** executes the custom task on all devices in parallel. Instead of running a built-in Nornir task directly (as in earlier examples), it runs the user-defined function,

which in turn calls the built-in **netmiko_send_config**.

Step 5. Display the results. The final **for** loop prints each device's configuration output. Because this program uses a custom task that calls another task inside it, the Netmiko output resides in **multi_result[1].result** (the second result entry).

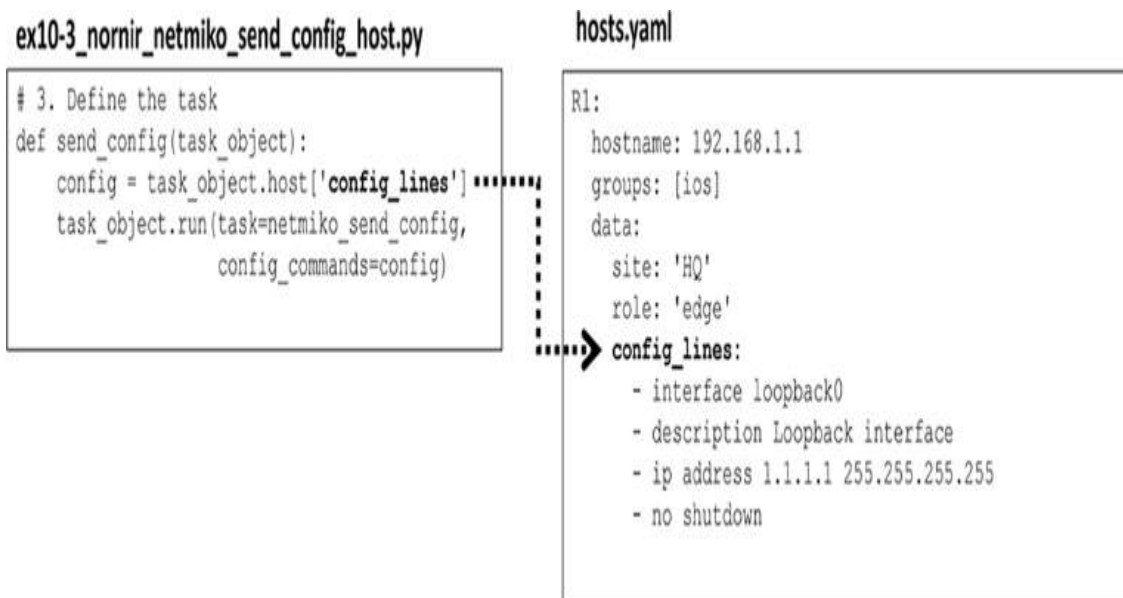


Figure 10-7 Retrieving a list of commands from *hosts.yaml*

Notice that you use **print(multi_result[1].result)**, with an index of **[1]** instead of an index of **[0]** (as in [Example 10-3](#)). The bottom line is that you use **[0]** when your program runs a single built-in task like **netmiko_send_command**. You use **[1]** when you've created a custom task function that calls another task inside it. In this case, **[0]** is the wrapper (your function) and **[1]** is the actual subtask result you want to display.

Note

Because this program defines a custom task function (**send_config**), the results returned by Nornir include both the parent task and its subtask. **Index [0]** refers to the parent task (**send_config** itself), and **index [1]** holds the output from **netmiko_send_config**. That's why the program prints **multi_result[1].result**: It displays the command output from the subtask that performed the configuration.

Understanding the User-Defined Function

This section is for those who may not be familiar with user-defined functions.

In Python, you can define your own function to perform a specific task. A function definition starts with the keyword **def**, followed by the function's name and parentheses.

[Example 10-12](#) defines a function called **greet()**.

Example 10-12 *Defining and Calling a User-Defined Function*

```
# Define the function
def greet(name):      # 'name' is the parameter
    print(f"Hello, {name}!")

# Call the function
greet("Alice")        # "Alice" is the argument
```

A *parameter* is a name (variable) you give a piece of data that a function expects to receive. The parameter *name* is the placeholder inside the function **greet()**. It acts as a placeholder for information that the function needs to do its job.

An *argument* is the value or data that gets passed to that parameter when the function runs. For example, when you call **greet("Alice")**, the argument **"Alice"** is passed into the parameter **name**.

The function then displays the output:

```
Hello, Alice!
```

In other words:

- Parameter = placeholder
- Argument = actual value provided

In [Example 10-11](#), step 3 defines a custom, or user-defined, function named **send_config()**:

```
def send_config(task_object):  
    config = task_object.host['config_lines']  
    task_object.run(task=netmiko_send_config, config_commands=config)
```

In this example, **task_object** (in the parentheses after **def send_config**) is the **parameter**. It's a placeholder name that tells Python, "This function expects one piece of information when it's called."

Within the function:

- **Task_object.host['config_lines']** retrieves the configuration commands for that specific device from hosts.yaml.
- **Task_object.run(task=netmiko_send_config, config_commands=config)** runs the **netmiko_send_config** task using those commands.

The function call is in step 4:

```
result = nr.run(task=send_config)
```

Nornir automatically calls your function once per device and passes in a **Task** object that represents that device. This **Task** object becomes the argument for the function's **task_object** parameter. So, in this case, **task_object** refers *both* to the parameter (the name used in the function definition) and to the argument (the **Task** object) that Nornir passes in at runtime.

This might seem a bit confusing, but the details are not as important as understanding what the function is accomplishing. The result of all of this is that you can access any key from the hosts.yaml file directly. For example, if you were to use the command **task_object.host['hostname']**, this would return the hostname defined for that device (for example, **192.168.1.1** for R1).

[Example 10-11](#) uses **task_object.host['config_lines']**, where the key **config_lines** retrieves a list of configuration commands for that specific device from the hosts.yaml file. Then **task_object.run(task=netmiko_send_config, config_commands=config)** is used to send those commands to the device.

The bottom line is that this user-defined function lets you tell Nornir exactly what to do for each device. The parameter **task_object** gives your function access to all the device-specific data (like **hostname** or **config_lines**) and the ability to call other Nornir tasks within it.

[Example 10-12](#) shows output from the code in [Example 10-11](#), which confirms that **Loopback0** was configured correctly on all three devices, with the desired IPv4 addresses.

Example 10-12 *Output from [Example 10-11](#)*

```
MyPrompt% python3 ex10-11_nornir_netmiko_send_config_host.py
```

```
-----r1-----
```

```
configure terminal
```

```
Enter configuration commands, one per line. End with CNTL/Z.
```

```
Router-R1(config)#interface loopback0
```

```
Router-R1(config-if)#description Loopback interface - used norn:
```

```
Router-R1(config-if)#ip address 1.1.1.1 255.255.255.255
```

```
Router-R1(config-if)#no shutdown
```

```
Router-R1(config-if)#end
```

```
Router-R1#
```

```
-----r2-----
```

```
configure terminal
```

```
Enter configuration commands, one per line. End with CNTL/Z.
```

```
Router-R2(config)#interface loopback0
```

```
Router-R2(config-if)#description Loopback interface - used norn:
```

```
Router-R2(config-if)#ip address 1.2.2.2 255.255.255.255
```

```
Router-R2(config-if)#no shutdown
```

```
Router-R2(config-if)#end
```

```
Router-R2#
```

```
-----r3-----
```

```
configure terminal
```

```
Enter configuration commands, one per line. End with CNTL/Z.
```

```
Router-R3(config)#interface loopback0
```

```
Router-R3(config-if)#description Loopback interface - used norn:
```

```
Router-R3(config-if)#ip address 3.3.3.3 255.255.255.255
```

```
Router-R3(config-if)#no shutdown
```

```
Router-R3(config-if)#end
```

```
Router-R3#
```

MyPrompt%

Remember that Nornir handles the work in *parallel*; that is, it connects to all devices at the same time and applies the configurations simultaneously. Nornir's threaded runner manages all of that automatically. This ability to configure many devices at once is one of Nornir's biggest advantages over running commands sequentially.

In this section, you have seen how Nornir can combine data stored in `hosts.yaml` with custom logic in Python to apply different configurations to multiple devices at once.

Summary

In this chapter, you learned how to use Nornir together with Netmiko to automate network operations across multiple devices in parallel. Starting from basic tasks and progressing toward more advanced use cases, this chapter has built a strong foundation for network automation workflows.

You began by running **show** commands using the **netmiko_send_command** task, which executes IOS commands across many devices simultaneously and returns the output in Nornir's result hierarchy: **AggregatedResult** → **MultiResult** → **Result**.

You then sent configuration commands using **netmiko_send_config** and learned how to push common configuration snippets to multiple routers or switches.

Next, you learned to handle device-specific configurations by storing per-host data in `hosts.yaml` under **config_lines** and using a custom Python function to apply those

commands automatically. You have seen how Nornir combines structured inventory data with Python logic to deliver targeted configurations efficiently.

Nornir's integration with Netmiko illustrates how easily Python can orchestrate real-world network changes across many devices.

Chapter 11. Using Nornir with NAPALM

In [Chapter 10, “Using Nornir with Netmiko,”](#) you learned about using Nornir with Netmiko to automate multiple devices at once. You saw how to use Netmiko to handle the SSH connection to each device (router) and Nornir to scale those connections across the entire inventory. With this combination, a simple Python program can execute commands or push configuration to many devices at the same time. This chapter takes the next natural step and looks at using Nornir with NAPALM.

A Quick Review

Before we begin discussing how Nornir can enhance the automation abilities of NAPALM, let's have a quick review of both NAPALM and Nornir.

A Quick Review of NAPALM

Earlier in the book, you saw that NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is a Python library that provides a consistent way to interact with network devices from different vendors. Instead of writing separate code for Cisco, Juniper, Arista, or other platforms, you can use NAPALM,

which offers a common interface that works across many operating systems, including Cisco IOS and IOS XE, Juniper Junos, and Arista EOS. With NAPALM, you can use the same Python code to retrieve structured information from a device—such as basic system details, interface status, or routing information—and to apply configuration changes in a safe and predictable way. It eliminates the need for vendor-specific logic and makes automation programs easier to write, read, and maintain.

When you use Nornir and NAPALM together, each tool plays a specific role:

- Nornir manages the inventory, credentials, grouping, and parallel execution.
- NAPALM handles interacting with the devices (collecting data, loading configurations, showing diffs, and committing or rolling back changes).

The result is a clean workflow for multivendor automation with a structure that is familiar from earlier chapters.

A Quick Review of Nornir

Nornir is a Python-based automation framework that is designed to run tasks across multiple devices. You have already worked with Nornir in previous chapters, where you loaded devices from `hosts.yaml`, `groups.yaml`, and `defaults.yaml`. Nornir uses the inventory in these files to determine which devices to connect to, what platform each device uses, which username/password to use, and what groups or roles each device belongs to.

Nornir tasks are simply Python functions. When you call **`nr.run(task=...)`**, Nornir executes that function on all the devices in the inventory—often in parallel.

These are a few of Nornir's core components that we discuss in this chapter:

- **Inventory:** This is where Nornir stores information about each device, such as its hostname or IP address, platform, and login credentials. The inventory is usually defined in the familiar `hosts.yaml`, `groups.yaml`, and `defaults.yaml` files you used in [Chapter 10](#). Nornir uses this information to determine how to connect to each device and how the devices should be grouped or filtered.
- **Tasks:** A task is simply a Python function that defines what action you want to perform on each device. When you call `nr.run(task=...)`, Nornir executes that function for every device in the inventory. Tasks allow you to structure your automation programs in a clear and reusable way.
- **Runners:** As you saw in [Chapter 10](#), a runner controls how Nornir executes tasks. The threaded runner, for example, allows multiple devices to be processed at the same time, which makes automation much faster than working device by device.
- **Plugins:** Plugins extend Nornir's functionality and allow it to integrate with external libraries. For example, the `nornir_netmiko` plugin lets Nornir use Netmiko for SSH access, and in this chapter you will use the `nornir_napalm` plugin so Nornir can use NAPALM to gather information and apply configuration changes. Plugins make Nornir flexible and allow it to work with many different network automation tools.

Nornir and NAPALM: Why Combine Them?

By integrating NAPALM into Nornir, you gain the strengths of both tools working together. Nornir manages the device inventory, grouping, connection details, and parallel execution, whereas NAPALM provides a structured, vendor-neutral way to retrieve information from devices and apply configuration changes. The `nornir_napalm` plugin connects these two capabilities, allowing Nornir to run NAPALM operations across the entire inventory in a seamless and consistent workflow. This combination makes it possible to automate many network tasks with short, readable Python programs.

Working together, Nornir and NAPALM allow you to:

- Run **show**-style commands across multiple devices at the same time
- Collect structured data in a consistent, vendor-neutral format
- Apply configuration snippets safely and predictably
- View a diff to see exactly what will change before you commit the changes
- Commit or roll back changes, giving you a safe way to test configurations

This chapter presents programs that follow the same Nornir patterns you learned previously, but now using NAPALM in place of Netmiko.

Installing Support for NAPALM: `nornir_napalm`

To use NAPALM inside Nornir, you can install the `nornir_napalm` plugin. The installation requires a single **pip** command:

```
pip install nornir_napalm
```

Note

The examples in this chapter were tested using Python 3.10, and the version of the plugin at the time of writing is 0.5.0. When you install it on your own system, make sure you are using a Python version that is supported by the plugin.

Once the plugin is installed, it allows Nornir to make NAPALM calls as part of normal Nornir tasks. Nornir still handles the inventory, grouping, threading, and execution order—just as you saw in [Chapter 10](#). In this chapter, however, instead of using Netmiko for device access, Nornir uses the NAPALM connection plugin, which loads the correct vendor driver, based on the device’s platform in your inventory.

When a Nornir program uses `nornir_napalm`, the workflow looks like this:

Step 1. Initialize Nornir: Nornir loads the inventory from the same `hosts.yaml`, `groups.yaml`, and `defaults.yaml` files you used previously.

Step 2. Define a task: A Python task function is defined, and inside that task, a `nornir_napalm` task (such as **`napalm_cli`** or **`napalm_configure`**) is called.

Step 3. Run the task: Nornir connects to each device by using the NAPALM driver for its platform (for example, IOS, IOS XE, Junos, EOS). The NAPALM

action runs (for example, gathering facts, running CLI commands, loading the configuration).

Step 4.Process the results: Results are returned to Nornir as structured Python dictionaries, which Nornir collects and prints or processes.

This workflow allows you to write short, readable programs that gather data or apply configuration changes across multiple devices in parallel, all while keeping the code consistent and vendor neutral.

nornir_napalm Tasks

Once the `nornir_napalm` plugin is installed, Nornir can use a variety of NAPALM-based tasks to interact with network devices. These tasks allow you to gather information, run commands, load configuration changes, validate device state, and safely roll back changes if needed. Each task uses the appropriate NAPALM driver for the device's platform, so the same Python code works across multiple vendors.

The `nornir_napalm` plugin provides a set of predefined tasks that map directly to NAPALM's core capabilities. [Table 11-1](#) summarizes these tasks, the primary purpose of each task, and the NAPALM methods each task invokes behind the scenes.

Table 11-1 *Nornir NAPALM Tasks*

Task Name	Primary Purpose	Common Use Cases	Underlying NAPALM Method(s)
napalm_cli	Run device CLI commands	Run show commands across many devices and collect raw command output	device.cli()
napalm_configure	Apply configuration changes	Deploy configuration updates, preview diffs, and commit or discard changes	load_merge_candidate() , load_replace_candidate() , compare_config() , commit_config() , discard_config()
napalm_get	Retrieve structured device data	Inventory collection, documentation, validation, and monitoring	get_facts() , get_interfaces() , get_arp_table() , get_bgp_neighbors() , get_route_to()
napalm_ping	Test network reachability	Verify connectivity from the device's perspective	ping()
napalm_validate	Validate state against a baseline	Compliance checks, audits, and post-change verification	validate()
napalm_rollback	Restore previous configuration	Recover from failed or incorrect changes	rollback()
napalm_confirm_commit	Confirm timed configuration commits	Prevent automatic rollback after a commit-confirm operation	confirm_commit()

Advantages of Using These Tasks with Nornir

Using NAPALM alone provides powerful, vendor-neutral device interaction. Adding Nornir on top offers several important advantages:

- You can execute tasks across many devices in parallel.
- You can use Nornir's inventory system to target specific groups (for example, "all routers," "all Cisco devices," or "all devices in Building A").
- You can collect and process results in a single unified data structure.
- You can easily combine NAPALM tasks with other automation tasks—including Netmiko tasks—in the same program.
- You can standardize configuration workflows across vendors and device types.
- You can build repeatable automation programs, using a consistent Python-based framework.

Together, these tasks allow Nornir to use NAPALM's capabilities across multiple devices in a clean, unified way. The following sections explore how to use these tasks through hands-on examples, starting with gathering structured information from devices and then progressing to configuration management and validation.

Using Nornir and the `napalm_cli` Task

Before we dive into the details of the **`napalm_cli`** task, it's important to point out that the programs in this chapter look slightly more complex than the ones you wrote in the earlier Netmiko and NAPALM chapters. This is expected. Nornir

introduces more structure, and `nornir_napalm` adds another layer on top of that. However, you do *not* need to memorize every line of code or understand every detail the first time you read this chapter. What matters most is understanding what a program does and how to use this framework to your advantage. As you progress through the examples in this chapter, you will recognize familiar concepts, and you will begin to see how Nornir and NAPALM combine to automate tasks across many devices at once. The goals are clarity and confidence, not complexity.

So, let's explore a Nornir and NAPALM example that uses the **`napalm_cli`** task. This task lets you run raw CLI commands on network devices using the NAPALM drivers, which makes it especially helpful for vendor-specific commands that are not supported by NAPALM's structured data-collection methods. Even better, Nornir allows you to run these commands on many devices at the same time. Let's start with two familiar IOS commands—**`show version`** and **`show ip interface brief`**—and run them across all devices in the inventory.

[Figure 11-1](#) shows the same three-router topology we have been using throughout this book. In [Chapter 10](#), you created applications that pushed commands to these routers by using Netmiko's SSH connection. That provided a solid foundation for understanding Nornir's inventory system and task execution model.

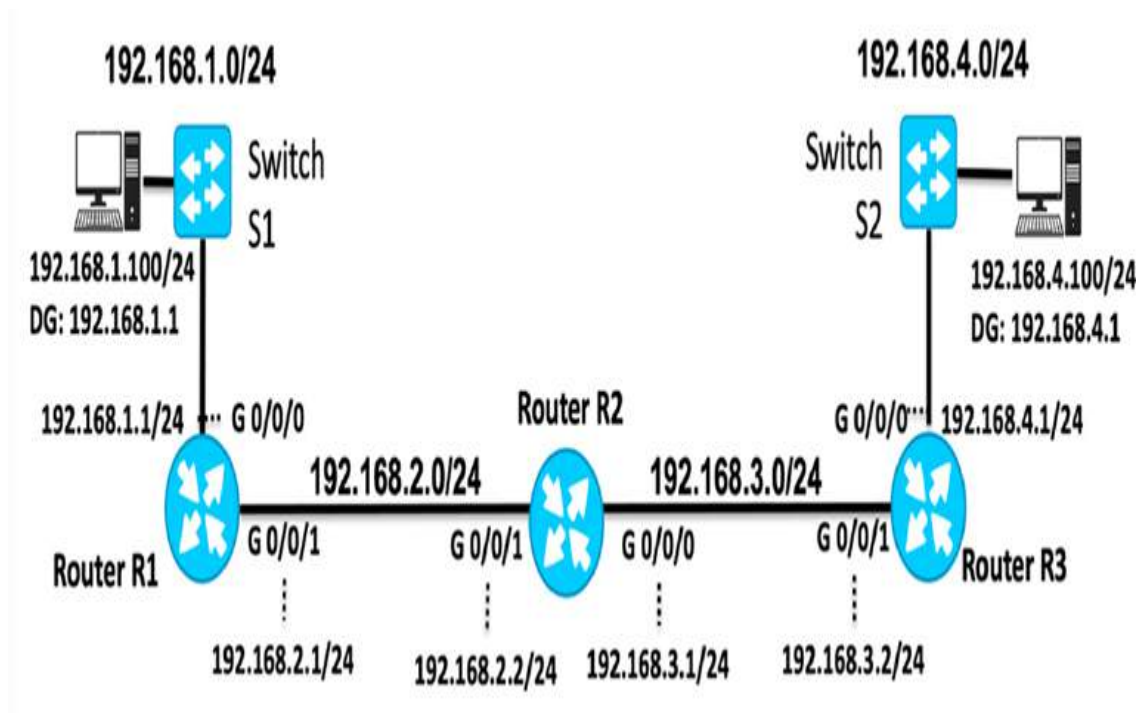


Figure 11-1 *Three-router topology used throughout this book*

Now you can take the next step and run CLI commands across multiple devices using NAPALM's vendor-neutral drivers. This approach gives you the best of both worlds: the flexibility of sending CLI commands and the consistency and structure offered by NAPALM.

To begin, you'll use common **show** commands to gather two pieces of basic operational data from each router:

- **show version:** This command verifies the IOS XE version running on each device.
- **show ip interface brief:** This command confirms the interface status and IP addressing.

napalm_cli Task Example

The program in [Example 11-1](#) demonstrates how you can use the **napalm_cli** task to send **show** commands to every

router at once and return the results in an organized structure. The next section goes through this example line by line to help you fully understand how the program works and how to build your own.

Example 11-1 *ex11-1_nornir_napalm_cli.py*

```
# 1. Import the required modules
from nornir import InitNornir
from nornir_napalm.plugins.tasks import napalm_cli

# 2. Define the task
def run_cli_commands(task):

    commands_to_run = [
        "show version",
        "show ip interface brief"
    ]

# 3. Execute the list of CLI commands on the device using napalm
    result = task.run(task=napalm_cli, commands=commands_to_run)

# 4. Return the result of napalm_cli which is a dictionary where
    values are their outputs.

    return result[0].result # Access the actual dictionary of co

if __name__ == "__main__":

# 5. Initialize Nornir
    nr = InitNornir(config_file="config.yaml")

    print("\n--- Running CLI Commands on Devices ---")
    results = nr.run(task=run_cli_commands)
```

```
# 6. Iterate through the results for each host
for host_name, multi_result in results.items():
    print(f"\n=====")
    print(f"Processing results for: {host_name}")
    print(f"=====")

    if multi_result.failed:
        print(f"Task failed for {host_name}. Error: {multi_
else:
    command_outputs = multi_result[0].result

    for command, output in command_outputs.items():
        print(f"\n--- Output for command: '{command}' -
        print(output)
```

The config.yaml file (see [Example 11-2](#)) and the three inventory files—hosts.yaml (see [Example 11-3](#)), groups.yaml (see [Example 11-4](#)), and defaults.yaml (see [Example 11-5](#))—are the same ones used in [Chapter 10](#), with one small change in defaults.yaml to support the NAPALM connection. This modification is explained after [Example 11-5](#).

Example 11-2 *config.yaml*

```
inventory:
  plugin: SimpleInventory
  options:
    host_file: "hosts.yaml"
    group_file: "groups.yaml"
    defaults_file: "defaults.yaml"
runner:
  plugin: threaded
```

```
options:  
  num_workers: 5
```

Example 11-3 *hosts.yaml*

```
R1:  
  hostname: 192.168.1.1  
  groups: [ios]  
  data:  
    site: "HQ"  
    role: "edge"  
  
R2:  
  hostname: 192.168.2.2  
  groups: [ios]  
  data:  
    site: "Distribution"  
    role: "core"  
  
R3:  
  hostname: 192.168.3.2  
  groups: [ios]  
  data:  
    site: "Branch"  
    role: "edge"
```

Example 11-4 *groups.yaml*

```
ios:  
  platform: ios  
  data:  
    os_version: 16.6  
    model: 4331
```

Example 11-5 *defaults.yaml*

```
username: admin
password: cisco
connection_options:
  napalm:
    extras:
      optional_args:
        secret: spot
```

In the `defaults.yaml` file, there is a subtle but important difference between Netmiko and NAPALM:

- Netmiko expects **secret** directly under **extras**.
- NAPALM expects **secret** under **optional_args**.

You use the same number of threads in the `config.yaml` file with NAPALM as you used with Netmiko, which means the CLI commands were executed on all the three routers simultaneously.

The Output

Before we break down how the program in [Example 11-1](#) works, let's look at the output it produces (see [Example 11-6](#)). When this program runs, Nornir connects to each device in the inventory and executes two familiar IOS commands—**show version** and **show ip interface brief**—in parallel across all routers. Nornir then prints the results for each device in a clear, organized format. Examine the output to get a sense of what the program is doing before you move on to how it works.

Example 11-6 *Output from [Example 11-1](#)*

```
MyPrompt% python3 ex11-1_nornir_napalm_cli.py
```

```
--- Running CLI Commands on Devices ---
```

```
=====
Processing results for: R1
=====
```

```
--- Output for command: 'show version' ---
```

```
Cisco IOS XE Software, Version 16.06.03
```

```
Cisco IOS Software [Everest], ISR Software (X86_64_LINUX_IOSD-UI
16.6.3, RELEASE SOFTWARE (fc8)
```

```
Technical Support: http://www.cisco.com/techsupport
```

```
Copyright (c) 1986-2018 by Cisco Systems, Inc.
```

```
Compiled Wed 28-Feb-18 23:54 by mcpre
```

```
<output omitted for brevity>
```

```
Technology Package License Information:
```

```
-----
Technology      Technology-package      Technology-package
                  Current          Type          Next reboot
-----
```

appxk9	None	None	None
uck9	None	None	None
securityk9	None	None	None
ipbase	ipbasek9	Permanent	ipbasek9

```
cisco ISR4331/K9 (1RU) processor with 1796073K/6147K bytes of me
```

```
Processor board ID FLM1951W0R6
```

```
3 Gigabit Ethernet interfaces
```

```
32768K bytes of non-volatile configuration memory.
```

4194304K bytes of physical memory.
3207167K bytes of flash memory at bootflash:.
0K bytes of WebUI ODM Files at webui:.

Configuration register is 0x2102

--- Output for command: 'show ip interface brief' ---

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.1.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.1	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

=====
Processing results for: R2
=====

--- Output for command: 'show version' ---

Cisco IOS XE Software, Version 16.06.03
Cisco IOS Software [Everest], ISR Software (X86_64_LINUX_IOSD-UI
16.6.3, RELEASE SOFTWARE (fc8)
Technical Support: <http://www.cisco.com/techsupport>
Copyright (c) 1986-2018 by Cisco Systems, Inc.
Compiled Wed 28-Feb-18 23:54 by mcpre

<output omitted for brevity>

cisco ISR4331/K9 (1RU) processor with 1796073K/6147K bytes of me
Processor board ID FD02302A08A
3 Gigabit Ethernet interfaces
32768K bytes of non-volatile configuration memory.
4194304K bytes of physical memory.
3125247K bytes of flash memory at bootflash:.
0K bytes of WebUI ODM Files at webui:.

```
Configuration register is 0x2102
```

```
--- Output for command: 'show ip interface brief' ---
```

Interface	IP-Address	OK?	Method	Status
GigabitEthernet0/0/0	192.168.3.1	YES	NVRAM	up
GigabitEthernet0/0/1	192.168.2.2	YES	NVRAM	up
GigabitEthernet0/0/2	unassigned	YES	NVRAM	administrative
GigabitEthernet0	unassigned	YES	NVRAM	administrative

```
=====
```

```
Processing results for: R3
```

```
=====
```

```
--- Output for command: 'show version' ---
```

```
<output omitted for brevity>
```

```
--- Output for command: 'show ip interface brief' ---
```

```
<output omitted for brevity>
```

When the program runs, Nornir connects to all the devices in the inventory, executes the **show version** and **show ip interface brief** commands in parallel, and then displays the collected results.

High-Level Explanation of **Example 11-1**: What the Program Does

We will break down the program line by line, but first, here is a simple step-by-step explanation of what **Example 11-1** is doing. This summary mirrors the numbering in the code so you can understand the workflow without getting lost in the details:

Step 1.Import the required modules: The program loads the tools it needs:

- **InitNornir** to start Nornir and load the inventory
- **napalm_cli** to send CLI commands to devices through NAPALM

These imports make Nornir and NAPALM available for use.

Step 2.Define the task function: The function **run_cli_commands(task)** describes the work that Nornir will perform on each device. Inside it, a list of two IOS commands is created:

- **show version**
- **show ip interface brief**

This function becomes the unit of work that Nornir applies to every router in the inventory.

Step 3.Execute the CLI commands by using napalm_cli: Inside the task function is this line:

```
result = task.run(task=napalm_cli, commands=commands_to_
```

It tells Nornir to:

- Use the NAPALM driver for the device.
- Connect to the device.
- Run both CLI commands.
- Capture all of the output.

NAPALM executes the commands and returns the results.

Step 4.Return the results for this device: This line extracts the dictionary returned by **napalm_cli**:


```
return result[0].result
```

This dictionary contains one key/value pair per command:

```
"show version": "<full output>"  
"show ip interface brief": "<full output>"
```

Returning this dictionary allows the main section of the program to print the output for each device later.

Step 5. Initialize Nornir and run the task across all devices: **InitNornir** loads the configuration and inventory, creating the **nr** object that represents all devices:

```
nr = InitNornir(config_file="config.yaml")
```

Then this line tells Nornir to run the **run_cli_commands** task on all the devices in parallel:

```
results = nr.run(task=run_cli_commands)
```

Nornir handles all looping, connecting, and data collection automatically.

Step 6. Process and print the results: Finally, the program loops through each device in the results and:

- a.** Prints a header for readability
- b.** Checks whether the task failed
- c.** If it succeeded, retrieves the dictionary returned in step 4
- d.** Prints the CLI output for each command

By the end, the screen displays organized output for each router, showing the results of both commands.

Breaking Down the Code

In case you want to explore the details of the program in [Example 11-1](#) more deeply, this section walks through the program step by step. But keep in mind that you already learned the core Nornir workflow in the previous chapter. The way Nornir loads the inventory, runs tasks in parallel, and organizes results (**AggregatedResult** → **MultiResult** → **Result**) works exactly the same here. The main difference in this chapter is simply the plugin being used. Instead of sending commands with Netmiko, Nornir now uses NAPALM's device drivers. This means the structure of the program is familiar; you're just using a different tool underneath.

Next, we'll break down [Example 11-1](#) section by section so you can see exactly how it works and why.

The first section, **Import the required modules**, includes the following code:

```
from nornir import InitNornir
from nornir_napalm.plugins.tasks import napalm_cli
```

InitNornir initializes a Nornir automation session. It provides the function used later to initialize Nornir and load configuration and inventory data.

napalm_cli is the Nornir task provided by the `nornir_napalm` plugin. It uses the NAPALM driver for each device to run CLI commands and capture their output.

The second section, **Define the task function**, includes the following code:

```
def run_cli_commands(task):  
    commands_to_run = [  
        "show version",  
        "show ip interface brief"  
    ]
```

run_cli_commands(task) defines a Nornir task. Whenever Nornir runs a task, it passes in a **task** object that represents the current device being processed. This **task** object allows the function to run subtasks using **task.run()**, which is used in the next step.

The list **commands_to_run** contains the IOS commands that will be executed on each device.

Note

In later examples, you will see the **task** object used to access device-specific inventory data (**task.host**), but it is not needed in this example.

The third section, **Execute the CLI commands with napalm_cli**, includes the following code:

```
result = task.run(task=napalm_cli, commands=commands_to_run)
```

task.run() is used to call another Nornir task from inside the **task** function.

task=napalm_cli tells Nornir to execute the **napalm_cli** task on the current device.

commands=commands_to_run passes the list of IOS commands that should be executed. This call returns a **MultiResult** container, which contains one **Result** object

for the **napalm_cli** task. The command output is stored inside that **Result** object.

The fourth section, **Return the command-output dictionary**, includes the following code:

```
return result[0].result
```

The **result** attribute of that object contains a dictionary where:

- Each key is the CLI command.
- Each value is the raw text output from the device.

This dictionary becomes the return value of **run_cli_commands**.

The fifth section, **Initialize Nornir and run the task**, includes the following code:

```
nr = InitNornir(config_file="config.yaml")
results = nr.run(task=run_cli_commands)
```

InitNornir() loads the configuration and inventory files and prepares the Nornir environment.

nr.run(task=run_cli_commands) executes the task on all devices in the inventory.

The returned object (**results**) is a dictionary-like structure that maps each device name (for example, R1, R2) to a **MultiResult** container that holds the task's output.

The sixth section, **Process and print results**, includes the following code:

```
for host_name, multi_result in results.items():
```

This loop processes the results for each device:

- **host_name:** The inventory name of the device (for example, **R1**).
- **multi_result:** The **MultiResult** container for that device.

This is what happens inside the loop:

- If the task failed, print the error:

```
if multi_result.failed:  
    print(f"Task failed for {host_name}. Error: {multi_result[0].e
```

- Otherwise, extract the dictionary of **command_outputs**:

```
command_outputs = multi_result[0].result
```

- Then print the output for each command:

```
for command, output in command_outputs.items():  
    print(f"\n--- Output for command: '{command}' ---")  
    print(output)
```

This formatting makes it easy to read the output for each router and for each command.

Note

If any part of this structure feels unfamiliar, go back to [Chapter 10](#), which includes a fuller explanation of how Nornir tasks and result objects work, including **nr.run()**, **Result**, and **MultiResult**. It may be helpful to review it as needed. Remember, though, that you don't need to focus on every line; just focus on what the framework is accomplishing.

Using Nornir and the `napalm_get` Task

The **napalm_get** task is one of the most important and widely used tasks in the **nornir_napalm** plugin. It provides a standardized way to retrieve structured operational data from network devices—regardless of vendor—using NAPALM’s unified data-collection methods. Instead of returning raw CLI text, **napalm_get** returns information in clean, Python-friendly dictionaries, which makes the information much easier to analyze, validate, or store.

A single call to **napalm_get** can retrieve many different types of operational data in one connection session. NAPALM handles the vendor-specific differences behind the scenes, so the same request works across devices from Cisco, Juniper, Arista, and others.

Note

In NAPALM documentation, each specific type of information you request (such as facts, interfaces, or BGP neighbors) is often referred to as a *getter*. In this book, we simply talk about “types of structured information” to keep things easier to follow. If you see the term *getter* in code or examples, know that it simply means “a type of information NAPALM can retrieve.”

At the time of this writing, NAPALM can collect the following types of structured information:

- **Device information:** With NAPALM, you can collect device information by using the following getters:
 - **facts:** Basic information such as hostname, model, OS version, uptime, and serial number
 - **environment:** Temperatures, fan status, and power supply information

- **Interface information:** With NAPALM, you can collect interface information by using the following getters:
 - **interfaces:** Administrative and operational status
 - **interfaces_ip:** IP addressing information
 - **interfaces_counters:** Interface traffic statistics
- **Routing information:** With NAPALM, you can collect routing information by using the following getters:
 - **route:** Routing table entries
 - **arp_table:** ARP cache entries
- **BGP information:** With NAPALM, you can collect BGP information by using the following getters:
 - **bgp_neighbors:** Neighbor states and summary
 - **bgp_neighbors_detail:** Detailed per-neighbor information
- **Configuration data:** With NAPALM, you can collect configuration data by using the following getters:
 - **config:** The running configuration (may be large)
- **Network services information:** With NAPALM, you can collect network services information by using the following getters:
 - **lldp_neighbors:** The LLDP neighbor list
 - **lldp_neighbors_detail:** Detailed LLDP information
 - **ntp_servers:** NTP server configuration
 - **ntp_peers:** NTP peer status
 - **ntp_stats:** Time synchronization statistics

- **Security information:** With NAPALM, you can collect security information by using the following getters:
 - **Users:** Local user accounts
 - **snmp_information:** SNMP configuration
- **Optical/physical information:** With NAPALM, you can collect optical/physical information by using the following getters:
 - **optics:** Transceiver and optical signal data

Note that not all devices support every type of information listed. Some may return empty results if a feature is not available or not configured. In addition, collecting very large datasets—for example, the full running configuration or a large routing table—may consume significant memory if done across many devices at once. It's important to use these options thoughtfully in large environments.

The **napalm_get** task is the foundation of automated data collection in Nornir and NAPALM workflows. Its structured, vendor-neutral output makes it an essential tool for documentation, compliance checks, troubleshooting, network validation, inventory building, and dashboards and reporting. The **napalm_get** task is one of the most valuable tasks in your automation toolbox.

Now that you've seen what kinds of structured information NAPALM can retrieve, let's look at a simple example that uses the **napalm_get** task to collect one of the most common types of device information: basic device facts (such as the hostname, vendor, operating system version, and uptime).

Introducing the `napalm_get` Task Example

The program in [Example 11-7](#) uses Nornir to connect to every device in our inventory, asks NAPALM to retrieve the **facts** dataset, and then prints a few key values in a clean, readable format for each router. Just as with the earlier **`napalm_cli`** example, Nornir handles running the task across multiple devices in parallel and returns a structured result for each device. At a high level, this program:

- Initializes Nornir and loads the inventory.
- Defines a task that retrieves device facts by using NAPALM.
- Checks for errors.
- Extracts the structured data returned by NAPALM.
- Prints selected fields (hostname, vendor, OS version, uptime).

When the program runs, it produces an output block for each device, showing these details. This makes it easy to confirm device identity and software versions across the network.

Example 11-7 *ex11-7_nornir_napalm_get.py*

```
# 1. Import required modules
from nornir import InitNornir
from nornir_napalm.plugins.tasks import napalm_get

# 2. Define the task
def get_device_facts(task):
    print(f"--- Attempting to get facts for {task.host.name} --")
    try:
```

```
# 3. Use NAPALM to retrieve facts
    result = task.run(task=napalm_get, getters=["facts"])

# 4. Check if the task was successful and returned data
    if result.failed:
        print(f"!!! Task FAILED for {task.host.name}: {result}")
    return

# 5. The result is a MultiResult object, with each sub-result containing facts
    facts = result[0].result['facts']

    print(f"--- Successfully retrieved facts for {task.host.name}")
    print(f"  Hostname: {facts.get('hostname', 'N/A')}")
    print(f"  Vendor: {facts.get('vendor', 'N/A')}")
    print(f"  OS Version: {facts.get('os_version', 'N/A')}")
    print(f"  Uptime: {facts.get('uptime', 'N/A')} seconds")
    print("-" * 40)

# 6. Catch any unexpected errors gracefully
    except Exception as e:
        print(f"!!! An unexpected error occurred for {task.host.name}: {e}")

if __name__ == "__main__":
# 7. Initialize Nornir
    nr = InitNornir(config_file="config.yaml")

# 8. Run the task
    results = nr.run(task=get_device_facts)

    print("\n--- Script Finished ---")
```

The Output from [Example 11-7](#)

Before we dive into the explanation, look at the output of the program shown in [Example 11-8](#). For each router in the inventory, Nornir calls NAPALM's **get_facts()** method through the **napalm_get** task. This method returns a consistent set of device information across vendors, and the script prints a few key fields, such as hostname, vendor, OS version, and uptime. Look at the output to get an idea of what the script is gathering before you move on to the discussion of how it does it.

Example 11-8 *Output from [Example 11-7](#)*

```
MyPrompt% python3 ex11-01_nornir_napalm_get.py
--- Attempting to get facts for R1 ---
--- Attempting to get facts for R2 ---
--- Attempting to get facts for R3 ---
--- Successfully retrieved facts for R3 ---
  Hostname: R3
  Vendor: Cisco
  OS Version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Ve
SOFTWARE (fc8)
  Uptime: 6960.0 seconds
-----
--- Successfully retrieved facts for R2 ---
  Hostname: R2
  Vendor: Cisco
  OS Version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Ve
SOFTWARE (fc8)
  Uptime: 6960.0 seconds
-----
--- Successfully retrieved facts for R1 ---
  Hostname: R1
```

```
Vendor: Cisco
OS Version: ISR Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Ve
SOFTWARE (fc8)
Uptime: 6960.0 seconds
-----
--- Script Finished ---
```

Note

Because Nornir runs tasks in parallel, devices may finish at different times. As a result, the output may appear in a different order than the device list in `hosts.yaml`. This is normal and expected when using the threaded runner. If you prefer the results to appear in a specific order, you can sort them when iterating over **`results.items()`**. Because this is standard Python logic and not specific to Nornir or NAPALM, you can use a generative AI tool to quickly generate an example of sorting and printing the results, if needed.

High-Level Explanation of **Example 11-7**: What the Program Does

Here is a simple step-by-step explanation of what the program in [Example 11-7](#) is doing. The steps listed here match the numbered comments in the code to help you understand the workflow without getting lost in syntax.

Step 1. Import required modules: The program loads the tools it needs:

- **InitNornir:** To start Nornir and load the inventory

- **napalm_get:** To retrieve structured information from devices using NAPALM

These imports make Nornir and NAPALM available for the rest of the script.

Step 2. Define the task: The function **get_device_facts(task)** describes the work that Nornir performs on each device:

- It prints a message showing which device it is about to process (**task.host.name**).
- It starts a **try** block so that any errors for this device can be handled cleanly without stopping the entire script.

This function is the unit of work that Nornir applies to every router in the inventory.

Step 3. Use NAPALM to retrieve facts: Inside the **task** function, notice this line:

```
result = task.run(task=napalm_get, getters=["facts"])
```

This line tells Nornir to:

- Use NAPALM's device driver for the current host.
- Call NAPALM's **get_facts()** method (via **napalm_get** with **getters=["facts"]**).
- Collect a standard set of basic device information, including hostname, vendor, OS version, and uptime.

NAPALM executes this operation and returns the results in a structured format.

Step 4. Check if the task was successful: Right after the call to **napalm_get**, the program checks whether the task was successful:

```
if result.failed:

    print(f"!!! Task FAILED for {task.host.name}: {result}")
    return
```

If something went wrong for this device—such as a connection error or authentication problem—**result.failed** will be **True**. In that case, an error message is printed for that device, and the function returns early, skipping the rest of the processing for this host and allowing Nornir to continue with the other devices.

Step 5. Extract and print the structured facts: If the task succeeded, the next line is as follows:

```
facts = result[0].result['facts']
```

It does the following:

- **result[0]** gets the first (and only) **Result** object from the **MultiResult** container returned by **task.run()**.
- **.result** is a dictionary of all the data returned by **napalm_get**.
- **['facts']** extracts the specific dictionary that contains the device facts.

Then the program prints selected fields:

```
print(f"  Hostname: {facts.get('hostname', 'N/A')}")

print(f"  Vendor: {facts.get('vendor', 'N/A')}")
print(f"  OS Version: {facts.get('os_version', 'N/A')}")
print(f"  Uptime: {facts.get('uptime', 'N/A')} seconds")
```

Using **facts.get('key', 'N/A')** is a safe way to access each value; if a key is missing, it prints **N/A** instead of causing an error.

At the end of this step, you see a short summary of key information for the device.

Step 6. Catch any unexpected errors gracefully: The **except Exception as e:** block catches any other unexpected errors that might occur inside the **try** block for this device:

```
except Exception as e:  
  
    print(f"!!! An unexpected error occurred for {task.l
```

This prevents a single problematic device from crashing the entire program and gives you a helpful error message instead.

Step 7. Initialize Nornir: In the section:

```
if __name__ == "__main__":
```

the script calls:

```
nr = InitNornir(config_file="config.yaml")
```

This loads the Nornir configuration from config.yaml, loads the inventory files (hosts.yaml, groups.yaml, and defaults.yaml), and creates the **nr** object, which represents all devices and their connection details.

At this point, Nornir is ready to run tasks on the inventory.

Step 8. Run the task and finish: Finally, the script runs:

```
results = nr.run(task=get_device_facts)
```

This tells Nornir, “Run the **get_device_facts** task on every device in the inventory (in parallel).” Nornir handles all the looping, connecting, and data collection.

When all devices have been processed, the script prints the following:

```
print("\n--- Script Finished ---")
```

This indicates that everything is complete.

By the end of this program, you have a clear, structured snapshot of basic facts for every device in your inventory—retrieved automatically and consistently using Nornir and NAPALM.

Note

To keep this chapter focused and avoid repeating material already covered in earlier examples, this program does not include a separate “[Breaking Down the Code](#)” section. Nearly all of the underlying Nornir mechanics—such as how tasks run, how **task.run()** works, and how **Result** and **MultiResult** objects behave—are explained in [Chapter 10](#) and earlier in this chapter. Here, the goal is simply to show how the same familiar Nornir workflow can be used with NAPALM’s data-collection capabilities, keeping the explanation as clear and straightforward as possible.

Using Nornir and the `napalm_configure` Task

Before Nornir can use NAPALM to push configuration changes, the network device must permit remote configuration loading. On Cisco IOS/IOS XE devices, NAPALM uses SCP (Secure Copy Protocol) behind the scenes to transfer a temporary configuration file to the router. Two prerequisites must be configured on the device:

```
Router(config)# ip scp server enable
Router(config)# username admin privilege 15 password 0 cisco
```

Note

On newer IOS XE releases (such as 17.12), you may see a warning indicating that the password was configured using type 0 (plaintext) encryption and recommending migration to stronger password types (such as type 6). This warning is informational and does not affect the operation of NAPALM in this example. For simplicity and consistency with earlier chapters, plaintext passwords are used here, but stronger password types are recommended in production environments.

ip scp server enable is required because it enables the built-in SCP server. NAPALM uploads candidate configuration files over SCP before merging or replacing them.

In addition, NAPALM needs a user with sufficient rights (privilege level 15) to load candidate configs and commit them. (This username and password must match what you have configured in your Nornir inventory.)

The **napalm_configure** task is the primary way Nornir uses NAPALM to push configuration changes to network devices. It provides a structured, vendor-agnostic workflow for safely modifying device configurations at scale. Whether you need to make a small incremental change or deploy an entire

configuration, **napalm_configure** handles the heavy lifting and keeps the process predictable and consistent.

At a high level, **napalm_configure** supports several important capabilities:

- **Configuration deployment:** It applies configuration changes to devices in a controlled manner.
- **Merge and replace modes:** It can add or modify specific lines (in merge mode) or overwrites the full configuration (in replace mode).
- **Dry-run capability:** It previews configuration changes without committing them (which is a critical best practice).
- **Diff generation:** It automatically shows what will change before any change is applied.
- **Atomic commits:** Configuration changes are applied as a single transaction.
- **Rollback support:** Some platforms can automatically revert changes if they are not confirmed.

Together, these features make **napalm_configure** one of the safest and most reliable tools for network automation, especially when managing multiple devices simultaneously. Even a simple configuration task benefits from the built-in safeguards NAPALM provides.

Understanding the **napalm_configure** Parameters

[Example 11-9](#) shows a block of code that is not meant to be run as a standalone Python program. Instead, it serves as a reference, showing what parameters the **napalm_configure** task accepts when called from within a

Nornir task. (This function is used later in the chapter, in [Example 11-10](#).)

Example 11-9 *napalm_configure* Parameters

```
from nornir_napalm.plugins.tasks import napalm_configure

result = task.run(
    task=napalm_configure,
    configuration=None,    # Configuration string (optional)
    filename=None,        # Path to configuration file (optional)
    dry_run=True,         # Preview changes without committing
    replace=False,        # Merge (default) or replace mode
    revert_in=None        # Auto-revert timer in seconds
)
```

Here is what each parameter does:

- **configuration:** This parameter is a string that contains the configuration commands you want to push to the device. Use this parameter when you want to define the configuration directly inside your Python program (as in [Example 11-10](#), later in this chapter). The commands must follow the exact syntax of the device's CLI, and the full configuration must be valid for the platform.
- **filename:** This parameter is a string that specifies the path to a file that contains the configuration commands. This is useful when storing configuration templates in external files, especially when working with version control systems such as Git.

Note

You must use *either* **configuration** or **filename**, but not both at the same time.

- **dry_run:** This parameter is a Boolean that controls whether the configuration is actually applied. It can be set to either **True** or **False**:
 - **dry_run=True:** Loads the configuration, generates a diff, and discards the diff without applying anything. It is strongly recommended to use this preview mode before pushing real changes. This is the default.
 - **dry_run=False:** Loads the configuration, generates a diff, and commits the changes to the device.

Note

It is a best practice to always run **dry_run=True** first so you can verify the diff and ensure that the expected changes will be applied.

- **replace:** This parameter is a Boolean that determines whether the configuration is merged or replaced. It can be set to either **True** or **False**:
 - **replace=False:** This is the default setting, which uses merge mode to add or modify configuration lines but does not remove anything unless explicitly negated. This is the safer and more commonly used mode.
 - **replace=True:** This setting uses replace mode to replace the entire running configuration with the supplied configuration. Anything that is not in the new configuration is removed. This mode is used for full provisioning and requires a complete, correct configuration. Use it with caution and verify platform support as some vendors do not implement replace mode.
- **revert_in:** This parameter is an integer that enables an automatic rollback timer. When it is used, the configuration is applied, and a countdown begins. If the

configuration is not confirmed within the timer period, the device automatically rolls back to the previous configuration. This is an excellent safety mechanism for remote changes, but a confirmation must be sent before the timer expires. In addition, not all platforms support it.

Introducing the `napalm_configure` Example

Now that you've seen how NAPALM retrieves data by using **`napalm_get`**, let's look at how it can push configuration changes to devices.

[Example 11-10](#) shows a simple but realistic use case: creating a new loopback interface and adding a description to it on every device in the Nornir inventory. This script demonstrates how **`napalm_configure`** loads configuration changes, applies them safely using NAPALM's merge mode, checks for errors, and reports success for each device. It's intentionally small and focused so you can clearly see how configuration changes flow through Nornir and NAPALM. Once you understand this pattern, you can expand it to larger configuration templates, per-device logic, or version-controlled configuration files.

Example 11-10 `ex11-10_nornir_napalm_configure.py`

```
# 1. Import required modules
from nornir import InitNornir
from nornir_napalm.plugins.tasks import napalm_configure

# 2. Define the Nornir task function
def configure_interface(task):
    print(f"*** Running configure_interface on {task.host.name}")
```

```

# 3. Define the configuration snippet to be applied
    config_snippet = """
interface Loopback 100
    description Configured by Nornir + NAPALM
    """

# 4. Execute the napalm_configure task
    result = task.run(
        task=napalm_configure,
        configuration=config_snippet,
        replace=False,
        dry_run=False
    )

# 5. Extract the result for the napalm_configure task
    cfg_result = result[0]

# 6. Check for configuration application failure
    if cfg_result.failed:
        print(f"!!! Configuration FAILED on {task.host.name}: {cfg_result.failed}")
        return # Exit the task for this host if it failed

# 7. Report success
    print(f"Configuration applied successfully on {task.host.name}")

if __name__ == "__main__":
# 8. Initialize Nornir
    nr = InitNornir(config_file="config.yaml")

# 9. Run the 'configure_interface' task across all hosts

```

```
results = nr.run(task=configure_interface)

# 10. Print overall failure status
print("Any failures overall?:", results.failed)
print("--- Script finished ---")
```

High-Level Explanation of **Example 11-10**: What the Program Does

Here is a simple, big-picture walkthrough of what the program in [Example 11-10](#) is doing. These steps correspond directly to the numbered comments in the code to help you understand the workflow without getting lost in syntax:

Step 1. Import required modules: The script begins by loading two components:

- **InitNornir:** This component starts the Nornir automation framework and loads the inventory.
- **napalm_configure:** This NAPALM task is responsible for pushing configurations to devices.

These are the only two imports needed to configure routers with Nornir and NAPALM.

Step 2. Define the Nornir task function: The function **configure_interface(task)** describes the work Nornir will perform on each device. It prints a message that shows which device is currently being configured. This function becomes the unit of work that Nornir executes across all routers in the inventory.

Step 3. Define the configuration snippet: Inside the task, a small block of IOS configuration is created

with the name **config_snippet**:

```
interface Loopback100
```

```
description Configured by Nornir + NAPALM
```

This is the exact configuration that will be applied to each device. In this example, the script creates Loopback100 if it does not exist or updates its description if it does.

Note

The triple quotes (""") are used to create a multiline string. To do the same thing on one line, use the following code:

```
config_snippet = "interface Loopback 100\n description Co
```

Step 4. Execute the napalm_configure task: This line tells Nornir to connect to the device using NAPALM, load the configuration snippet, merge it into the existing running configuration (**replace=False**), and actually apply the change (**dry_run=False**):

```
result = task.run(task=napalm_configure, configuration=c  
dry_run=False)
```

NAPALM handles loading, diffing, and committing the configuration safely.

Step 5. Extract the result: Because **task.run()** returns a **MultiResult** container, the script fetches the first result:

```
cfg_result = result[0]
```


This object contains the outcome of the configuration attempt, including any diff or error messages.

Step 6. Check for configuration failures: If something went wrong (for example, syntax issue, missing privilege, connection failure), the result of this line will be **True**:

```
cfg_result.failed
```

In that case, an error is printed for that particular device, and the task stops for that host. Nornir then moves on to the next device.

Step 7. Report success: If no errors occurred, the script prints a simple success message:

```
Configuration applied successfully on R1.
```

This confirms that the device accepted and committed the configuration.

Step 8. Initialize Nornir: This line loads the inventory, connection settings, defaults, and groups:

```
nr = InitNornir(config_file="config.yaml")
```

It creates the **nr** object, which represents all devices on which Nornir will operate.

Step 9. Run the task across all hosts: This call tells Nornir to run **configure_interface()** on **every device** in the inventory, do it in parallel (threaded runner), and collect all results into one object:

```
results = nr.run(task=configure_interface)
```

:

This is where the configuration is actually deployed.

Step 10. Print the overall failure status: Finally, the script shows whether any device failed:

```
print("Any failures overall?:", results.failed)
```

If even one router encountered an error, this prints **True**. Otherwise, it prints **False**, meaning the entire automation run was successful.

Note

To keep this chapter focused and avoid repeating concepts already covered, there is no detailed “[Breaking Down the Code](#)” section for this example. The high-level explanation provided contains everything you need to understand how the program works.

The Output

Running the code in [Example 11-10](#) in a lab environment results in the output shown in [Example 11-11](#).

Example 11-11 Output from [Example 11-10](#)

```
MyPrompt% python3 ex11-10_nornir_napalm_configure.py
*** Running configure_interface on R1 ***
*** Running configure_interface on R2 ***
*** Running configure_interface on R3 ***
Configuration applied successfully on R3.
Configuration applied successfully on R2.
Configuration applied successfully on R1.
```

```
Any failures overall?: False
--- Script finished ---
```

Because Nornir executes tasks in parallel, you might notice that the success messages appear out of order relative to the `hosts.yaml` inventory. This is normal. Each device finishes as soon as its own configuration operation completes.

To confirm that the configuration was actually applied, you can display the relevant portion of the running configuration on each router, as shown in [Example 11-12](#) for R1.

Example 11-12 *Verifying That R1 Has Loopback 100 Configured*

```
R1#show running-config | begin interface Loopback100
interface Loopback100
  description Configured by Nornir + NAPALM
  no ip address
!
<output omitted>
```

Note

Because this example uses **replace=False**, NAPALM performs a *merge* operation, adding or modifying only the lines specified in the configuration snippet. This is the safest and most commonly used mode for incremental changes.

Simplified Workflow of the `napalm_configure` Task (Optional)

This section summarizes the internal workflow NAPALM follows when the **`napalm_configure`** task is executed. You do *not* need to know this process to use Nornir or **`napalm_configure`** effectively; Nornir handles all of these steps automatically. However, you might find it helpful to understand what is happening behind the scenes when a configuration is pushed to a device. If you're curious about the underlying mechanics, read on as this optional section provides a clear, high-level look at the sequence NAPALM follows, from start to finish:

Step 1.Task invocation: This is where the process begins. Nornir calls the **`napalm_configure`** task and validates the input parameters (configuration, filename, replace, dry_run, and so on) before attempting any device interaction.

Step 2.Connection establishment: NAPALM determines which type of device it is connecting to and establishes a management connection by using the credentials from the inventory. It loads the correct driver (IOS, NX-OS, Junos, and so on), establishes a connection to the device, and performs authentication.

Step 3.Configuration loading: NAPALM prepares the configuration changes by loading them into a candidate configuration buffer. Nothing is applied to the device yet. NAPALM calls either **`load_merge_candidate()`** (merge mode) or **`load_replace_candidate()`** (replace mode). The configuration (from a string or file) is staged as a candidate configuration.

Step 4. Difference calculation: Before applying anything, NAPALM compares the candidate configuration to the device's running configuration and generates a diff. It executes **compare_config()**, and a diff is generated, showing what would change. NAPALM checks whether the candidate differs from the running configuration.

Step 5. Decision point: At this stage, NAPALM decides whether to apply the changes or simply return the diff, depending on the **dry_run** setting. If **dry_run=True**, the diff is returned and the candidate configuration is discarded. If **dry_run=False**, NAPALM proceeds to commit the configuration.

Step 6. Configuration commit (only when dry_run=False): NAPALM now applies the configuration to the device in a controlled, transactional way. **commit_config()** applies the candidate configuration, and the new configuration becomes active. If **revert_in** is set, a rollback timer begins.

Step 7. Result return: Once the operation is complete, NAPALM returns the results and cleans up the connection. The final diff is returned (whether it was committed or not), and success or failure status is included. Then the connection to the device is closed.

Summary

In this chapter, you expanded your Nornir automation skills by learning how to integrate NAPALM, a powerful multivendor network automation library that provides

structured, consistent access to operational and configuration data across different network platforms. Whereas earlier chapters focus on retrieving information and configuring devices using Netmiko, this chapter shows how NAPALM allows you to work at a higher, more standardized level—free of device-specific CLI parsing or syntax differences.

You began the chapter by using **napalm_cli** to execute **show** commands in a structured way and then moved on to using **napalm_get** to retrieve normalized operational data such as facts, interface status, and routing information. Through these tasks, you saw that NAPALM returns clean Python dictionaries instead of raw CLI output, making it ideal for automation.

Finally, you learned how to apply configuration changes safely by using **napalm_configure**. You learned about merge behavior and replace behavior, dry-run validation, and rollback mechanisms. You also saw that NAPALM requires SCP and a properly privileged user account on IOS devices before configuration changes can be applied successfully.

Across these examples, you saw that the Nornir workflow remains consistent, regardless of the underlying plugin:

Step 1. Define a task.

Step 2. Run the task across all devices.

Step 3. Process the **Result** and **MultiResult** objects.

This chapter shows how seamlessly you can shift from Netmiko-based automation to the richer, vendor-agnostic capabilities provided by NAPALM. With the techniques learned here, you can now gather structured data, validate device state, and push configuration changes across many

network devices reliably and safely. These are essential building blocks for scalable network automation.

Chapter 12. Inventory Management with Nornir

In this chapter, we move into more advanced capabilities of Nornir by taking a deeper look at inventory management. If some of the ideas introduced here feel unfamiliar—or even a bit overwhelming at first—don’t worry. You are not expected to fully understand every detail.

The goal of this chapter is to help you understand the big picture: how Nornir represents devices, how information about those devices is organized, and why this structured approach is so powerful for automation. As with earlier chapters, focus on the concepts and patterns rather than the syntax. The details will naturally fall into place as you revisit these ideas and apply them in your own work.

In [Chapter 11](#), “[Using Nornir with NAPALM](#),” you learned how to use Nornir together with NAPALM to automate configuration changes across multiple devices at once. This combination is powerful because each tool focuses on what it does best: NAPALM handles device interaction, while Nornir handles orchestration and coordination. When used together, Nornir and NAPALM allow you to manage networks more efficiently, especially in

environments that include devices from multiple vendors.

Before moving forward, let's briefly review why this combination works so well:

- **Consistent multivendor interaction:** NAPALM provides a single, vendor-agnostic API for interacting with different network operating systems, allowing you to use the same Python methods across platforms such as Cisco IOS, Juniper Junos, and Arista EOS.
- **Structured data and safer configuration management:** NAPALM returns information in predictable, structured formats and supports configuration diffs, merge/replace operations, and rollbacks—making automation both easier to reason about and safer to apply.
- **Scalable, parallel execution:** Nornir is designed to run tasks across many devices simultaneously, allowing you to collect data or apply changes efficiently across large networks.
- **Pure Python flexibility and extensibility:** Both Nornir and NAPALM are Python based and plugin driven, giving you full control over logic, error handling, and workflow design without relying on a domain-specific language.
- **Centralized inventory management:** Most importantly for this chapter, Nornir provides a flexible way to store and manage information about devices—such as platforms, credentials, roles, and metadata—that drives all automation behavior.

Up to this point, we've focused mainly on *what* tasks Nornir runs and how those tasks interact with devices. In this chapter, we shift our focus to how Nornir knows *which* devices exist and *what* it knows about them. In other words, this chapter is about *inventory*—the structured source of truth that makes everything else in Nornir possible.

A Quick Overview with a Focus on Inventory

Before we dive into the details of inventory management with Nornir, it's important to understand what Nornir's inventory is—and why it matters.

At a high level, Nornir's inventory is simply a structured way to describe a network. It answers questions such as:

- What devices exist?
- How do I connect to them?
- What type of devices are they?
- What characteristics do they share?

You can think of the inventory as a centralized repository, or even a well-organized spreadsheet, that Nornir consults every time it runs a task. Instead of hard-coding IP addresses, usernames, or device types into your Python programs, you allow that information to live in the inventory and be reused consistently across your automation workflows.

As you may remember, the `config.yaml` file tells Nornir where to find the inventory files. The inventory itself is divided into three parts—`hosts.yaml`, `groups.yaml`, and `defaults.yaml`—as shown in [Figure 12-1](#) (and originally

introduced in [Chapter 9](#), “Introducing Nornir: A Pythonic Framework for Network Orchestration”).

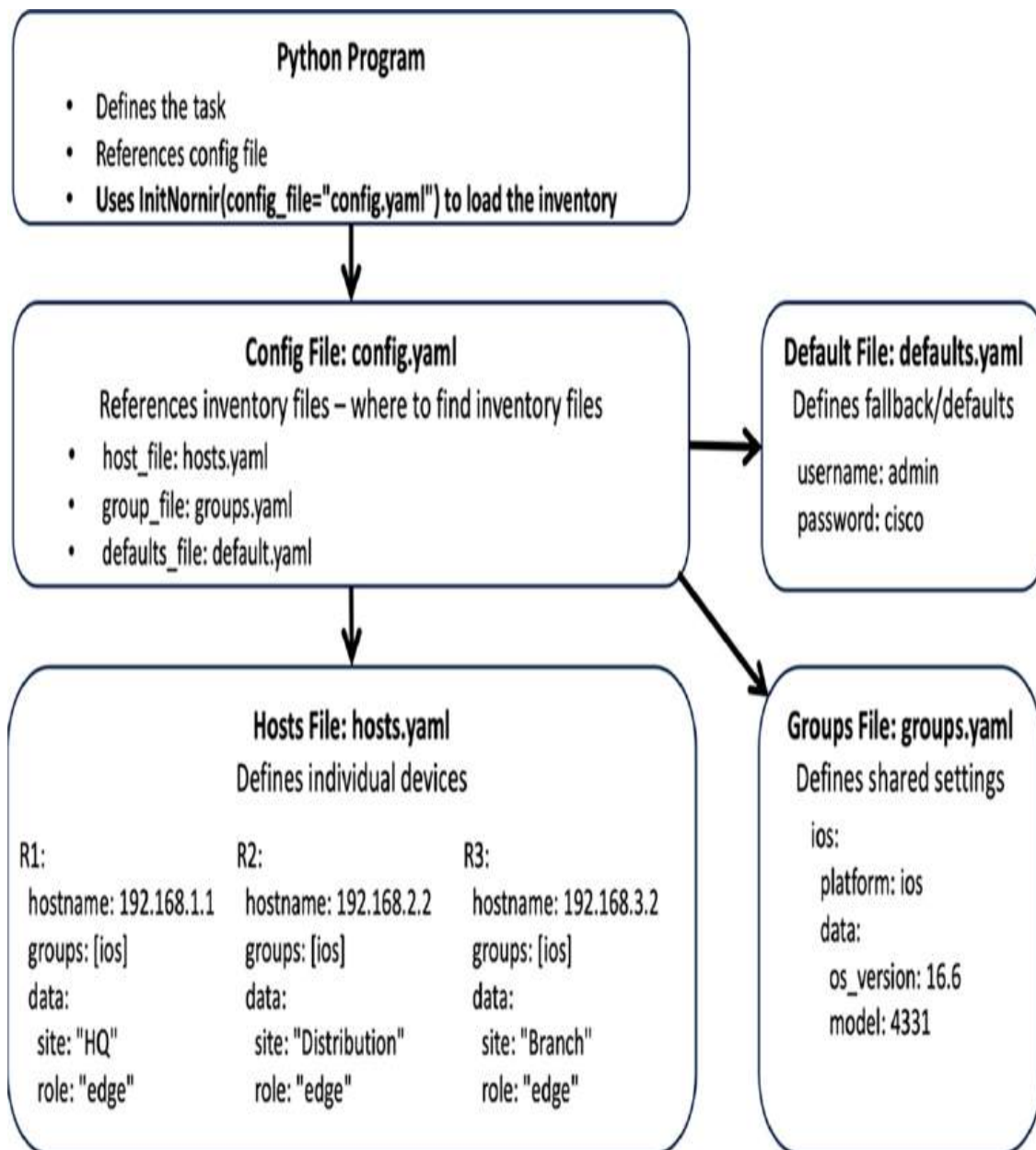


Figure 12-1 *How Nornir uses configuration and inventory files to generate task output*

When inventory data is loaded from YAML files (hosts.yaml, groups.yaml, and defaults.yaml), Nornir uses its built-in SimpleInventory plugin, which is configured as part of

Nornir's initialization—typically in the `config.yaml` file. This plugin is commonly used when getting started and is the inventory source used in earlier chapters. `SimpleInventory` and other inventory plugins are discussed later in this chapter.

Note

Internally, Nornir loads this information and converts it into Python objects—called host objects—that your automation tasks can access and work with dynamically. A *host object* contains all the inventory data for a device.

Nornir's inventory system is built around a simple hierarchical model with three layers:

- **Hosts file (`hosts.yaml`):** Hosts are individual network devices, such as routers, switches, and firewalls. Each host describes a single device and includes information such as its IP address or hostname, platform type, credentials (usernames and passwords), and any custom metadata you want to associate with it.
- **Groups file (`groups.yaml`):** Groups allow you to organize hosts that share common characteristics—such as device role, location, vendor, or network operating system (NOS). Settings defined at the group level are automatically inherited by all member hosts, reducing repetition and keeping your inventory easy to maintain. Group settings may also be overridden by host-specific settings.
- **Defaults file (`defaults.yaml`):** Defaults define values that apply to all hosts unless overridden. `defaults.yaml` is typically where global settings—such as default credentials or common network-wide parameters—are stored.

Inheritance

Nornir applies inventory values using a clear inheritance order:

- Hosts values override both groups and defaults. (They have the highest priority.)
- Group values override default values.
- Defaults apply to all devices. (They have the lowest priority.)

[Figure 12-2](#) illustrates this hierarchical relationship and fallback order.

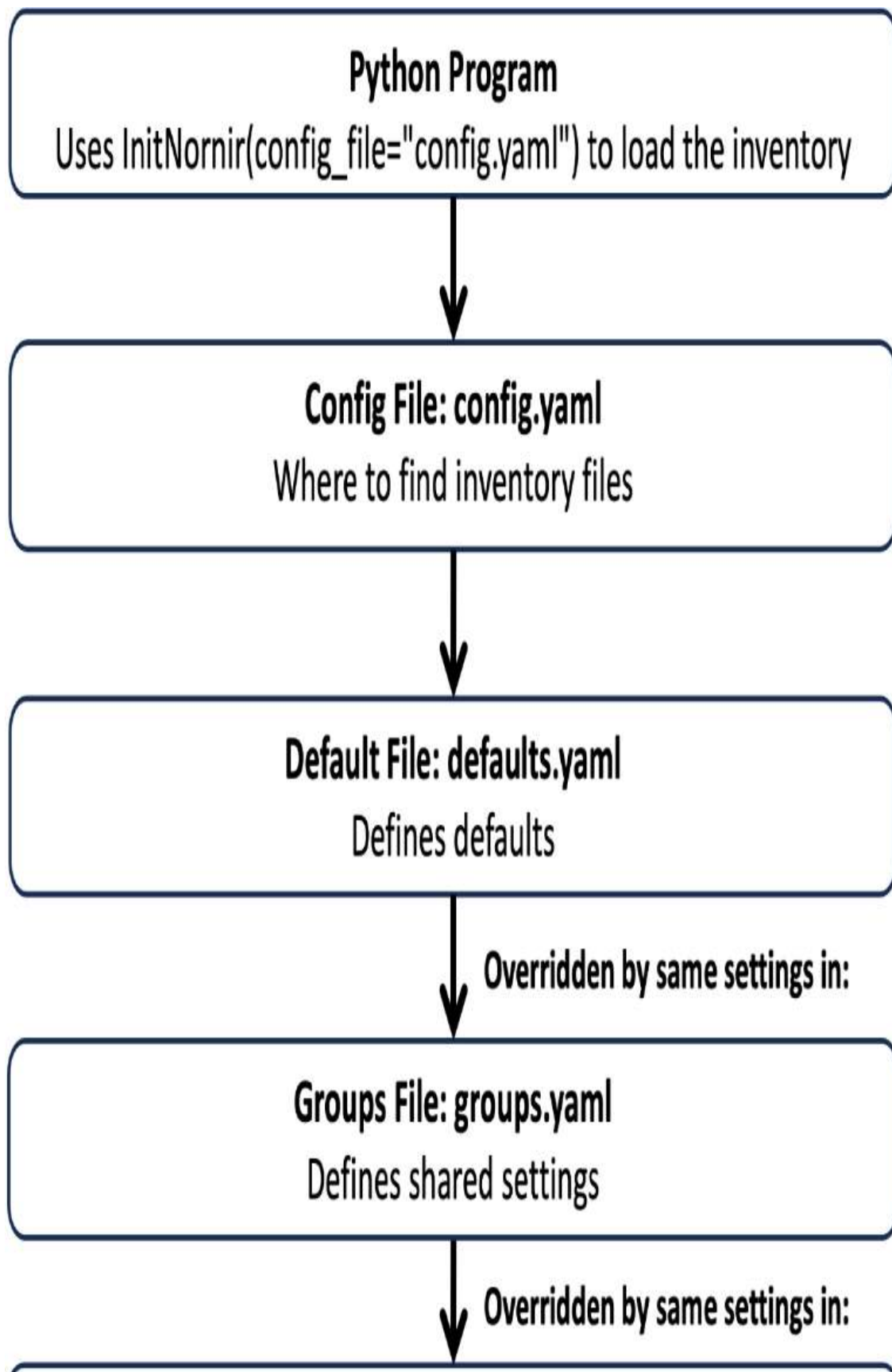


Figure 12-2 *Fallback order of inventory files*

Host-specific values defined in `hosts.yaml` take precedence. If a value is not defined on the host, Nornir checks the groups in `groups.yaml` and inherits the value, if it is present. If the value is not defined at either the host or group level, Nornir falls back to `defaults.yaml`. This inheritance model ensures that every host has a complete set of inventory values while allowing more specific definitions to override more general ones.

Together, these three layers form the foundation of how Nornir understands a network.

Where Inventory Data Comes From

While the structure of the inventory is always the same, Nornir is flexible about where the data comes from. Inventory information can be loaded from a variety of sources using inventory plugins.

These are some common inventory sources:

- **YAML files:** This is the option we've been using so far. It is simple and popular, especially when getting started. Inventory data is stored in three YAML files—`hosts.yaml`, `groups.yaml`, and `defaults.yaml`—which together describe all devices and shared settings. In earlier chapters, you used Nornir's built-in `SimpleInventory` plugin (defined in `config.yaml`) to load inventory data from these three files:

```
inventory:
  plugin: SimpleInventory
  options:
    host_file: "hosts.yaml"
```

```
group_file: "groups.yaml"  
defaults_file: "defaults.yaml"
```

- **Python dictionaries:** By using the DictInventory plugin, inventory data can be created programmatically inside a Python program.
- **Network source of truth (NSoT) systems:** Tools such as NetBox act as centralized systems of record for network infrastructure and can dynamically populate Nornir's inventory.
- **Ansible inventories:** Existing Ansible inventory files can be reused, allowing organizations to adopt Nornir without duplicating data.
- **CSV or Excel files:** Inventory stored in spreadsheets can be imported using plugins such as `nornir_table_inventory`.
- **Other specialized systems:** Platforms like InfraHub provide modern infrastructure data sources.

Regardless of the source, all inventory data is normalized into the same hosts, groups, and defaults model inside Nornir. Once Nornir is initialized, your automation tasks do not need to know—or care—where the data originally came from.

The goal of this chapter is *not* to teach you how to fully implement Nornir using every available inventory plugin. Instead, the intent is to introduce you to the different ways inventory data *can* be supplied to Nornir and to help you understand *why*, *when*, and *where* each option might be used.

By being exposed to these inventory sources—such as YAML files, DictInventory (Python dictionaries), NetBox, Ansible inventories, Excel or CSV spreadsheets, and modern source-

of-truth platforms (InfraHub)—you can begin to recognize which approaches are appropriate for different environments. The examples in this chapter focus on concepts and use cases rather than complete implementations, so you can begin to make informed decisions as your automation needs and network scale grow.

A Note on Filtering

One of the most powerful aspects of Nornir’s inventory management system is the ability to filter devices based on inventory data. Filtering allows you to select specific subsets of hosts—such as core routers, devices at a particular site, or hosts with a certain role—using the information defined in the inventory (hosts, groups, and defaults).

Because inventory data describes what devices exist and what characteristics they have, filtering is the primary way inventory is put into action. Rather than operating on all devices, Nornir uses inventory attributes to determine which devices a task should run on.

For example, imagine a network with dozens of devices spread across multiple locations. Using inventory-based filtering, you could:

- Run a configuration backup only on core routers.
- Apply a change only to devices at the San Jose site.
- Collect interface information from access switches but not routers.

Instead of writing separate programs or manually selecting devices, you describe the devices you want based on inventory attributes, and Nornir takes care of the rest.

In this chapter, we introduce the idea of filtering as it relates to inventory data and show how inventory attributes can be used to target specific devices.

Inventory Management Core Architecture

In [Chapter 9](#), we introduced Nornir’s inventory files—`hosts.yaml`, `groups.yaml`, and `defaults.yaml`—and showed how they work together at a basic level to load device information into Nornir.

Now we will take the next step. In this chapter, we will focus on how Nornir internally organizes inventory data; how inheritance works across hosts, groups, and defaults; and how this structured data can later be used for filtering, decision making, and automation logic inside your Python programs. Think of this section as moving from “what the files look like” to “how Nornir actually thinks about your network.”

The inventory examples in this chapter—such as `router1`, `switch1`, and site or role labels—may look familiar from earlier chapters. However, these examples are not tied to any specific network topology. Their purpose is not to represent a complete or realistic design but to illustrate how Nornir’s inventory model works.

The goal of this chapter is to help you understand the *structure* and *behavior* of inventory management in Nornir, independent of network size or layout. The same inventory concepts apply whether you are managing a small lab with a few devices or a large enterprise network with hundreds or thousands of nodes.

Hosts

At the most basic level, a host in Nornir represents one network device—such as a router, switch, firewall, or load balancer. If you think of Nornir’s inventory as a spreadsheet or database, each host is equivalent to one row that describes a single device.

In [Chapter 9](#), you learned about the `hosts.yaml` file and how Nornir reads it during initialization. In this section, we go a step further and look at how host data becomes accessible inside Python and why each field exists from an automation perspective.

Each host entry typically includes:

- A name that is used internally by Nornir
- A hostname or IP address that is used to connect to the device
- A platform that identifies the device operating system
- Optional connection settings, such as port or credentials
- One or more groups the device belongs to
- Optional custom data used for filtering and logic

Setup: Accessing a Host in Python

Before looking at individual fields, consider this simple setup that we’ll use throughout this section:

```
from nornir import InitNornir

nr = InitNornir(config_file="config.yaml")
host = nr.inventory.hosts["router1"]
```

At this point, Nornir has been initialized, and it has read and merged `hosts.yaml`, `groups.yaml`, and `defaults.yaml`. **host** is

a *host object* (variable) that contains all the inventory data for router1.

Note

Throughout this chapter, when you see references to **host** or **task.host**, they refer to the same internal host object created by Nornir during initialization. This is the Python object mentioned earlier that contains all inventory data for a device.

[Example 12-1](#) defines the hosts.yaml files for two devices: a router and a switch. [Example 12-2](#) defines config.yaml, [Example 12-3](#) defines groups.yaml, and [Example 12-4](#) defines defaults.yaml for completeness.

Example 12-1 *hosts.yaml*

```
router1:
  hostname: 192.168.1.1
  platform: ios
  groups:
    - cisco_devices
    - core_routers
  data:
    site: san_jose
    role: core

switch1:
  hostname: 192.168.1.10
  platform: nxos
  groups:
    - cisco_devices
    - access_switches
  data:
```

```
site: san_jose
role: access
```

Example 12-2 *config.yaml*

```
inventory:
  plugin: SimpleInventory
  options:
    host_file: "hosts.yaml"
    group_file: "groups.yaml"
    defaults_file: "defaults.yaml"

runner:
  plugin: threaded
  options:
    num_workers: 5
```

Example 12-3 *groups.yaml*

```
cisco_devices:
  data:
    vendor: cisco

core_routers:
  data:
    layer: core
    device_type: router

access_switches:
  data:
    layer: access
    device_type: switch
```

Example 12-4 *defaults.yaml*

```
username: admin
password: cisco
connection_options:
  napalm:
    extras:
      optional_args:
        secret: cisco
```

Let's walk through each part of a host definition and see how it is used in Python.

Host Name (Nornir Device Identifier)

The top-level keys (router1, switch1) are the hostnames that Nornir uses internally. They do not need to match the device's actual hostname; they simply act as unique identifiers. Think of these names as labels you assign to devices so Nornir can keep track of them internally.

Here's how you access the host in Python:

```
print(host.name)
```

Note

Reminder that **host** is the host object, and **name** is an attribute of that host object. You will see this syntax throughout this chapter.

This is the output:

```
router1
```

You'll use this **host** name when referencing devices in tasks, filtering inventory, and accessing inventory entries (for

example, `nr.inventory.hosts["router1"]`)

Hostname (IP Address or DNS Name)

The **hostname** field specifies how Nornir connects to the device. This is typically an IP address, but it can also be a DNS name if name resolution is available.

Here's how you access the hostname in Python:

```
print(host.hostname)
```

This is the output:

```
192.168.1.1
```

When you use connection-based plugins such as Netmiko or NAPALM, this is the address they use to establish the SSH connection—unless it is overridden by more specific settings in groups or defaults.

It's important not to confuse **host.name** with the **host.hostname**:

- **host.name:** This is the internal Nornir identifier (for example, `router1`).
- **host.hostname:** This is the actual IP address or DNS name used to connect to the device (for example, `192.168.1.1` or `router1.cisco.com`).

In other words, **host.name** answers “What do I call this device inside Nornir?” and **host.hostname** answers “How does Nornir connect to this device?”

Platform (Device Operating System)

The **platform** field tells Nornir what type of device it is working with, such as Cisco IOS, Cisco NX-OS, Juniper Junos, or Arista EOS.

Here's how you access the platform in Python:

```
print(host.platform)
```

This is the output:

```
ios
```

This value becomes especially important when using plugins like Netmiko or NAPALM, which rely on the platform to select the correct driver or connection behavior.

Port (SSH Connection Port)

The **port** field specifies the TCP port used for SSH connections. In the hosts.yaml file shown in [Example 12-1](#), no port is explicitly defined, so Nornir automatically uses the default SSH port, which is 22. This value typically comes from defaults.yaml or from Nornir's built-in defaults when no override is provided.

Here's how you access the port in Python:

```
print(host.port)
```

This is the output:

```
22
```

This behavior allows you to avoid repeating common values—such as the SSH port—for every device in your inventory.

Username and Password (Credentials)

Credentials can be defined at multiple levels in the inventory. In our example, username and password are not defined in `hosts.yaml`, so they are inherited from `defaults.yaml`. Credentials may be defined at the host level (`hosts.yaml`) or group level (`groups.yaml`), or they may be inherited from defaults (`defaults.yaml`).

Here's how you access credentials in Python:

```
print(host.username)
print(host.password)
```

Here is an example of the output you might get:

```
admin
cisco
```

These values are used by connection plugins such as Netmiko and NAPALM when authenticating to devices. Host-level credentials, if defined, would override group and default values.

Note

If no username or password is defined in the inventory, Nornir does not prompt the user. Instead, any connection attempts fails.

Groups (Group Membership)

The **groups** field associates a host with one or more groups defined in `groups.yaml`.

Here's how you access groups in Python:

```
print(host.groups)
```

Here's how you display just the group names:

```
for group in host.groups:  
    print(group.name)
```

Here is an example of the output you might get:

```
cisco_devices  
core_routers
```

Group membership is what allows hosts to inherit shared settings such as platform defaults, credentials, or metadata.

Data (Custom Metadata)

The **data** section is a free-form dictionary used to store any information that is useful for automation logic, such as site, role, environment, or ownership. At runtime, these values are stored on the host object that Nornir creates for each device; they are accessed through the **host** variable in Python.

Note

These values are accessed using square brackets (for example **host["site"]**) because they are stored as a key in the free-form data dictionary rather than as a built-in attribute like **hostname** or **platform**.

Here's how you access custom data in Python:

```
print(host["site"])  
print(host["role"])
```

This is the output:

Unlike fields such as **hostname** or **platform**, the keys under **data** are completely up to you. Common examples include the following:

- Site or location
- Device role
- Environment (production, lab, or test)
- VLANs, models, or ownership information

Later in this chapter, you will see that these values become extremely powerful because they can be used for filtering and for conditional logic inside tasks. For example, you might run different logic for core routers than for access switches based on the value of a role.

Why Hosts Matter

The `hosts.yaml` file answers the fundamental questions "What devices exist, and what do I know about each one?"

Each host entry defines a single device and provides the raw information that Nornir uses to make automation decisions. Once loaded, this data becomes part of a structured Python object that can be queried, filtered, and used inside tasks—without requiring you to hard-code values into your programs.

Summary Example

The following short example demonstrates how host inventory data can be accessed and used together in Python:

```
print(f"Device name: {host.name}")
print(f"IP address: {host.hostname}")
print(f"Platform: {host.platform}")
print(f"Site: {host['site']}")
print(f"Role: {host['role']}")
```

This example highlights an important idea: Inventory data is live Python data. Rather than embed device-specific details into your code, you describe your network once in the inventory and let Nornir handle the rest.

Groups

Whereas hosts describe individual devices, groups describe what devices have in common. Groups allow you to organize hosts based on shared characteristics and define values once rather than repeating them for every device.

You can group devices in a number of ways. These are some of the common ones:

- Geographic location (for example, `san_jose`, `new_york`, `emea`)
- Device type (for example, routers, switches, firewalls)
- Vendor or platform (for example, Cisco, Juniper, Arista)
- Role or function (for example, core, distribution, access)

If you think of the inventory as a spreadsheet, groups are similar to shared columns or categories that apply to many rows at once.

How Groups Work in Nornir

Groups are defined in the `groups.yaml` file. When a host belongs to a group, it inherits all variables defined in that

group. Groups (called nested groups) can also inherit from other groups, and any value defined at the host level will override values inherited from groups. In other words, inheritance follows this order:

- Hosts (highest priority)
- Groups
- Defaults (lowest priority)

Remember: If a value is not defined on the host (hosts.yaml), Nornir checks the groups in groups.yaml and inherits the value, if it is present.

[Example 12-5](#) defines three groups: one common Cisco group (**cisco_devices**) and two role-based groups (**core_routers** and **access_switches**).

Example 12-5 *groups.yaml*

```
cisco_devices:
  platform: ios
  data:
    vendor: cisco

core_routers:
  data:
    priority: high
    backup_frequency: daily

access_switches:
  data:
    priority: medium
    backup_frequency: weekly
```

Let's walk through what each group defines and how those values are used.

The `cisco_devices` Group

The **`cisco_devices`** group defines values that apply to all Cisco devices:

```
cisco_devices:
  platform: ios
  data:
    vendor: cisco
```

This group identifies the following settings:

- **platform: ios:** This value becomes the default platform for all hosts in the **`cisco_devices`** group. Any host that does not explicitly define its own platform in `hosts.yaml` will inherit this value.
- **vendor: cisco (a custom data field):** This value is defined under the **data** section, which means it is a custom, user-defined variable. Just like the **data** section in `hosts.yaml`, the **data** section in `groups.yaml` allows you to define any keys that are meaningful for your environment. In this case, all hosts in the group inherit the custom variable **vendor**, which is **cisco**.

Here's how you access group-derived values in Python:

```
print(host.platform)
print(host["vendor"])
```

Here is an example of output for `router1`:

```
ios
cisco
```

If a host explicitly defines its own platform in hosts.yaml (as switch1 does with **nxos** in [Example 12-1](#)), that host-level value overrides the group value.

It is important to note the difference between these two values. **platform** is a built-in inventory attribute that Nornir and its plugins use to determine how to communicate with a device, whereas **vendor** is a user-defined variable that is stored in the free-form data dictionary. Because **vendor** is custom data, it is accessed using square brackets (**host["vendor"]**) rather than dot notation.

Because router1 and switch1 in hosts.yaml both belong to the **core_devices** group, these values are automatically available for both devices inside tasks and for filtering.

The core_routers Group

The **core_routers** group defines role-specific operational metadata for devices that function as core routers in the network.

```
core_routers:
  data:
    priority: high
    backup_frequency: daily
```

This group identifies the following settings:

- **priority: high (a custom data field):** This value indicates that devices in this group have a higher operational importance. It can be used to prioritize tasks such as monitoring, configuration checks, or backups.
- **backup_frequency: daily (a custom data field):** This value specifies how often configuration backups should be performed for devices in this group.

Both values are defined under the **data** section, which means they are custom, user-defined variables. As with other group data, these values are inherited by all hosts that belong to the **core_routers** group.

Here's how you access inherited group data in Python:

```
print(host["priority"])  
print(host["backup_frequency"])
```

Here is an example of output for router1:

```
high  
daily
```

Because router1 in hosts.yaml belongs to the **core_routers** group, these values are automatically inherited by those hosts and are available inside tasks and for filtering.

The access_switches Group

The **access_switches** group defines different operational characteristics for access-layer switches, which typically have different requirements than core devices:

```
access_switches:  
  data:  
    priority: medium  
    backup_frequency: weekly
```

This group identifies the following settings:

- **priority: medium (a custom data field):** This value reflects a lower operational priority compared to core devices, which may influence how frequently tasks are run or how issues are escalated.

- **backup_frequency: weekly (a custom data field):**

This value indicates that configuration backups for access switches should be performed less frequently than for core routers.

As with other group-level data, these values are inherited by all hosts that are members of the **access_switches** group.

Here's how you access inherited group data in Python:

```
print(host["priority"])
print(host["backup_frequency"])
```

Here is an example of output for switch1:

```
medium
weekly
```

Because switch1 in hosts.yaml belongs to the **access_switches** group, these values are automatically inherited by those hosts and are available inside tasks and for filtering. These inherited values allow Nornir tasks to automatically adapt their behavior based on device role—without requiring separate programs for routers and switches.

Effective Values After Inheritance

Once Nornir loads the inventory files—hosts.yaml, groups.yaml, and defaults.yaml—it does not treat them as separate sources of information. Instead, Nornir merges all applicable values into a single, unified view for each host. This unified view is stored in the **host** object (discussed previously) that Nornir creates for each device, and it represents the effective inventory that your automation tasks actually interact with. In other words, when a task runs, it does not need to know *where* a value was defined—

whether in hosts, groups, or defaults. It simply accesses the final value that results from Nornir's inheritance rules. Let's look at how this works in practice.

Effective Values for router1

The host router1 belongs to both the **cisco_devices** and **core_routers** groups and also defines its own host-specific data. After inheritance is applied, Nornir builds the following effective set of values:

- **Connection and identity fields:** These fields define how Nornir identifies the device and how it connects to it, including values such as the hostname, IP address, platform, and connection parameters.
 - **name:** router1
 - **hostname:** 192.168.1.1
 - **platform:** ios (explicitly defined at the host level and also matches the value inherited from **cisco_devices**)
- **Inherited group data:** These values are inherited from one or more groups that the host belongs to and typically represent shared characteristics or operational settings that are common to multiple devices.
 - **vendor:** cisco (from cisco_devices)
 - **priority:** high (from core_routers)
 - **backup_frequency:** daily (from core_routers)
- **Host-defined data:** These values are defined directly in the host's entry in hosts.yaml and override any conflicting values inherited from groups or defaults.
 - **site:** san_jose

- **role:** core

All of these values—regardless of where they originated (that is, in which YAML file)—are now available through the **task.host** object when a task runs.

Here's how you access these values in a Nornir task:

```
task.host.hostname          # 192.168.1.1
task.host.platform          # ios
task.host["vendor"]         # cisco
task.host["priority"]       # high
task.host["backup_frequency"] # daily
task.host["site"]           # san_jose
task.host["role"]           # core
```

Effective Values for switch1

For switch1, the inheritance works the same way, with one important difference: **platform** is overridden at the host level.

- **Connection and identity fields**
 - **name:** switch1
 - **hostname:** 192.168.1.10
 - **platform:** nxos (overrides ios from **cisco_devices**)
- **Inherited group data**
 - **vendor:** cisco (from cisco_devices)
 - **priority:** medium (from access_switches)
 - **backup_frequency:** weekly (from access_switches)
- **Host-defined data**
 - **site:** san_jose

- **role:** access

Once again, all these values are merged into a single effective view for the host that is available through the **task.host** object.

Here's how you access these values in a task:

```
task.host.hostname          # 192.168.1.10
task.host.platform          # nxos
task.host["vendor"]         # cisco
task.host["priority"]       # medium
task.host["backup_frequency"]# weekly
task.host["site"]           # san_jose
task.host["role"]           # access
```

Why Effective Values Matter

Understanding effective values is critical because this is the only inventory view that tasks ever see. Your Python code does *not* need to check whether a value came from a host, a group, or defaults; it simply uses the final result. This design allows you to:

- Define shared behavior once at the group level
- Override values only where necessary
- Write automation tasks that adapt automatically based on inventory data

In practice, this means your automation logic can focus on what needs to be done; Nornir handles where the data comes from.

Using Group Data for Filtering

Once inventory data is loaded and inheritance has been applied, Nornir allows you to select specific subsets of devices based on that data, using Nornir's **filter()** method. The filtering process is one of the most common ways inventory is used in automation.

Filtering does not modify the inventory itself. Instead, it creates a new Nornir object (for example, **cisco_filter**, **core_filter**, **high_priority_filter**) that contains only the hosts that match the filter criteria. Any tasks you run against this filtered object will execute only on those devices.

Example 12-6 *Filtering in Nornir*

```
# All Cisco devices
cisco_filter = nr.filter(vendor="cisco")

# Only core routers
core_filter = nr.filter(role="core")

# Only access switches
access_filter = nr.filter(role="access")

# Devices with high priority
high_priority_filter = nr.filter(priority="high")

# Devices at the San Jose site
sj_filter = nr.filter(site="san_jose")
```

Each of the filters in [Example 12-6](#) selects devices based on values defined in `hosts.yaml` or inherited from `groups.yaml`. For example, using values defined in `hosts.yaml`, the **core_filter** object contains only hosts where the following data is defined:

```
data:
  role: core
```

Inspecting a Filtered Inventory

The important takeaway for filtering is that it gives you a way to select the right devices, and Nornir handles the complexity of inventory merging for you behind the scenes.

To make this more concrete, you can inspect which devices are included in a filtered inventory. The following code prints the name and role of each device in **core_filter**:

```
for name, host in core_filter.inventory.hosts.items():
    print(f"Device: {name}")
    print(f"  Hostname: {host.hostname}")
    print(f"  Role: {host['role']}")
```

Here is an example of output from this code:

```
Device: router1
  Hostname: 192.168.1.1
  Role: core
```

When this code runs, Nornir is no longer working with the specific YAML files. This is what is happening in the code:

Step 1. Nornir loads inventory data. It reads `hosts.yaml`, `groups.yaml`, and `defaults.yaml`. It applies inheritance and overrides. It builds an internal inventory made up of host objects.

Step 2. Filtering creates a new Nornir object, **core_filter** = **nr.filter(role="core")**, from [Example 12-5](#). This new object contains only the matching host objects.

Step 3.`hosts.items()` iterates over host objects.
`core_filter.inventory.hosts` is a dictionary of host objects. The keys are hostnames (such as `router1`), and the values are fully merged host objects (not YAML entries).

So, when you see this code, you are iterating over Nornir's internal inventory, not over the contents of `hosts.yaml`:

```
for name, host in core_filter.inventory.hosts.items():
```

Although the original inventory data comes from multiple files (`hosts.yaml`, `groups.yaml`, and `defaults.yaml`), the **`hosts.items()`** call iterates over Nornir's internal host objects, which already contain the fully merged and inherited inventory data.

Filtering is driven entirely by inventory metadata, not by where devices appear in a file or how many devices exist. As your inventory grows, this approach scales naturally. You simply update inventory data, and your filters continue to work even though you don't have to change any Python code.

Task Logic

In addition to filtering which devices a task runs on, inventory data can also influence how a task behaves on each device. This is often referred to as *task logic*—the decisions Python code makes while a task is running, such as performing extra steps on core devices or running actions more frequently on high-priority systems.

We do not explore task logic in detail here; you just need to understand that inventory data can drive both device selection and device-specific behavior within a single task.

Defaults

Defaults are the final layer in Nornir's inventory hierarchy, providing global fallback values that apply to all hosts unless overridden by group or host definitions. Because they have the lowest priority, defaults are typically used for shared values like credentials or connection parameters, avoiding repetition across the inventory.

[Example 12-7](#) shows a simple example of a defaults.yaml file.

Example 12-7 *defaults.yaml*

```
username: admin
password: cisco
port: 22
data:
  domain: cisco.com
  ntp_server: 10.0.0.1
```

Based on these default values, every host will use the username admin and password cisco when establishing SSH connection over TCP port 22. All the hosts will also have the domain [cisco.com](#) and NTP server 10.0.0.1 available in **task.host["domain"]** and **task.host["ntp_server"]**.

Note

Remember that, as with group-level data, custom values defined under **data** in defaults.yaml are accessed using square brackets (for example, **task.host["domain"]**).

Now that we have seen the contents of all the inventory files—hosts.yaml, groups.yaml, and defaults.yaml—let's

take a look at how router1 looks like from Nornir's point of view. These are the connection and identity variables:

- **name:** router1(from host data)
- **hostname:** 192.168.1.1 (from host data)
- **platform:** ios (from host data)
- **username:** admin (from defaults)
- **password:** cisco (from defaults)
- **port:** 22 (from defaults)

The following data can be accessed through **task.host["..."]**:

- **domain:** [cisco.com](https://www.cisco.com) (from defaults)
- **ntp_server:** 10.0.0.1 (from defaults)
- **vendor:** cisco (from **cisco_devices** group)
- **priority:** high (from **core_routers** group)
- **backup_frequency:** daily (from core_routers group)
- **site:** san_jose (from host data)
- **role:** core (from host data)

For switch1 we have the following connection and identity variables:

- **name:** switch1 (from host data)
- **hostname:** 192.168.1.10 (from host data)
- **platform:** nxos (the value of the platform from hosts.yaml overrides the **cisco_devices** group value)
- **username:** admin (from defaults)
- **password:** cisco (from defaults)
- **port:** 22 (from defaults)

The following data can be accessed through **task.host["..."]**:

- **domain:** [cisco.com](https://www.cisco.com) (from defaults)
- **ntp_server:** 10.0.0.1 (from defaults)
- **vendor:** cisco (from the **cisco_devices** group)
- **priority:** medium (from the **access_switches** group)
- **backup_frequency:** weekly (from the **access_switches** group)
- **site:** san_jose (from host data)
- **role:** access (from host data)

You should now have a good understanding of Nornir's inventory architecture based on hosts, groups, and defaults. This hierarchical model is a fundamental building block of Nornir and remains the same regardless of which inventory plugin you use—whether it is the built-in SimpleInventory plugin shown in earlier chapters or third-party integrations such as NetBox or Ansible. (SimpleInventory, NetBox, and Ansible are discussed later in this chapter.)

Accessing Inventory Data

At this point in the chapter, Nornir has already done the heavy lifting for you, creating internal host objects that represent network devices. Once this process is complete, your Python code no longer interacts with YAML files directly. Instead, it works with these host objects, which provide a structured and consistent way to access inventory data from hosts, groups, and defaults.

[Example 12-8](#) shows how to initialize Nornir and retrieve a specific host from the inventory.

Example 12-8 *Host objects*

```
from nornir import InitNornir

nr = InitNornir(config_file="config.yaml")

# Access a specific host
host = nr.inventory.hosts["router1"]

# Access host attributes
print(host.hostname)      # 192.168.1.1
print(host.platform)      # ios
print(host["site"])       # san_jose
print(host.groups)        # List of group objects

# Access inherited data
print(host["vendor"])     # Inherited from group: cisco
```

Here, **host** is a host object created by Nornir during initialization. It contains the fully merged inventory data for router1, including values defined directly on the host, values inherited from groups, and values inherited from defaults.

Accessing Built-in Host Attributes

Inventory values with well-known keys—such as **hostname**, **platform**, and **groups**—are exposed as built-in attributes of the host object (**host**), and you access them by using dot notation:

```
print(host.hostname)      # 192.168.1.1
print(host.platform)      # ios
print(host.groups)        # List of group objects
```

These attributes describe how Nornir identifies the device and how it connects to it.

Accessing Custom Inventory Data

Custom inventory values—values defined under the **data** section in `hosts.yaml`, `groups.yaml`, or `defaults.yaml`—are stored as user-defined keys in a free-form dictionary on a host object. Because they are not built-in host attributes, they are accessed using dictionary-style syntax rather than dot notation:

```
print(host["site"])          # san_jose
print(host["vendor"])        # Inherited from group: cisco
```

These values are entirely user defined and can represent any metadata that is useful for automation logic, such as site, role, environment, ownership, or priority.

Why This Matters

Being able to access inventory data in this way is what makes Nornir automation flexible and scalable. Instead of hard-coding device details into your Python programs, you describe your network once in the inventory and let Nornir provide that information wherever it is needed. This same access pattern is used throughout Nornir—for filtering devices, making decisions inside tasks, and adapting automation behavior as your network grows.

Inventory files describe your network, but host objects are what your Python code actually works with. When Nornir runs a task, each device in the inventory is represented as a host object, which provides structured access to inventory data such as hostnames, platform details, credentials, groups, and custom metadata.

Inventory Plugins

Up to this point in the chapter, we have focused on how Nornir organizes inventory data into hosts, groups, and defaults and how that data is used once Nornir is running. The next question is “Where does that inventory data come from?”

Nornir uses inventory plugins to answer this question. An inventory plugin is responsible for loading device information from some source—such as YAML files, a database, or an external system—and converting it into Nornir’s internal inventory model. This separation allows Nornir to remain flexible and adaptable to many different environments.

Why We Need Inventory Plugins

Why are inventory plugins necessary? Networks rarely store inventory information in the same way. Some teams keep device details in YAML files (as we have been discussing so far), others use spreadsheets, and larger organizations often rely on centralized systems such as NetBox, InfraHub, or Ansible inventories. Inventory plugins allow Nornir to work with all of these approaches without changing how tasks, filters, or automation logic are written. In other words, plugins let you change where inventory data comes from without changing how you use it.

Why are there different inventory plugin options? Different inventory plugins exist because different environments have different needs. Consider these examples:

- Small labs or learning environments often use static YAML files.

- Environments using automation workflows may generate inventory data dynamically in Python.
- Some teams manage inventory in spreadsheets shared across departments.
- An enterprise typically maintains a centralized source of truth such as Ansible, NetBox, or InfraHub.

Rather than provide only one option with a single format, Nornir provides multiple inventory plugins so you can choose the one that best fits how your organization already works.

Where Inventory Plugins Are Configured

Inventory plugins are configured when Nornir is initialized—either directly in Python using **InitNornir()** or indirectly through a config.yaml file. In both cases, you *explicitly specify which inventory plugin to use* and provide any required options, such as file locations or API credentials. Once Nornir is initialized, the rest of your code interacts with inventory data in the same way, regardless of which plugin was used.

Built-in Versus Third-Party Inventory Plugins

Nornir provides two categories of inventory plugins:

- **Built-in plugins:** These plugins are included with Nornir itself and require no additional installation. They are typically used for learning, testing, or environments where inventory data is simple and self-contained.

- **Third-party plugins:** These plugins integrate Nornir with external systems and must be installed separately. They are commonly used in production environments where inventory data already exists in tools such as NetBox, InfraHub, Ansible, or spreadsheets.

Inventory Plugin Options

Nornir supports the following inventory plugins:

- Built-in plugins
- SimpleInventory
- DictInventory
- Third-party plugins
- NetBox
- Ansible inventory
- Table inventory (CSV/Excel spreadsheets)
- InfraHub

In the sections that follow, we'll briefly explore each of these inventory plugins and discuss when you might choose one over another, focusing on concepts rather than implementation details.

The SimpleInventory Plugin

The SimpleInventory plugin is Nornir's built-in inventory plugin and the plugin you will use most commonly as you are getting started with Nornir. This is the plugin we have been using so far. It is designed to load inventory data from YAML files—specifically `hosts.yaml`, `groups.yaml`, and `defaults.yaml`.

Configuring SimpleInventory with config.yaml

Inventory plugins are configured as part of Nornir's initialization, typically using a config.yaml file. When working with YAML inventory files, the config.yaml file explicitly tells Nornir to use the SimpleInventory plugin and where to find the inventory data.

[Example 12-9](#) shows a typical config.yaml file using SimpleInventory.

Example 12-9 *config.yaml Using SimpleInventory*

```
inventory:
  plugin: SimpleInventory
  options:
    host_file: "inventory/hosts.yaml"
    group_file: "inventory/groups.yaml"
    defaults_file: "inventory/defaults.yaml"

runner:
  plugin: threaded
  options:
    num_workers: 10
```

With this configuration in place, Nornir knows to read inventory data from the three YAML files and load it using the SimpleInventory plugin.

Nornir is then initialized in Python code as follows:

```
from nornir import InitNornir

nr = InitNornir(config_file="config.yaml")
```

At this point, Nornir reads the inventory files, applies inheritance and overrides, and creates the internal host

objects that your automation code will work with.

Note

Throughout [Part 3](#), “[Nornir](#),” inventory data has been loaded from YAML files using the built-in SimpleInventory plugin, configured as part of Nornir’s initialization in config.yaml. While you may see some implementations appear to work without explicitly naming the SimpleInventory inventory plugin, this behavior should not be relied upon. For clarity, correctness, and long-term maintainability, it is considered best practice to explicitly specify the inventory plugin in config.yaml.

Configuring SimpleInventory Directly in Python

You can specify the inventory plugin directly in your Python code instead of using a config.yaml file. As you can see in [Example 12-10](#), this method produces the same result as using a config.yaml file.

Example 12-10 SimpleInventory Configured in Python

```
from nornir import InitNornir

nr = InitNornir(
    runner={
        "plugin": "threaded",
        "options": {
            "num_workers": 10
        }
    },
    inventory={
        "plugin": "SimpleInventory",
        "options": {
```

```
        "host_file": "inventory/hosts.yaml",
        "group_file": "inventory/groups.yaml",
        "defaults_file": "inventory/defaults.yaml"
    }
}
)
```

Notice that the SimpleInventory plugin can be specified directly in Python during Nornir's initialization rather than being defined in a config.yaml file. In this case, all inventory configuration information—such as the plugin name and the locations of the YAML files—is provided inline as part of the **InitNornir()** call.

Whether you configure SimpleInventory in config.yaml or directly in Python, the outcome is the same: a Nornir object (**nr**) with a fully populated inventory. Internally, Nornir creates the same host objects from the YAML files in both cases, meaning your tasks, filters, and automation logic do not need to change based on how SimpleInventory is configured.

When to Use SimpleInventory

SimpleInventory is best suited for:

- Learning and experimenting with Nornir
- Labs and test environments
- Small to medium networks with relatively static inventory
- Situations where you want full control over inventory data in simple YAML files

As environments grow or inventory data moves into external systems, other inventory plugins—such as NetBox or Ansible

integrations—may be more appropriate. We'll explore those options later in this chapter.

The DictInventory Plugin

The DictInventory plugin is another built-in inventory plugin provided by Nornir. Instead of loading inventory data from YAML files, DictInventory allows you to define the entire inventory directly in Python by using dictionaries, as shown in [Example 12-11](#).

Conceptually, DictInventory works exactly the same as SimpleInventory: You still describe hosts, groups, and defaults.

[Example 12-11](#) shows a minimal DictInventory configuration. Don't worry about the details of the code; the key idea is that the inventory data is defined as key/value pairs in Python dictionaries instead of YAML files.

Example 12-11 *Using DictInventory*

```
from nornir import InitNornir

inventory_dict = {
    "hosts": {
        "router1": {
            "hostname": "192.168.1.1",
            "platform": "ios"
        }
    },
    "groups": {},
    "defaults": {}
}
```

```
nr = InitNornir(
    inventory={
        "plugin": "DictInventory",
        "options": {
            "hosts": inventory_dict["hosts"],
            "groups": inventory_dict["groups"],
            "defaults": inventory_dict["defaults"]
        }
    }
)

# Access the host object created by Nornir
host = nr.inventory.hosts["router1"]

print(host.hostname)    # 192.168.1.1
print(host.platform)    # ios
```

Notice in [Example 12-11](#) that the inventory plugin is set to `DictInventory`. The **options** section contains three keys—**hosts**, **groups**, and **defaults**—which correspond to the three layers of Nornir’s inventory model. Each of these keys is populated using values from the **inventory_dict** Python dictionary.

In this example, the inventory data is created entirely in Python and passed directly to Nornir during initialization. Internally, Nornir converts this data into the same host objects you’ve seen throughout the chapter.

Although the inventory data is initially defined in a Python dictionary (**inventory_dict**), your automation code does not interact with that dictionary directly. Once Nornir is initialized, it converts the inventory data into internal host objects (**host.hostname** and **host.platform**). From that

point on, inventory values are accessed through the host object in the same way as with YAML-based inventories.

In other words, DictInventory changes *where* inventory data comes from—not *how* your automation code uses it.

DictInventory Versus SimpleInventory

It can be helpful to think of the difference between SimpleInventory and DictInventory in terms of *where the inventory data lives*:

- With SimpleInventory, inventory information is stored in YAML files on disk. These files are easy for humans to read and edit, work well with version control systems such as Git, and are ideal for learning, labs, and environments where the inventory changes infrequently.
- With DictInventory, the inventory data lives entirely in Python dictionaries. Instead of being read from files, the inventory is constructed dynamically by code at runtime. This makes DictInventory a better fit when inventory data is generated programmatically, retrieved from another system, or assembled as part of an automation workflow.

DictInventory does not change how Nornir works; it changes *how inventory data is supplied*. If your inventory already exists as structured data in Python, DictInventory lets you use it directly. If your inventory is static and human managed, SimpleInventory is usually the better choice.

Despite these differences, it's important to remember that once Nornir is initialized, both approaches produce the same internal inventory model. Your automation code interacts with hosts, groups, and defaults in exactly the same way, regardless of whether the inventory originated from YAML files or Python dictionaries.

When DictInventory Is Useful

DictInventory is most useful when inventory data is:

- Generated dynamically at runtime
- Retrieved from another system (such as an API, a database, or a discovery process)
- Built programmatically as part of your automation logic

Rather than write or update YAML files on disk, your Python program constructs the inventory data in memory and passes it directly to Nornir. This makes DictInventory a good choice for:

- Dynamic or temporary inventories
- Proof-of-concept automation
- Environments where inventory changes frequently
- Situations where inventory data already exists in Python data structures

The NetBox Plugin

NetBox is a popular open-source network source of truth (NSoT) that is used to store and manage information about network infrastructure. It provides a centralized place to track devices, IP addresses, sites, roles, platforms, and other metadata that describes a network.

In practical terms, NetBox is server-based software that provides a web interface, a REST API, and a backend database (typically PostgreSQL) for storing network inventory data.

NetBox does not automate network devices directly. Instead, it provides the authoritative data that automation tools like

Nornir rely on to know what devices exist and how they should be treated.

Many organizations use NetBox as the authoritative system for answering questions such as:

- What devices exist in the network?
- Where are they located?
- What role does each device play?
- What IP address and platform does it use?

NetBox makes this information available through a REST API, which makes it ideal for integration with automation tools like Nornir.

Why Use NetBox with Nornir?

When using SimpleInventory or DictInventory, you manually define inventory data using YAML files or Python dictionaries. This works well for learning, labs, and smaller environments, but it does not scale easily as networks grow.

NetBox integration allows Nornir to pull inventory data dynamically from an existing system of record, a NSoT, instead of maintaining separate inventory files. This provides several advantages:

- Inventory data is stored in one central location.
- Changes made in NetBox are automatically reflected in Nornir.
- Hosts and groups can be generated automatically.
- It reduces duplication and configuration drift.

Relying on a single, authoritative source for device information means that automation tools and documentation remain aligned with the actual network,

helping prevent systems from slowly drifting out of sync. In short, NetBox becomes the source of inventory truth, and Nornir simply uses that data.

How Nornir Uses NetBox

The NetBox inventory plugin for Nornir is called `nornir_netbox`. Its purpose is to query the NetBox API, retrieve device information, and populate Nornir's internal inventory model. Importantly, *Nornir's inventory structure does not change*. Even when using NetBox, Nornir still builds hosts, groups, and defaults. The plugin simply fills those structures using data retrieved from NetBox instead of YAML files.

Conceptually, each NetBox device becomes a Nornir host object, with fields such as:

- **name:** The NetBox device name
- **hostname:** Typically the device's primary IP address
- **platform:** The NetBox platform
- **data:** Additional metadata, such as site, role, tags, or custom fields

Once Nornir is initialized, your automation code interacts with inventory exactly the same way as before.

Installing and Configuring the NetBox Plugin

The NetBox inventory plugin is not included with Nornir and must be installed separately, using the following command:

```
pip install nornir_netbox
```

To use NetBox as an inventory source, you configure Nornir to use the NetBox inventory plugin in `config.yaml`, as shown in [Example 12-12](#).

Example 12-12 *config.yaml* Using NetBox

```
inventory:
  plugin: nornir_netbox.plugins.inventory.NetBoxInventory
  options:
    nb_url: "https://netbox.example.com"
    nb_token: "YOUR_NETBOX_API_TOKEN"
    ssl_verify: true

runner:
  plugin: threaded
  options:
    num_workers: 10
```

This configuration tells Nornir to:

- Connect to the NetBox API
- Authenticate to NetBox using an API token
- Dynamically build the inventory at runtime, using information stored in the NetBox backend database

After you define `config.yaml`, Nornir is initialized the same way as before:

```
from nornir import InitNornir
nr = InitNornir(config_file="config.yaml")
```

Hosts, Groups, and Defaults with NetBox

When using NetBox as an inventory source, Nornir still builds the same three-layer inventory model—hosts, groups, and defaults—but the source of that data changes:

- Hosts are created from device (or virtual machine) records stored in NetBox’s backend database. Each NetBox device is retrieved through the NetBox API and

converted into a Nornir host object, using fields such as device name, primary IP address, platform, and other metadata.

- Groups are typically derived automatically from attributes associated with those devices in NetBox. For example, with Nornir, you can group together devices that share the same site, role, platform, or tags in the NetBox database without manually defining groups in a groups.yaml file.
- Defaults are usually still managed locally in Nornir. In practice, this means you often continue to use a defaults.yaml file to define global settings such as credentials, connection parameters, or network-wide values that are not stored in NetBox.

In this model, NetBox defines what devices exist and how they are categorized, while Nornir defaults provide global automation settings that apply across all devices.

The most important thing to understand is that switching to NetBox does not change how you write Nornir code; it only changes where the inventory data originates.

Accessing Inventory Data with NetBox

Once Nornir has been initialized using the NetBox inventory plugin, inventory access works exactly the same way as it does with YAML-based or dictionary-based inventories.

After initialization, Nornir has already queried the NetBox backend database via its API, applied inheritance rules, and built internal host objects. From this point forward, your automation code does not interact with NetBox directly; it interacts with Nornir's inventory model.

For example, your Python code does not need to know whether these values came from YAML files, NetBox, or any

other inventory source:

```
# Access the host object created by Nornir
host = nr.inventory.hosts["router1"]

print(host.hostname)
print(host.platform)
```

This is a key design principle in Nornir: Inventory plugins only change where the data comes from and not how your code uses it. Once Nornir is initialized, inventory access, filtering, and task execution behave consistently across all inventory plugins. This abstraction allows you to start with SimpleInventory for learning and labs and later move to NetBox in production environments without rewriting your automation logic.

What Stays the Same

One of the most important things to understand is that when you use NetBox as your inventory, nothing changes in your Nornir automation code. Filtering, inheritance, task execution, and data access all work exactly as described earlier in this chapter. Whether inventory data comes from YAML files or NetBox, Nornir presents the same internal inventory objects to your Python code.

When to Use NetBox Inventory

Using NetBox as an inventory source makes the most sense when inventory data already exists in NetBox or when inventory management is shared across teams and tools. In these environments, NetBox acts as a centralized source of truth, and Nornir simply consumes that data instead of maintaining a separate copy.

NetBox integration is particularly well suited for environments where inventory changes frequently, multiple automation tools rely on the same device data, or manually maintaining YAML inventory files would be error prone or difficult to scale. By pulling inventory directly from NetBox, Nornir can always work with up-to-date information without requiring changes to automation code.

For learning, labs, or small environments, SimpleInventory is usually sufficient and often preferable due to its simplicity and transparency. NetBox becomes increasingly valuable as environments grow, automation matures, and inventory management becomes a shared responsibility rather than a single engineer's task.

The Ansible Plugin

Ansible is a popular open-source automation platform that is used to configure systems, deploy applications, and manage network devices. Unlike Nornir, which is a Python automation framework, Ansible uses declarative configuration files called *playbooks* to describe what actions should be performed on devices.

In this section, we do not focus on how to write or run Ansible playbooks. Instead, the goal is to understand how Nornir can reuse Ansible's existing inventory data. Think of Ansible in this case as the source of device information, not as a tool for executing automation.

Many organizations already use Ansible to manage their infrastructure and have invested significant time building Ansible inventories, group variables, and host variables. These inventories often represent mature and trusted sources of network information.

Why Use Ansible Inventory with Nornir?

The `nornir_ansible` plugin allows Nornir to reuse an existing Ansible inventory instead of requiring you to re-create that inventory in YAML or Python dictionaries. This is especially useful when:

- Ansible is already in use.
- Inventory data is well maintained in Ansible.
- Multiple teams rely on the same inventory.
- You want to adopt Nornir gradually without disrupting existing workflows.

In short, `nornir_ansible` lets you keep Ansible as the source of truth for inventory and use Nornir for Python-based automation.

How Ansible Inventory Works with Nornir

When using the Ansible inventory plugin, Nornir does not execute Ansible playbooks. Instead, it reads Ansible's inventory files and variables and converts them into Nornir's internal inventory model:

- Ansible hosts become Nornir hosts.
- Ansible groups become Nornir groups.
- Ansible variables become Nornir host or group data.

Once this conversion happens, Nornir behaves exactly the same as it does with any other inventory source.

Installing and Configuring the Ansible inventory Plugin

To install the Ansible inventory plugin, you use this command:

```
pip install nornir_ansible
```

To use Ansible inventory with Nornir, you specify the Ansible inventory plugin in `config.yaml` and point it to an existing Ansible inventory file, as shown in [Example 12-13](#).

Example 12-13 *config.yaml Using Ansible inventory*

```
inventory:
  plugin: nornir_ansible.plugins.inventory.ansible.AnsibleInventory
  options:
    hostsfile: "inventory/hosts.ini"

runner:
  plugin: threaded
  options:
    num_workers: 10
```

In this configuration:

- **hosts.ini** is a standard Ansible inventory file.
- If present, Ansible's **group_vars/** and **host_vars/** directories are automatically loaded.
- Both INI and YAML Ansible inventory formats are supported.

You then initialize Nornir in the usual way:

```
from nornir import InitNornir

nr = InitNornir(config_file="config.yaml")
```

How Ansible Variables Map into Nornir

Ansible inventory fields are mapped into Nornir's inventory model using familiar concepts:

- Connection-related variables such as **ansible_host**, **ansible_user**, and **ansible_network_os** are standard Ansible inventory fields with predefined meanings. The **nornir_ansible** plugin recognizes these variables and maps them to Nornir's built-in host attributes.
- All other Ansible variables are treated as user-defined data and are stored in the host object's free-form data dictionary, which is accessible via **task.host["variable_name"]**

For example, an Ansible inventory entry such as this:

```
router1 ansible_host=192.168.1.1 ansible_network_os=ios ansible_us
```

becomes a Nornir **host** object that can be accessed as follows:

```
host = nr.inventory.hosts["router1"]

host.name          # "router1"
host.hostname      # "192.168.1.1"
host.platform      # "ios"
host.username      # "admin"

host["site"]       # "sjc"
host["role"]       # "core"
```

From this point on, filtering, task execution, and inventory access work exactly as described in earlier parts of this chapter.

When to Use Ansible inventory

Using Ansible inventory with Nornir is a good choice when Ansible is already established in your environment and serving as a trusted source of device information. In many organizations, Ansible inventories have been built and refined over time and already contain detailed, well-maintained data about hosts, groups, and variables. By integrating Nornir with Ansible inventory, you can avoid duplicating this information across multiple tools and instead reuse what already exists. This approach is especially helpful if you want to introduce Python-based automation gradually—leveraging Nornir’s flexibility and performance while continuing to rely on Ansible’s inventory management. In this way, Nornir and Ansible can coexist, each bringing its strengths without an all-at-once migration.

The Ansible inventory plugin allows Nornir to reuse existing Ansible inventory data without changing how Nornir tasks, filters, or automation logic work.

To summarize, for learning, labs, or environments without Ansible, SimpleInventory is usually the better starting point. Ansible inventory integration is most valuable in environments where Ansible is already the established system of record for inventory data.

CSV and Excel (.xlsx) Plugin

A spreadsheet is a table of rows and columns, commonly managed using tools like Microsoft Excel or Google Sheets. Many network teams already use spreadsheets to track devices, IP addresses, roles, sites, and credentials because they are easy to edit, easy to share, and familiar to both technical and nontechnical staff. In many organizations, a

spreadsheet is the original or primary source of inventory information.

The spreadsheet must be formatted (saved) as either an Excel file (.xlsx) or a CSV (comma-separated values) file; Nornir's table-based inventory plugin, `nornir_table_inventory`, can use both formats.

A CSV file is a much simpler format than an Excel file and many times the preferred format. It contains plaintext where each line represents a row, and values are separated by commas. CSV files do not support formatting, formulas, or multiple sheets; they store only raw data. Because CSV files are simple text files, they work especially well with automation tools, version control systems like Git, and scripting environments. For this reason, many automation workflows prefer CSV files even when the original data is maintained in a spreadsheet.

The `nornir_table_inventory` plugin allows Nornir to use this existing data directly, without requiring you to convert it into YAML files or write custom parsing code. Both CSV and Excel (.xlsx) formats use Nornir's table-based inventory plugin.

Why Use Spreadsheet-Based Inventory with Nornir?

Using CSV or Excel inventory makes sense when:

- Inventory data already exists in spreadsheets.
- Multiple people (not all of them programmers) maintain the inventory.
- Inventory changes frequently.
- You want to introduce automation without redesigning how inventory is stored.

Instead of asking teams to adopt a new format immediately, you can use Nornir to adapt to what already exists.

How the CSV and Excel Plugin Works

The `nornir_table_inventory` plugin reads tabular data—either from a CSV file or an Excel worksheet—and maps each row to a Nornir host object.

Each column becomes either a built-in host attribute (such as `hostname` or `platform`) or a custom data field accessible via `host["field_name"]`. Once the inventory is loaded, Nornir treats these hosts exactly the same way as if they had been defined in `hosts.yaml`. Internally, Nornir still builds the same host objects, applies the same filtering logic, and runs tasks in the same way.

How Spreadsheet Inventory Maps to Nornir

Once Nornir loads inventory data from a CSV or Excel file, each row in the table is converted into a Nornir host object, following the same inventory model used throughout this chapter. At this point, your Python code no longer interacts with the spreadsheet file directly. Instead, it works with the host objects created by Nornir.

From the earlier CSV example, consider the following row:

```
router1,192.168.1.1,ios,"cisco,routers",san_jose,core,admin,cisco1
```

This becomes a Nornir host that can be accessed in Python as follows:

```
host = nr.inventory.hosts["router1"]

# Built-in host attributes
print(host.name)          # router1
print(host.hostname)      # 192.168.1.1
```

```
print(host.platform)    # ios
print(host.username)    # admin

# Custom data fields (from spreadsheet columns)
print(host["site"])     # san_jose
print(host["role"])     # core
```

Columns such as hostname, platform, username, and password map to built-in host attributes, while all other columns (such as site and role) are stored as custom inventory data and accessed using dictionary-style syntax.

Once the inventory is loaded, filtering, task execution, and data access behave exactly the same way as with YAML-based, NetBox-based, or Ansible-based inventories.

Installing and Configuring the Spreadsheet Plugins

To use spreadsheet-based inventory with Nornir—whether your data is stored in CSV or Excel format—you must install the `nornir-table-inventory` plugin:

```
pip install nornir-table-inventory
```

If you plan to use Excel files (.xlsx), you must also install an additional dependency:

```
pip install openpyxl
```

A simple CSV inventory file might look like this:

```
name,hostname,platform,groups,site,role,username,password
router1,192.168.1.1,ios,"cisco,routers",san_jose,core,admin,cisc01
router2,192.168.1.2,ios,"cisco,routers",new_york,core,admin,cisc01
switch1,192.168.1.10,nxos,"cisco,switches",san_jose,access,admin,c
switch2,192.168.1.11,nxos,"cisco,switches",new_york,access,admin,c
```

In this example:

- Each row represents a device.
- The name column becomes the Nornir hostname.
- Columns like hostname and platform map to built-in host attributes.
- Columns such as site and role become custom data fields that are accessible via **host["site"]** and **host["role"]**.
- The groups column assigns the host to one or more Nornir groups.

To load inventory data from a CSV file, you configure the inventory plugin in config.yaml as shown in [Example 12-14](#).

Example 12-14 *config.yaml Using CSV inventory*

```
inventory:
  plugin: nornir_table_inventory.plugins.inventory.table.TableInventory
  options:
    table_file: "inventory.csv"
    table_type: "csv"

runner:
  plugin: threaded
  options:
    num_workers: 10
```

If your inventory is stored in an Excel file, the configuration is nearly identical, as shown in [Example 12-15](#).

Example 12-15 *config.yaml Using Excel inventory*

```
inventory:
  plugin: nornir_table_inventory.plugins.inventory.table.TableInventory
  options:
    table_file: "inventory.xlsx"
    table_type: "excel"
    sheet_name: "devices" # Optional; defaults to the first sheet

runner:
  plugin: threaded
  options:
    num_workers: 10
```

When to Use CSV or Excel inventory

Using spreadsheet-based inventory with Nornir is a good choice when your device information already exists in spreadsheets and you want to leverage that data directly for automation. This approach is especially useful in environments where non-programmers—such as network engineers or operations staff—are responsible for maintaining inventory data since spreadsheets are familiar and easy to update. This approach also provides a low barrier to entry for getting started with automation, allowing teams to avoid learning new inventory formats right away. For many organizations, spreadsheet-based inventory offers a quick and practical way to onboard existing device data into Nornir without restructuring how inventory is managed.

For long-term, large-scale environments, teams often migrate from spreadsheets to centralized systems like NetBox. However, CSV and Excel inventory provide an excellent bridge between manual inventory management and automated workflows.

In summary, the CSV and Excel inventory plugin allows Nornir to meet teams where they already are. You don't need to change how inventory is stored to begin automating. Once Nornir loads the data, your automation code works the same way, regardless of whether inventory came from YAML files, spreadsheets, or an external system.

The InfraHub Plugin

InfraHub is a modern infrastructure source of truth (ISoT) platform used to store and manage information about network and infrastructure resources, such as devices, sites, IP addresses, roles, and the relationships between those objects.

Like NetBox, InfraHub is a server-based platform that provides a centralized system where infrastructure data is stored and maintained. Where InfraHub differs is in its Git-like version-controlled approach to infrastructure data. It is possible to track changes over time, review them before being applying them, and organize them using branches—in much the same way that software developers manage source code.

InfraHub is designed for environments that treat infrastructure data as code and integrate closely with automation workflows and CI/CD pipelines. CI/CD stands for continuous integration/continuous deployment, a set of practices commonly used in software development and DevOps to automatically test, validate, and apply changes in a controlled way. If concepts such as CI/CD, Git workflows, or infrastructure as code are unfamiliar, don't worry. These are DevOps and DevNet concepts that go beyond the scope of this book. What's important for now is understanding that InfraHub provides a highly structured, version-controlled

source of truth that tools like Nornir can use as an inventory source.

Why Use InfraHub with Nornir?

When InfraHub is used as the system of record for infrastructure data, Nornir can query InfraHub directly to build its inventory dynamically. Instead of maintaining `hosts.yaml` and `groups.yaml` files manually, an organization can use InfraHub as the authoritative source that defines what devices exist, how they are categorized, and what metadata is associated with them. Nornir then consumes the data and converts it into its familiar inventory model.

The key idea is the same as with NetBox: InfraHub supplies the inventory data, and Nornir supplies the automation logic.

Installing and Configuring the InfraHub Plugin

The Nornir inventory plugin for InfraHub is called `nornir-infrahub`. This plugin connects to InfraHub's API, retrieves device data, and maps the data into Nornir's internal inventory structure.

You install the plugin using `pip`:

```
pip install nornir-infrahub
```

An InfraHub instance is required and can be deployed on a local server (for example, using Docker), as a virtual machine, or as a hosted/cloud service. An API token must be generated in the InfraHub web interface so Nornir can authenticate to the API. As with other inventory plugins, InfraHub is configured in `config.yaml`, as shown in [Example 12-16](#).

Example 12-16 *config.yaml Using InfraHub*

```
inventory:
  plugin: nornir_infrahub.plugins.inventory.infrahub.InfrahubInventory
  options:
    infrahub_url: "http://localhost:8000"
    infrahub_token: "YOUR_API_TOKEN"
    branch: "main"

runner:
  plugin: threaded
  options:
    num_workers: 10
```

In this configuration:

- **infrahub_url** specifies where the InfraHub API is hosted.
- **infrahub_token** is used for API authentication.
- **branch** allows you to select which version (branch) of the infrastructure data to use.

Once configured, Nornir is initialized in the same way as before:

```
from nornir import InitNornir

nr = InitNornir(config_file="config.yaml")
```

At this point, Nornir queries InfraHub and builds its inventory dynamically.

How InfraHub Data Maps to Nornir Inventory

Like NetBox and the other inventory sources discussed in this chapter, InfraHub stores infrastructure information in its

own data model. The role of the nornir-infrahub plugin is to read that data and convert it into Nornir's standard inventory structure.

Conceptually, the mapping works as follows:

- InfraHub devices become Nornir hosts.
- InfraHub attributes and relationships become host data.
- InfraHub classifications such as site, role, or tags can be used to create Nornir groups.

For example, an InfraHub device with a name, primary IP address, platform, site, role, and tags is converted into a Nornir host with connection-related fields such as `hostname` and `platform` and custom metadata that is accessible through keys like **`host["site"]`**, **`host["role"]`**, and **`host["tags"]`**.

Internally, Nornir still creates the same host objects you've seen throughout this chapter. Once the inventory is loaded, your automation code, filters, and tasks work exactly the same way, regardless of whether the data originated from YAML files, NetBox, Ansible, CSV files or Excel spreadsheets, or InfraHub. InfraHub does not change how Nornir works; it changes where inventory data comes from.

Accessing Inventory Data from InfraHub

Once Nornir has been initialized using the InfraHub inventory plugin, inventory access works exactly the same way as with any other inventory source discussed in this chapter. (You are probably detecting a pattern with all these plugins!)

At runtime, Nornir has already queried InfraHub's API, retrieved device data, applied inheritance rules, and created internal host objects. From this point forward, your

automation code does not interact with InfraHub directly; it interacts with Nornir's inventory model. Here is an example:

```
# Access the host object created by Nornir
host = nr.inventory.hosts["router1"]

print(host.name)          # "router1"
print(host.hostname)      # "192.168.1.1"
print(host.platform)      # "ios"

print(host["site"])       # "san-jose"
print(host["role"])       # "core-router"
print(host["tags"])       # ["production", "core"]
```

In this example:

- **host.name**, **host.hostname**, and **host.platform** are built-in host attributes populated from InfraHub device fields.
- Values such as `site`, `role`, and `tags` are custom metadata derived from InfraHub relationships and are accessed using dictionary-style syntax.

The key takeaway—consistent across all inventory plugins—is that once Nornir is initialized, your code does not need to know (and it does not care) where the inventory data came from. Whether the inventory originated from YAML files, Python dictionaries, NetBox, Ansible, spreadsheets, or InfraHub, Nornir presents the same host objects to your automation code.

When to Use InfraHub Inventory

Using InfraHub with Nornir is most appropriate in environments where infrastructure data is already managed in InfraHub and treated as a shared, authoritative source of truth. It is particularly well suited for teams that value

version control and change history for inventory data and that manage infrastructure using infrastructure-as-code practices. In these environments, inventory data is often consumed by multiple automation tools, CI/CD pipelines, and operational workflows, making InfraHub a natural integration point for Nornir-based automation.

For learning, labs, or smaller environments, SimpleInventory is usually sufficient. InfraHub becomes valuable in larger, more mature environments where infrastructure data is treated like software and tightly integrated into automated workflows.

Summary

In this chapter, you have learned how Nornir manages inventory and why inventory is such a foundational concept in network automation. Rather than focusing on individual tasks or device interactions, this chapter emphasizes how Nornir represents your network—and why that representation enables scalable, flexible automation.

You have learned that Nornir's inventory is built around a consistent three-layer model:

- Hosts, which describe individual devices
- Groups, which describe what devices have in common
- Defaults, which provide global fallback values

Together, these layers form a hierarchical system that allows values to be inherited, overridden, and reused across your network. Once inventory data is loaded, Nornir merges it into internal host objects, which are what your Python code actually works with—regardless of where the inventory data originally came from.

You have seen that while the inventory structure stays the same, Nornir is flexible about where inventory data comes from. Inventory plugins allow Nornir to load device data from a variety of sources, including:

- YAML files using SimpleInventory
- Python dictionaries using DictInventory
- Centralized systems such as NetBox
- Existing Ansible inventories
- CSV and Excel spreadsheets
- Modern infrastructure sources like InfraHub

Each of these approaches serves different environments and organizational needs, but all of them ultimately produce the same internal inventory model inside Nornir. This means your tasks, filters, and automation logic do not need to change when the inventory source changes.

The key takeaway from this chapter is that inventory is the source of truth that drives automation. By separating device data from automation logic, Nornir allows you to write cleaner, more reusable programs that adapt automatically as your network grows and changes.

Part 4: What's Next

Chapter 13. What's Next

Before we talk about new tools, frameworks, or technologies, it's important to pause and take stock of what you've already learned.

If you understand Netmiko, NAPALM, and Nornir, you already understand most of modern network automation. That may not feel obvious—especially when terms like API (application programming interface), YANG model, NETCONF, and RESTCONF start appearing. These technologies are often presented as a sharp break from traditional networking, and it's common for CCNA or CCNP-level engineers to feel that they are suddenly entering an entirely new world. In reality, you are not.

Throughout this book, you have been automating the same tasks network engineers have always performed: connecting to devices, retrieving information, and making configuration changes—and doing so in a consistent, repeatable way. What has changed is how those tasks are expressed, not what the tasks are.

The tools you've learned already embody the core ideas behind modern network automation: abstraction, structure, consistency, and intent. APIs

and data models do not replace these ideas; they build on them. As you'll see in this chapter, many of the concepts that seem unfamiliar at first are simply more formal versions of patterns you already understand. This chapter is not about learning a new tool. It's about recognizing how the skills you've developed position you to move forward with confidence.

Why Network Automation Became Necessary

Network automation did not emerge because network engineers suddenly wanted to write code. It emerged because networks changed.

As Ethan Banks succinctly states, “A network is a complex interaction of multiple dependencies.” Modern networks are no longer defined simply by the number of devices they contain but by the interdependencies between services, protocols, policies, and users that all must function correctly—simultaneously.

Enterprise networks, service provider networks, and data centers now support far more than basic connectivity. They are expected to deliver:

- High availability and resiliency
- Predictable performance and quality of service (QoS)
- Real-time applications such as voice and video
- Strong security and segmentation
- Rapid scalability and frequent change

At the same time, these networks are expected to behave like utilities—always on, always available, and largely

invisible to the users who depend on them, much like electricity or water.

What Changed

Historically, networks were smaller, changes were infrequent, and failures—while undesirable—were often localized and manually recoverable. Today, even modest environments can include hundreds or thousands of interconnected devices, layered security controls, dynamic routing, overlays, tunnels, virtualization, and cloud integration.

What changed is not just scale but expectation. Networks must now:

- Be deployed faster
- Recover from failures automatically or nearly instantly
- Be secured continuously, not periodically
- Change frequently without user impact
- Be observable and diagnosable in real time

Manual, device-by-device configuration and troubleshooting simply do not scale to meet these demands.

Automation as an Operational Requirement

Automation provides a way to manage complexity without increasing risk. By making network changes repeatable, consistent, and verifiable, automation reduces human error—the leading cause of network outages—while enabling faster response to incidents and vulnerabilities.

Automation allows network teams to:

- Implement changes across many devices consistently
- Enforce policy and configuration standards
- Detect and resolve issues more quickly
- Validate network state continuously
- Make changes with minimal or no user disruption

In other words, automation shifts the network from a fragile system dependent on individual actions to a resilient system governed by intent and verification.

SDN and IBN in Context

Two terms that frequently appear in discussions about modern networks are software-defined networking (SDN) and intent-based networking (IBN). Both SDN and IBN emerged as responses to growing network complexity and the need to manage networks at scale, but their meanings and implementations have evolved over time.

SDN originally promised a clean separation between the control plane and the data plane, with centralized control and simplified forwarding devices. In practice, SDN has shifted from that original, narrow definition into a broader architectural approach that emphasizes programmability, APIs, and centralized policy control, often implemented incrementally rather than as a wholesale redesign of the network. While the early vision of SDN was ambitious, its lasting impact has been the normalization of automation, abstraction, and software-driven control in networking.

IBN builds on this evolution by focusing on what the network should do rather than how individual devices should be configured. In an intent-based model, high-level requirements—such as security policies, performance guarantees, or availability goals—are translated into

configuration, validated continuously, and adjusted as conditions change. Achieving this requires reliable automation, structured data, and ongoing verification.

Both SDN and IBN reinforce the same conclusion: Modern networks can no longer be managed device by device, one command at a time. Whether or not an organization formally adopts SDN or IBN, the underlying requirements—automation, consistency, validation, and scale—remain the same. The tools and approaches explored in this book address those requirements directly, regardless of the architectural labels applied to them.

A Natural Evolution, Not a Replacement

It is important to understand that automation does not replace networking fundamentals. Routing, switching, security, and design principles remain essential. What automation changes is how those fundamentals are applied—at speed, at scale, and with greater reliability. The tools explored in this book—Netmiko, NAPALM, and Nornir—exist because they address this reality directly. They help network engineers manage growing complexity while preserving control, visibility, and trust in the network.

Automation, then, is not about doing something new. It is about doing what network engineers have always done—more reliably, more consistently, and at the scale that modern networks demand.

Netmiko, NAPALM, and Nornir in Context

In [Chapter 1, “Introducing Netmiko, NAPALM, and Nornir,”](#) we introduced the same figure shown here as [Figure 13-1](#). This figure provides a useful way to step back and see how the tools covered in this book fit into the broader network automation landscape. Rather than view Netmiko, NAPALM, and Nornir as isolated technologies, the diagram highlights the specific problems each one solves and how they build on one another.

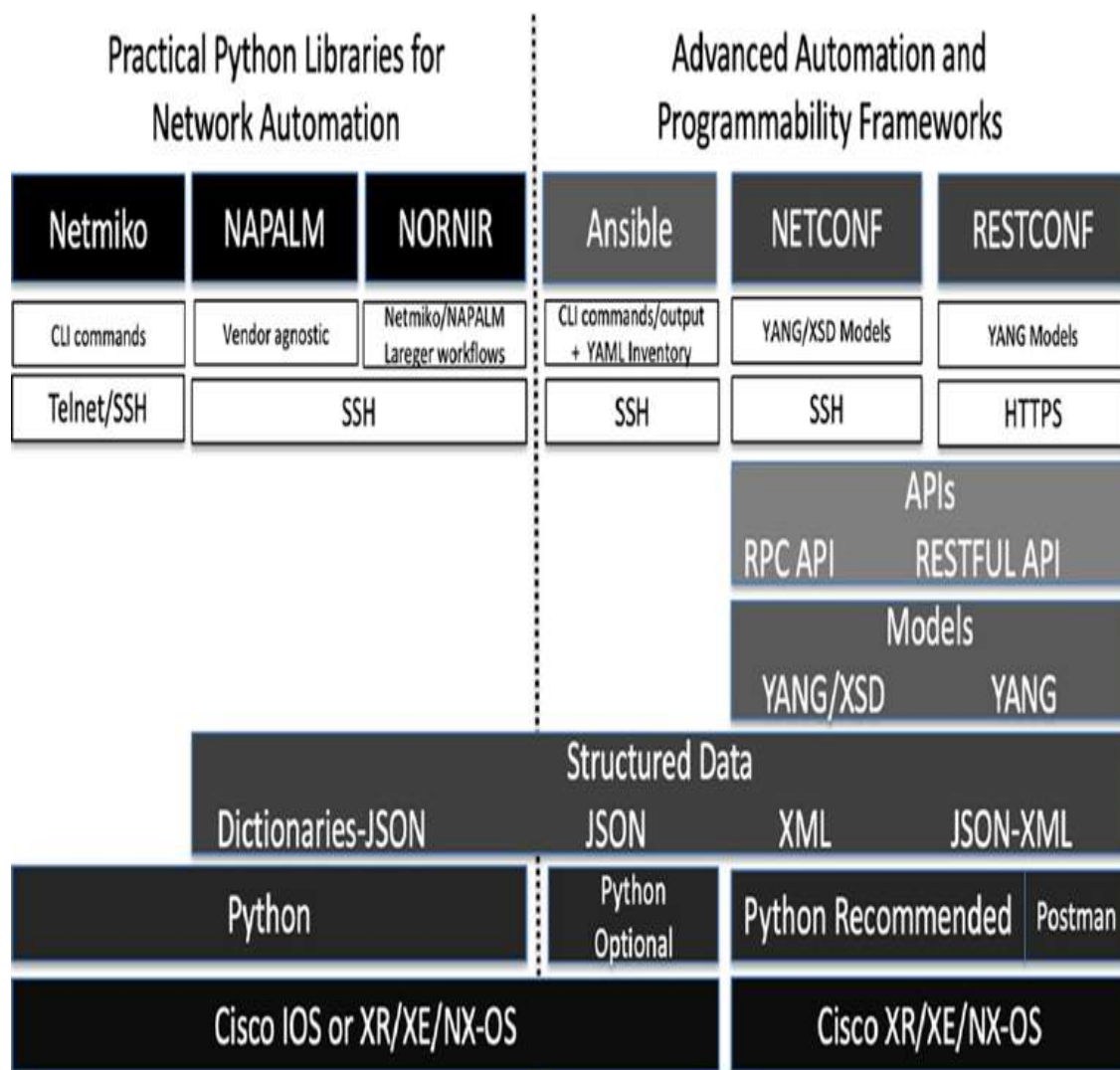


Figure 13-1 Network automation tools: From CLI to API-driven frameworks

Netmiko sits closest to the traditional way network engineers work. It automates direct CLI access over SSH, allowing Python programs to send commands and receive output just as a human would. Netmiko excels at scaling familiar workflows—running **show** commands, applying configuration changes, and gathering information across many devices—without requiring changes to the devices themselves. If you understand the command-line interface, you already understand Netmiko.

NAPALM moves automation one step further by introducing structure and consistency. Instead of working primarily with raw CLI output, NAPALM provides a vendor-neutral abstraction layer and returns data in structured formats such as dictionaries and JSON. This makes it easier to compare device state, validate configurations, and reason about the network programmatically. NAPALM shifts the focus from *how* commands are issued to *what* the network state should be.

Nornir addresses a different problem altogether: orchestration. Rather than replace Netmiko or NAPALM, Nornir organizes how automation is executed across the network. It cleanly separates inventory, automation logic, and execution, allowing you to scale workflows, reuse data, and apply the same operations consistently across many devices. Nornir does not change what tools you use; it changes how you manage them.

As shown in [Figure 13-1](#), these tools form a natural progression. Netmiko automates individual interactions, NAPALM introduces structure and abstraction, and Nornir ties everything together into repeatable, scalable workflows. Together, these tools move you from automating commands to automating intent.

Each of the tools covered in this book was developed for a purpose, and for beginners, their greatest strength is how they lower the barrier to entry into network automation:

- Netmiko is often the easiest place to start because it closely mirrors how network engineers already interact with network operating systems. By working directly with the CLI over SSH, Netmiko allows new Python users to focus on basic programming concepts—such as variables, loops, and conditionals—without needing to learn new data models or APIs up front. This makes it especially effective for building confidence while automating familiar operational tasks.
- NAPALM builds on that foundation by introducing structure and consistency in a controlled way. Instead of parsing raw command output, beginners begin working with structured data returned as dictionaries and JSON, which helps them develop a practical understanding of how network state can be represented programmatically. At the same time, NAPALM's vendor-neutral interface reduces the need to understand every platform-specific detail of a network operating system, allowing learners to focus on what information they need rather than how each device provides it.
- Nornir supports the next step in the learning process by showing how automation scales beyond individual devices or one-off programs. Nornir helps beginners understand how real-world automation systems are organized. It cleanly separates inventory, automation logic, and execution, allowing you to scale workflows, reuse data, and apply the same operations consistently across many devices. Nornir does not change what tools you use; it changes how you manage them.

Used together, these tools allow beginners to grow naturally: starting with familiar CLI interactions, progressing

to structured data and abstraction, and ultimately learning how automation workflows are organized at scale. Rather than replace existing networking knowledge, they build on it, providing a clear and approachable path forward for anyone beginning their journey into network automation.

While these tools are well suited for beginners, they are not limited to introductory use. Netmiko, NAPALM, and Nornir are widely used by experienced network engineers in production environments to automate operational tasks, enforce consistency, and manage large, complex networks reliably.

Ansible, NETCONF, and RESTCONF: Expanding the Automation Toolkit

As shown in [Figure 13-1](#), Netmiko, NAPALM, and Nornir are part of a broader automation ecosystem that also includes configuration frameworks and API-based interfaces. [Chapter 12, “Inventory Management with Nornir,”](#) discusses how Ansible can be used as the source of truth for device information. This section dives a bit deeper into Ansible, as well as NETCONF and RESTCONF. It focuses on what they are, how they differ from the tools you’ve already learned, how your existing knowledge applies, and how to access each of these tools.

Ansible

Ansible is a task-based automation framework that is widely used across IT, particularly in server administration, system configuration, and application deployment. Its popularity in those areas is due to its simple, declarative approach to automation, which allows users to describe what actions should be performed rather than write procedural code to

perform them. Automation tasks are defined using YAML playbooks and executed against groups of devices or systems. For network automation, Ansible typically connects to devices either through SSH (CLI-based modules) or through API-based modules such as NETCONF or RESTCONF.

Ansible is often described as an example of infrastructure as code (IaC), which makes it possible to manage and provision infrastructure using machine-readable definition files rather than through manual configuration or interactive command-line sessions. In the context of Ansible, these definitions are written in YAML (Yet Another Markup Language), which is a simple, human-readable format used to describe automation tasks and desired system state.

For network engineers, this concept is not as foreign as it may sound. Much like a well-documented change plan, an Ansible playbook explicitly defines what actions should be taken and can be reused, reviewed, and applied consistently. Instead of relying on ad hoc CLI sessions, automation logic is captured in files that can be versioned, shared, and executed repeatedly.

[Figure 13-2](#) illustrates the basic Ansible execution model for network automation. Ansible runs from a central control node and connects to network devices using SSH, much like the tools covered earlier in this book. Devices are defined in an inventory, often grouped by role, such as routers or switches, allowing tasks to be applied consistently across similar systems.

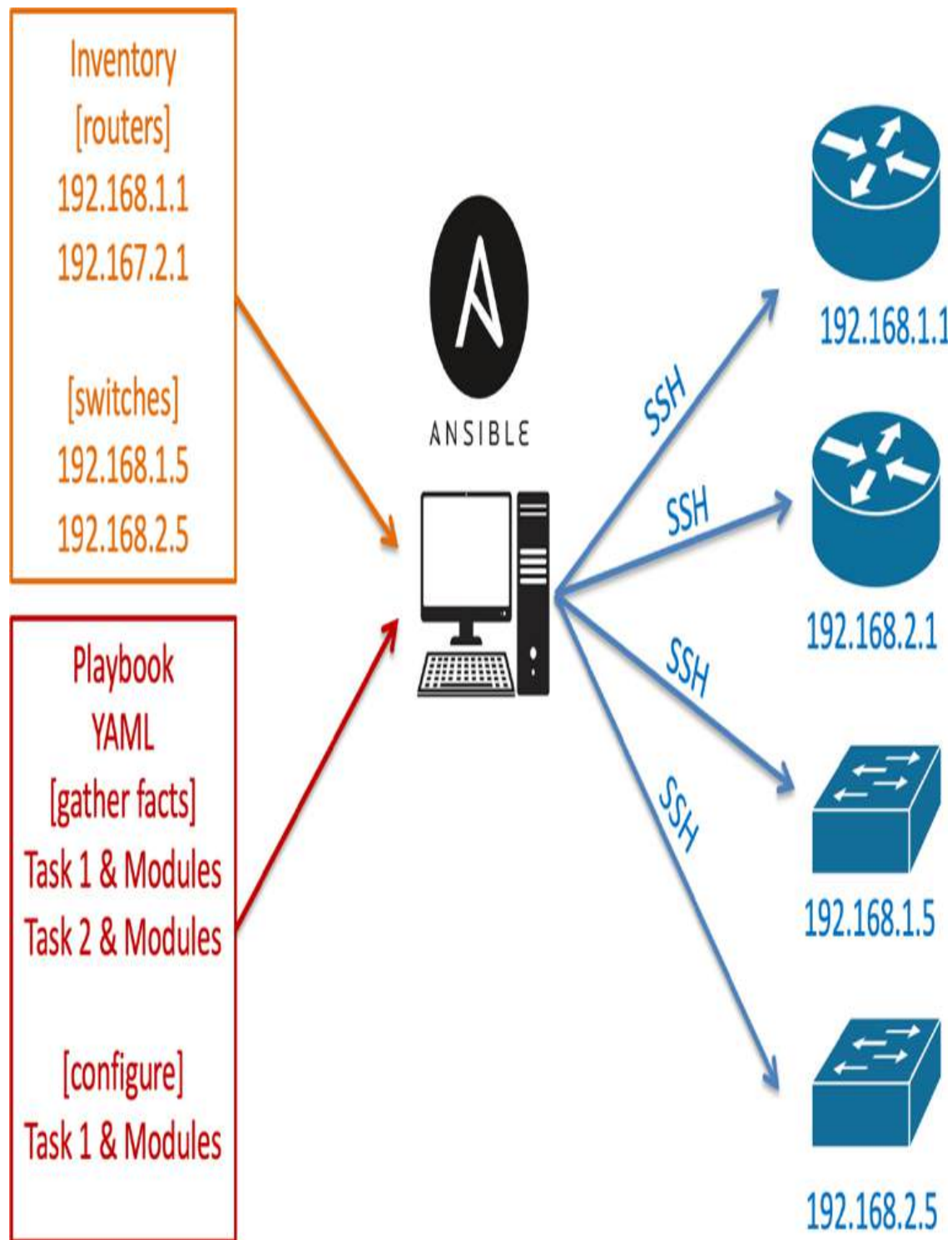


Figure 13-2 *Ansible execution model*

Automation logic is defined in YAML playbooks, which contain sequences of tasks. Each task invokes a module to

perform a specific action, such as gathering information or applying configuration. Rather than writing procedural code, the user describes the desired actions, and Ansible handles execution across the inventory.

For readers familiar with Nornir, the concepts shown here should be recognizable. Ansible's inventory serves a similar purpose, and its task-based execution model parallels how automation workflows are organized—just expressed declaratively in YAML instead of Python. The key difference is that Ansible emphasizes what should happen and abstracts away much of the underlying execution logic.

How Ansible Differs from Netmiko, NAPALM, and Nornir

Unlike Netmiko and NAPALM, Ansible is not a Python library that you import into a program. Instead, it is a framework that executes predefined modules. Whereas Nornir is Python driven and code centric, Ansible emphasizes declarative workflows defined in YAML. Whereas Netmiko and NAPALM directly control how a connection is made, Ansible abstracts those details and selects the access method (SSH or API) based on the module being used.

How Your Existing Knowledge Helps

If you understand Nornir inventory files, Ansible's inventory model will feel familiar. Experience with NAPALM's structured data makes it easier to understand Ansible facts and module output, which are also returned as dictionaries and JSON. Even Netmiko experience helps because many Ansible network modules ultimately perform the same operational tasks—retrieving information and applying configuration—just through a different interface.

Why Someone Would Choose Ansible

Ansible is often chosen when teams want standardized, readable workflows that can be shared across engineering and operations teams. For organizations already using Ansible to manage servers, applications, or cloud infrastructure, extending Ansible to network automation allows them to use the same tools, workflows, and operational model across their environment. Its ability to use SSH for CLI-based automation or APIs for model-driven automation makes it flexible across different network platforms, while it also fits naturally into existing automation practices and organizational skillsets.

RESTCONF

RESTCONF is a RESTful API for connecting to network devices that provides access to YANG-defined configuration and operational data over HTTP or HTTPS. (YANG is discussed later in this chapter.) These same YANG models are also used by other model-driven management protocols, such as NETCONF. RESTCONF commonly uses JSON as the data format and relies on standard HTTP methods such as GET, POST, PUT, and DELETE to retrieve or modify network state.

[Figure 13-3](#) illustrates the basic interaction model used by RESTCONF. At a high level, RESTCONF works in much the same way as a web application: A client sends an HTTP request to a server, and the server returns a response. The key difference is that, instead of returning HTML meant for human users, a RESTCONF-enabled network device returns structured data—most commonly JSON, and sometimes XML—intended for automation tools.

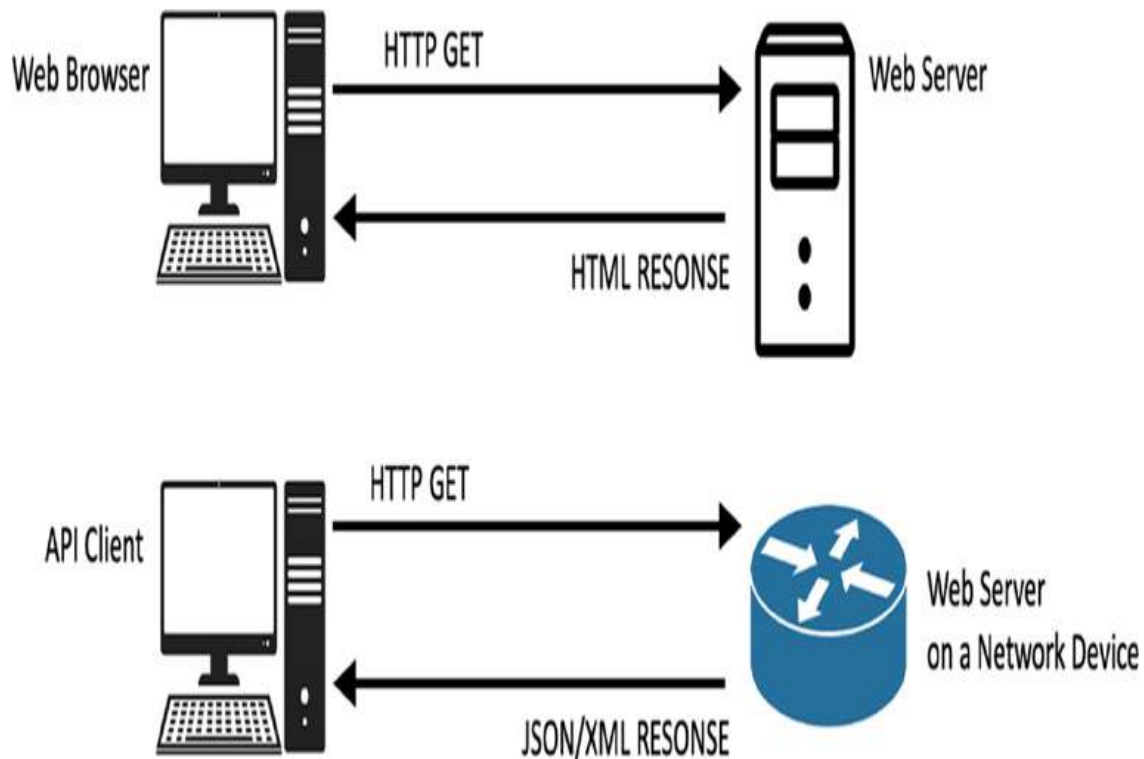


Figure 13-3 *RESTCONF operation*

RESTCONF exposes parts of a network device's configuration and operational state as RESTful resources. Clients interact with these resources using standard HTTP methods such as GET, POST, PUT, and DELETE over HTTP or HTTPS. The data exchanged is defined by YANG models, which describe the structure and meaning of the information being retrieved or modified.

By relying on widely used web technologies, RESTCONF makes network devices accessible to a broad range of tools and programming languages. For readers familiar with working with JSON data in Python or retrieving structured output through NAPALM, RESTCONF represents a natural extension of those ideas: Rather than parse CLI output, automation systems request specific data and receive predictable, machine-readable responses.

How RESTCONF Differs from Netmiko, NAPALM, and Nornir

RESTCONF differs fundamentally from CLI-based tools by using web-based APIs rather than SSH-based command-line access. Unlike Netmiko or NAPALM, RESTCONF does not send commands; it manipulates structured resources exposed by the device. While Nornir can coordinate RESTCONF calls, RESTCONF itself defines the API interface offered by the network operating system.

How Your Existing Knowledge Helps

Experience with structured data through NAPALM makes RESTCONF far less intimidating. JSON payloads, key/value pairs, and predictable data structures closely resemble what you've already worked with in Python. Understanding how Nornir organizes inventory and workflows also makes it easier to integrate RESTCONF into larger automation systems.

Why Someone Would Choose RESTCONF

RESTCONF is often used when network automation needs to integrate with external systems such as web applications, controllers, or monitoring platforms. Its reliance on standard web APIs (HTTP/HTTPS) and widely supported data formats makes it accessible across many programming languages and tools.

NETCONF

NETCONF is a network management protocol designed specifically for configuring and managing network devices using structured data, but it uses a different interaction model than RESTCONF. It operates using remote procedure

calls (RPCs) that are exchanged over SSH, allowing a client to request specific operations from a device. Instead of sending CLI commands, NETCONF uses structured, XML-encoded messages defined by YANG data models. These RPC-based exchanges allow clients to retrieve, validate, and modify configuration data directly, with support for features such as locking, validation, and transactional changes.

Figure 13-4 illustrates the basic interaction model that NETCONF uses. Rather than exchanging CLI commands, a client communicates with a network device using RPCs over SSH. Each RPC represents a specific operation, such as retrieving configuration data or modifying device state. A NETCONF RPC is an XML message—wrapped in the `<rpc>` element—that the client sends to a device to request an operation such as retrieving configuration data or modifying settings. The device processes this request and returns a corresponding `<rpc-reply>` message, which is another XML response that contains either the requested data or the result of the operation. In NETCONF, every `<rpc>` sent by the client generates a matching `<rpc-reply>` from the device.

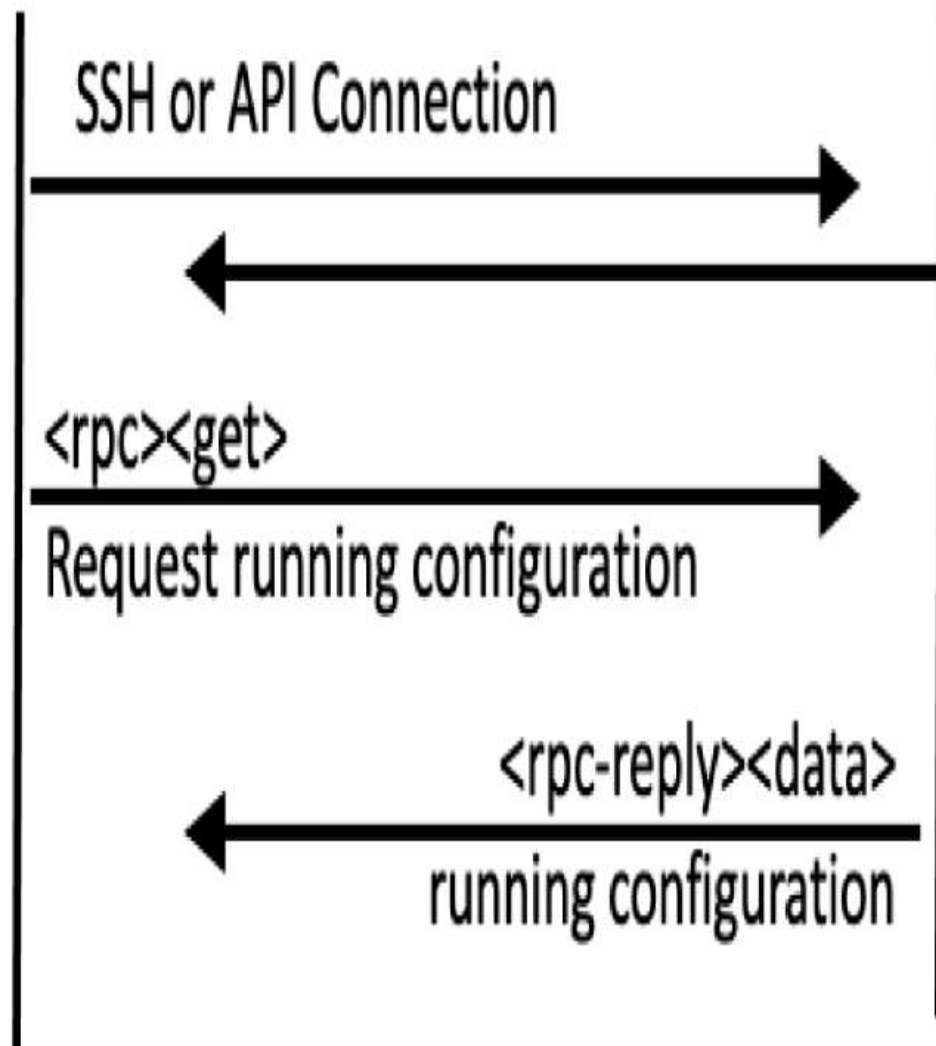


Figure 13-4 *NETCONF operation*

In the example shown in [Figure 13-4](#), the client sends an `<rpc><get>` request to retrieve the device's running configuration. The device responds with an `<rpc-reply>` that contains the requested data, encoded in a structured format defined by YANG models. This request/response exchange is explicit, predictable, and machine readable.

By operating on structured data instead of command output, NETCONF allows automation tools to retrieve, validate, and modify configuration state directly. Features such as configuration locking, validation, and transactional commits are built into the protocol to reduce ambiguity and minimize unintended changes. While NETCONF still uses SSH as its transport, the interaction model is fundamentally different from CLI-based automation: The client asks for specific data or operations, and the device responds accordingly.

How NETCONF Differs from Netmiko, NAPALM, and Nornir

While Netmiko and NAPALM use SSH to interact with the CLI, NETCONF uses SSH as a transport for structured API messages. Nornir can orchestrate NETCONF-based tasks, but NETCONF itself defines the protocol used to access the device. Rather than automating command execution, NETCONF automates configuration state through model-driven interactions.

How Your Existing Knowledge Helps

NAPALM's use of structured data provides an important conceptual bridge to NETCONF. If you are comfortable working with dictionaries and structured data formats such as JSON, the idea of YANG-defined configuration data—

regardless of its encoding—is a natural progression. Your understanding of network operating systems—interfaces, routing protocols, VLANs, and policies—still applies; NETCONF simply represents these concepts in a formal data model rather than as CLI commands.

Why Someone Would Choose NETCONF

NETCONF is often chosen when accuracy, validation, and transactional configuration changes are important. Its use of SSH-based, model-driven APIs makes it well suited for environments that require precise control over configuration and strong guarantees about network state.

Comparisons

Ansible, NETCONF, and RESTCONF expand network automation beyond the CLI, but they do not replace the tools you’ve learned about in this book. Instead, they provide additional interfaces—SSH-based APIs and web-based APIs—that build on the same foundational ideas: structure, consistency, and intent.

[Table 13-1](#) provides a high-level comparison of the tools and interfaces discussed in this chapter. Rather than focus on syntax or implementation details, it highlights how the approaches differ in terms of access method, data formats, system requirements, and typical use cases, making it easier to see where each approach fits in a modern network automation workflow.

Table 13-1 *Comparison of Automation Tools*

Tool/Interface	Primary Purpose	Access Method	Output/Data Format	System Requirements	Typical Use Cases
Netmiko	Automate direct CLI interaction	SSH (CLI)	Raw text output (strings)	SSH enabled on device	Running show commands, pushing configuration, automating existing operational workflows
NAPALM	Provide structured, vendor-neutral automation	SSH (CLI-based drivers)	Python dictionaries, JSON	SSH enabled on device	Retrieving structured state, validating configuration, comparing network state
Nornir	Orchestrate and scale automation workflows	SSH and/or APIs (via plugins)	Text, dictionaries, JSON (depending on the task)	Python environment; SSH enabled on device (with access handled by underlying tools)	Managing inventory, coordinating tasks, scaling automation across many devices
Ansible	Provide a declarative automation framework	SSH (CLI modules) or APIs (NETCONF/RESTCONF modules)	Dictionaries, JSON, YAML	Ansible control node; SSH or API access on devices	Using standardized workflows, managing configurations, providing cross-team automation
RESTCONF	Provide RESTful API	HTTP/HTTPS (API)	JSON (commonly),	RESTCONF enabled on	Integrating with web apps,

Because you already understand Python, structured data, and network operating systems, using these technologies does not require a leap into the unknown. Instead, using them is a natural continuation of the journey you've already started.

Understanding YANG Models

A YANG model is a structured blueprint that defines how data on a network device is organized and accessed—what information exists, how it is formatted, and how different pieces of data relate to one another. Rather than rely on free-form CLI output, YANG models describe device configuration and operational state in a precise, machine-readable way.

YANG models are central to model-driven automation. They allow automation tools to retrieve and configure data consistently across different devices, vendors, and platforms, making automation more predictable and less dependent on platform-specific command syntax.

YANG models ensure that data is presented in a consistent format across vendors, platforms, and device models.

Which Tools Use YANG Models and Which Do Not

It is important to understand where YANG fits in the automation ecosystem discussed in this book:

- NETCONF and RESTCONF both rely directly on YANG models.
- NETCONF uses YANG-defined data structures exchanged via RPCs (typically encoded in XML).

- RESTCONF exposes YANG-defined data as RESTful resources, most commonly using JSON.
- Ansible often uses YANG indirectly, through NETCONF- or RESTCONF-based modules. In these cases, Ansible playbooks interact with YANG-modeled data without requiring the user to work directly with the models.
- Netmiko, NAPALM, and Nornir do not require YANG models.
 - Netmiko and NAPALM primarily interact with devices through the CLI.
 - Nornir orchestrates tasks and workflows but depends on the underlying tool or protocol being used.

This distinction is important: Learning Netmiko, NAPALM, and Nornir does not require knowledge of YANG, but it prepares you conceptually for working with structured, model-driven data when you encounter NETCONF, RESTCONF, or API-based automation.

Types of YANG Models

There are two common categories of YANG models you will encounter:

- **IETF models:** These models are standards based and defined by the Internet Engineering Task Force (IETF). They typically begin with the prefix `ietf-` (for example, `ietf-interfaces`, `ietf-ip`, `ietf-routing`). IETF models describe vendor-neutral concepts like interfaces, IP addressing, and routing. They are designed to work consistently across different vendors' devices.
- **Vendor-specific (native) models:** These models extend standard functionality with platform-specific features. For Cisco IOS XE, these models typically begin

with Cisco-IOS-XE- (for example, Cisco-IOS-XE-native, Cisco-IOS-XE-vlan-oper). Native models expose capabilities that are unique to a particular platform, such as VLAN behavior, QoS features, EEM, or device-specific operational data.

Note

You can explore both standard and vendor-specific YANG models, including their hierarchical tree structures, at yangcatalog.org.

Why YANG Models Matter

Using YANG models provides several important advantages for network automation:

- They produce consistent data structures and field names across devices and vendors.
- They make automation more reliable by allowing tools to expect predictable output formats.
- They enable configuration and operational data to be parsed, validated, and analyzed programmatically.
- They support interoperability between automation tools, controllers, and APIs such as NETCONF and RESTCONF.

In short, YANG models replace ambiguous, human-oriented command output with clearly defined data structures that automation systems can understand and trust.

A Helpful Way to Think About YANG

Here is a useful mental model for thinking about YANG:

- The CLI shows you how to configure a device.

- YANG models describe what the device is capable of and how its data is structured.

The shift from commands to data models is what enables APIs, controllers, and intent-based networking. And because you already understand structured data and automation workflows, YANG is an extension of concepts you already know rather than a completely new way of thinking.

Understanding APIs

An API (application programming interface) is a defined way for one system to request data or services from another system and receive a structured response. Rather than interact directly with a device or application through a user interface or command line, an API provides a formal, predictable interface that software can use.

[Figure 13-5](#) illustrates this idea using a familiar analogy. Just as a customer places an order through a waiter rather than entering the kitchen, an application sends a request to an API instead of interacting directly with the underlying system. The API receives the request, passes it to the appropriate service, and returns a response in a structured format.

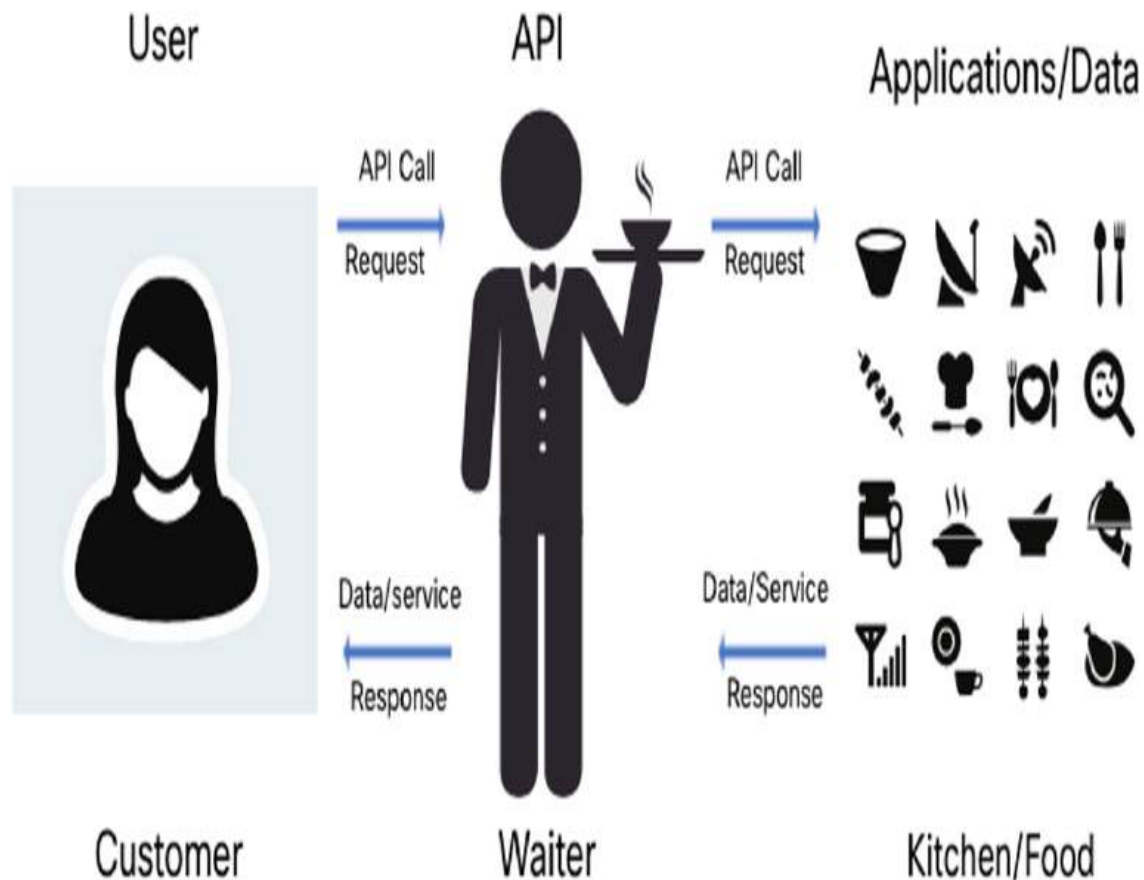


Figure 13-5 *A waiter as an API analogy*

APIs in Everyday Life

Most users interact with APIs daily, even if they are not aware of it. When you use Google Maps to get directions, book a flight or restaurant reservation, or check the weather on your phone, your device is making API requests behind the scenes. In each case, your application sends a request, and the service responds with structured data that the application displays in a human-friendly way. Network automation uses the same model, except that the client is a Python program or automation framework, and the server is a network device.

Why APIs Are Useful for Automation

APIs offer several advantages over manual or CLI-based interaction:

- They provide structured, machine-readable data instead of free-form text.
- They are predictable and consistent, making automation more reliable.
- They reduce the need to parse CLI output.
- They enable integration with external systems, such as controllers, monitoring platforms, and web applications.
- They allow automation to focus on what data is needed rather than how to retrieve it.

In network automation, APIs are a key enabler for model-driven and intent-based approaches.

How APIs Are Used in Practice

In practice, an API interaction follows a simple request/response model:

Step 1. A client sends a request to an API endpoint.

Step 2. The server processes the request.

Step 3. The server returns a response that contains structured data, most commonly in JSON or XML format.

Python is commonly used as the API client, using libraries that handle HTTP requests and responses. From a programmer's perspective, working with an API often means sending a request and receiving a dictionary or data structure that can be inspected, validated, or acted upon.

Which Tools in This Chapter Use APIs

Several of the tools and technologies discussed in this chapter rely on APIs:

- NETCONF uses an API-based protocol over SSH, exchanging structured data via RPCs.
- RESTCONF uses a RESTful API over HTTP or HTTPS and typically returns JSON.
- Ansible can use APIs indirectly through NETCONF- or RESTCONF-based modules.
- Nornir can orchestrate API-based tasks through plugins.
- Netmiko and NAPALM primarily use SSH and the CLI and do not require APIs.

Keep in mind that APIs extend automation capabilities, but they do not replace CLI-based tools. They are another interface option when a platform supports them.

What Is Required to Use an API

Using an API requires support on both the server and the client:

- **On the server (network device):** The API must be enabled (for example, via RESTCONF or NETCONF), the device must support structured data models (often YANG), and authentication and authorization must be configured.
- **On the client (automation system):** An API-capable tool or library (such as a Python program or automation frameworks) is used, credentials and network connectivity are required to reach the device, and the

client must understand the structure of the data being requested or modified.

Once these pieces are in place, API-based automation becomes highly reliable and scalable.

A Helpful Way to Think About APIs

A simple way to think about APIs in network automation is this: An API is a contract that defines how software can ask a device for information or request a change—and what kind of response it will receive.

This section on APIs is intentionally brief, given the scope of the topic. It is not meant to explain AI in depth or provide a comprehensive treatment of its use in networking but rather to offer a glimpse into what is already happening and what lies ahead.

Artificial Intelligence and Network Automation

It is impossible to discuss “what’s next” in networking without mentioning artificial intelligence. AI is now part of nearly every conversation in technology, and it is evolving at a pace that makes detailed explanations quickly outdated. As Omar Santos aptly notes in *The AI Revolution in Networking, Cybersecurity, and Emerging Technologies*, one of the most striking realities of AI is captured in the title of an early section of the book: “Significant Milestones in AI Development (This Book Is Already Obsolete).” That observation serves as a useful premise for this discussion.

Rather than focus on how AI works internally, this section focuses on how AI fits with the skills you have already developed and how it is beginning to change the way

networks are learned, managed, and secured. This section is extremely brief considering the topic. It is not meant to explain AI's use in networking but to give you a glimpse of what is and will be happening.

AI as a Learning and Exploration Tool

As we stated at the beginning of this book, from a pedagogical standpoint, AI can play a valuable role alongside this book—especially for learners who do not have direct access to networking hardware or full lab environments. In many cases, AI can function as a conceptual emulator, allowing students to interact with *representations* of Cisco IOS, Python programs, and automation tools such as Netmiko, NAPALM, and Nornir.

While AI is not a replacement for real devices, it can simulate configuration workflows, execute sample Python logic, and produce representative output, making it possible to explore automation concepts without requiring physical equipment or virtual lab infrastructure. This can be particularly helpful when someone is learning how tools behave, what output looks like, and how different components interact. It can also lower the barrier to learning these tools and technologies for those without access to physical equipment or virtual lab infrastructure.

AI can also assist by:

- Explaining Cisco IOS configuration concepts and command behavior
- Walking through Netmiko, NAPALM, or Nornir programs step by step
- Simulating program execution and showing sample output

- Helping debug Python logic or automation workflows
- Generating sample configurations or structured data outputs for study

Used thoughtfully, AI becomes a learning companion. It allows students to experiment, ask “what if” questions, and reason through automation workflows—even when physical devices or lab time is not available.

How AI Is Changing Network Operations and Security

In operational environments, AI is already influencing how networks are managed and secured. Rather than replace network engineers, AI systems are increasingly used to:

- Analyze large volumes of telemetry and logs
- Detect anomalies and potential security threats
- Correlate events across devices and layers
- Assist with troubleshooting and root-cause analysis

As networks grow in size and complexity, the ability to quickly interpret data becomes just as important as the ability to configure devices. AI helps shift the focus from manual data inspection to decision support, allowing engineers to spend more time on design, validation, and intent.

Using AI with Netmiko, NAPALM, and Nornir

The tools you have learned about in this book are particularly well suited to working alongside AI:

- Netmiko provides direct access to device output, which AI systems can help interpret, summarize, or validate.
- NAPALM's structured data is especially powerful when combined with AI, as consistent data formats make it easier to analyze state, detect changes, or identify anomalies.
- Nornir enables scalable workflows where AI-assisted logic can help decide what actions to take across large inventories.

In practical terms, AI can help generate automation logic, explain results, suggest next steps, or identify patterns—while Netmiko, NAPALM, and Nornir perform the actual interaction with the network. This separation of responsibilities is important: AI reasons about the network, and your automation tools act on it.

A Final Thought on AI

Throughout this book, you have learned how to automate networks using Python, structured data, and well-designed workflows. Those skills are not made obsolete by AI; they are amplified by it.

AI does not replace networking fundamentals, automation frameworks, or operational discipline. Instead, it builds on them. Because you already understand how networks work, how automation is structured, and how data flows between systems, you are well positioned to take advantage of AI as it continues to evolve.

What comes next will change. The foundation you've built will not.

Closing Thoughts

If you have made it this far, *thank you!*

Learning network automation—especially as a network engineer—is not always easy. It requires stepping outside familiar workflows, thinking a bit differently about problems you already know how to solve, and being willing to experiment. That effort matters, and it shows.

This book was written with a simple goal: to make network automation approachable, practical, and grounded in real networking work. Rather than ask you to abandon the skills you already have, we set out to show how Python, Netmiko, NAPALM, and Nornir build directly on them. If this book has helped demystify automation, lowered the barrier to getting started, or given you confidence to keep exploring, then it has done its job.

Network automation is not a destination. It is an ongoing journey—one that evolves as networks, tools, and expectations change. You do not need to know everything, and you do not need to adopt every new technology. What matters is understanding the foundations, knowing how to learn, and being able to adapt thoughtfully as your environment grows.

Whether you continue by exploring APIs, Ansible, model-driven automation, or AI-assisted workflows, you are now equipped to do so with context and confidence. You understand how automation works, why it exists, and how it fits into real networks.

Most importantly, remember this: Automation is not about replacing network engineers; it is about enabling them.

Thank you for taking this journey with us. We hope this book has been a helpful guide, and we encourage you to keep experimenting, keep learning, and keep building. As the

network—and your role in shaping it—continues to evolve, so will you.

Appendixes

Appendix A. Python Virtual Environments

We briefly mentioned Python virtual environments in [Part 1: Netmiko](#). We'll discuss them in more detail in this appendix.

A *virtual environment* in Python is an isolated workspace where you can install packages separately from the systemwide Python installation. This helps in managing dependencies and avoiding conflicts between different projects. For example, let's say you are working on a project that requires to use version 4.2 of Netmiko and another project that requires version 3.5. Without virtual environments, you'd be limited to using the system installation of Python and could only work on one project because you cannot have two versions of the same library installed at the same time. By using virtual environments, you can create a new project-specific folder and make a copy (or link) to the Python interpreter available inside it. When you create a virtual environment, several directories are created, including /bin, /lib, and /include. The Python interpreter and pip are available inside this new environment.

When using Python virtual environments, each project that you are working on gets its own isolated folder structure where you can **pip install** whatever versions of the Python libraries you need for your project. This means you can use different versions of the same Python packages and libraries

across different projects and keep the global Python installation clean.

Creating a virtual environment is as easy as running the following command:

```
MyPrompt% python3 -m venv nornir_venv
```

This command creates a new folder named `nornir_venv`. shown in [Figure A-1](#) shows the structure of this folder on Linux and macOS.

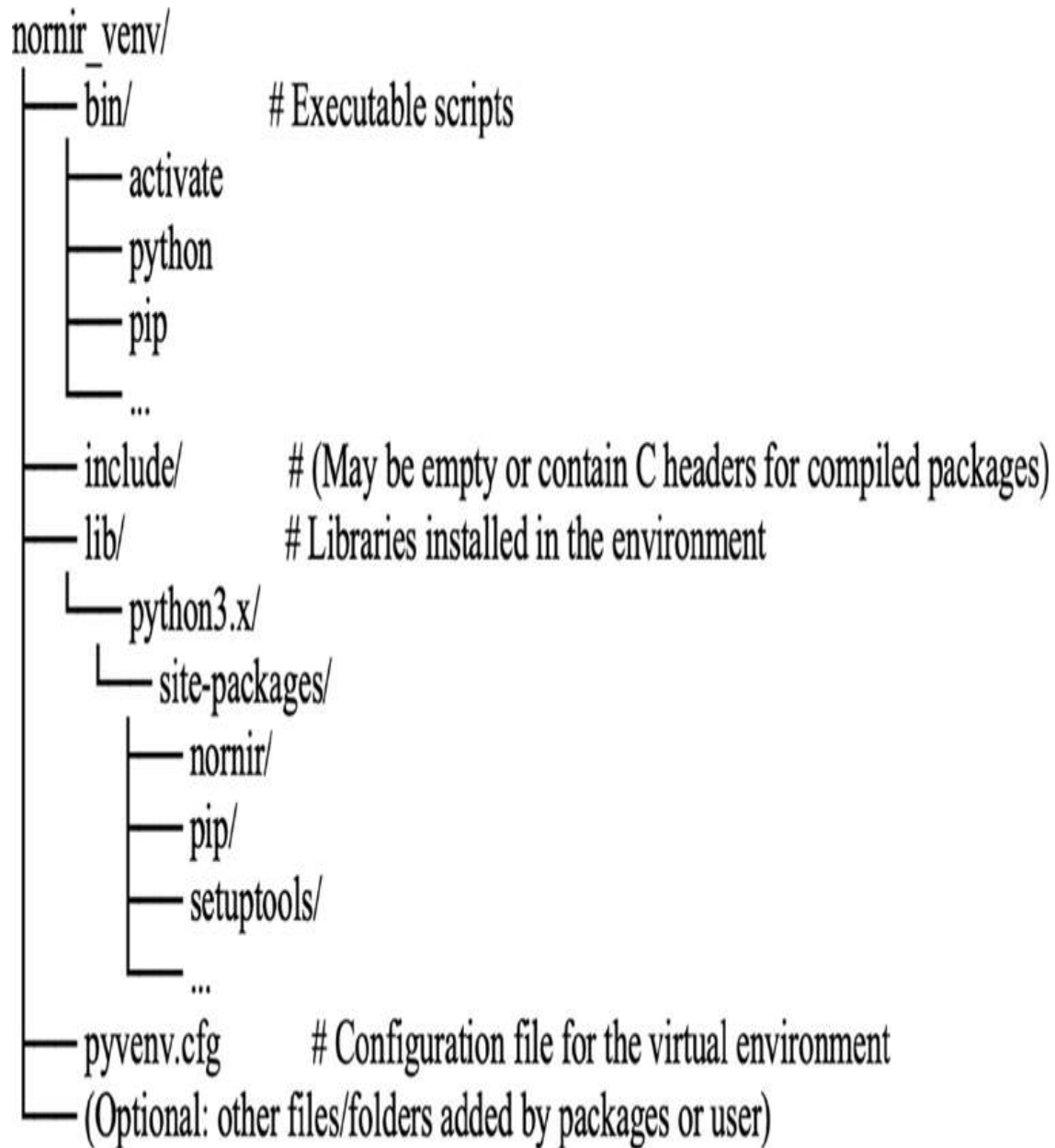


Figure A-1 *Viewing a virtual environment on Linux and macOS*

Figure A-2 shows the structure of the nornir_venv folder in Microsoft Windows.

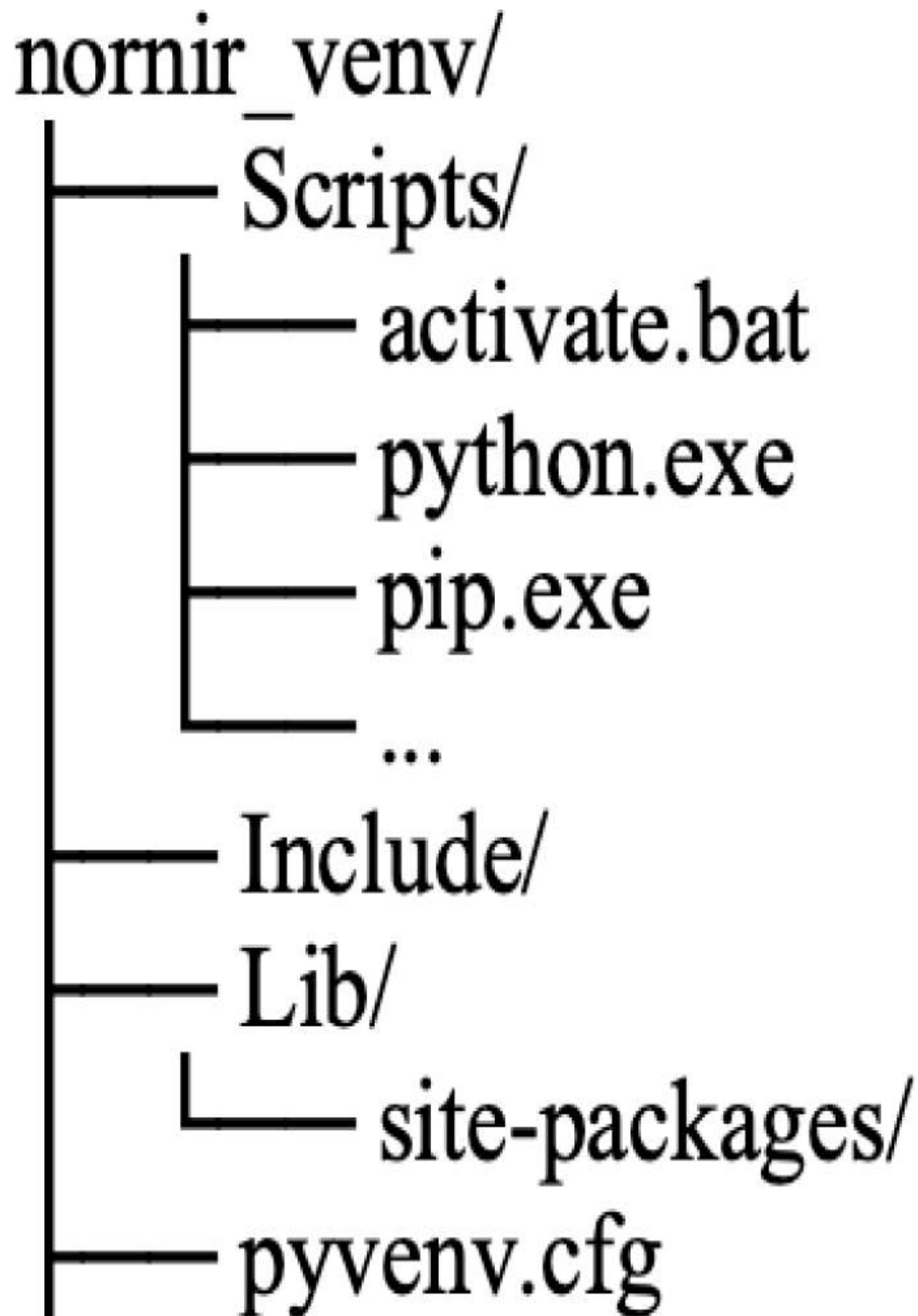


Figure A-2 *Viewing a virtual environment in Windows*

Once you have the virtual environment and the new folder structure created, you must activate the virtual environment before you can start using it. You activate the `nornir_venv` virtual environment on Linux and macOS by issuing the following command in a terminal window:

```
MyPrompt% source nornir_venv/bin/activate
```

On Microsoft Windows, you issue the following command in a command prompt window:

```
MyPrompt> nornir_venv\Scripts\activate.bat
```

If you are using PowerShell, you use this command:

```
MyPrompt> nornir_venv\Scripts\Activate.ps1
```

If everything goes well, the virtual environment is activated, and the prompt changes to reflect this change by displaying `(nornir_venv)` before the cursor. You can now install any Python libraries by using `pip`, and they will be stored in the new folder structure that you just created.

When you are done with the virtual environment or when you want to stop using it and perhaps move to a different one, you have to deactivate it. You do so by issuing the following command in the terminal or command prompt where you have the virtual environment active:

```
(nornir_venv)MyPrompt% deactivate
```

This command deactivates the virtual environment, and the prompt reflects this change by removing `(nornir_venv)` before the cursor and returning the prompt to its previous state.

It is a best practice to use virtual environments whenever possible with Python. Using virtual environments creates

separation between different projects with different library versions requirements, and it also keeps the global Python installation clean and assists in resolving conflicting dependencies.

Appendix B. Understanding `expect_string` with `send_command()`

In [Part 1: Netmiko](#), you saw that `send_command()` global configuration mode statements can be used for configuration. However, the `send_config_set()` method is generally preferred for applying multiple configuration lines—especially when entering sub-modes, such as interface configuration mode.

The `send_command()` method can be used to enter sub-modes, such as interface configuration mode (`interface GigabitEthernet0/0/0`), but the `expect_string` parameter must be used to handle prompt changes. The `expect_string` parameter in Netmiko's `send_command()` method helps Netmiko determine when a command has completed execution by specifying a pattern to look for in the device's prompt.

[Example B-1](#) shows that when entering interface configuration mode (`'interface GigabitEthernet0/0/0'`), the prompt changes from `Router-R1(config)#` to `Router-R1(config-if)#`. Without `expect_string`, Netmiko expects the default global configuration prompt (`Router-R1(config)#`), causing a timeout.

Example B-1 *app_send_command_expect.py*

```
import netmiko

connection = netmiko.ConnectHandler(ip='192.168.1.1',
                                    device_type='cisco_ios',
                                    username='admin',
                                    password='cisco',
                                    secret='spot',
                                    session_log='my_session_log.')

# send_command() for configuration is similar to CLI
# secret password is required
connection.enable()

# Global config mode
connection.config_mode()

# Interface sub-mode with exit
connection.send_command('interface GigabitEthernet0/0/0',
                        expect_string=r'Router-R1\(config-if\)#')
connection.send_command('description Configured via Netmiko',
                        expect_string=r'Router-R1\(config-if\)#')
connection.send_command('exit',
                        expect_string=r'Router-R1\(config\)#')

# Exit global config mode
connection.exit_config_mode()

# Verify
print(connection.send_command(
    'show running-config | begin interface GigabitEthernet0/

connection.disconnect()
```

In Python, parentheses (`( )`) are used to indicate the beginning and end of a function call, such as **in** `print("Hello")` or `send_command("show version")`. However, sometimes you need to include an actual parenthesis character inside a string, especially when working with patterns like **expect_string** in Netmiko.

Because parentheses have a special meaning in Python and regular expressions, you need to escape them by adding a backslash (`\`) before them to tell Python to treat them as regular characters rather than as part of the function syntax:

- `\(` is the same as `"("` and will be printed as `(`.
- `\)` is the same as `)"` and will be printed as `)`.

In [Example B-1](#), `expect_string=r'Router-R1\(config-if)#` includes the following parts:

- `r'...'`: Creates a raw string, ensuring that backslashes are treated literally.
- **Router-R1**: Matches the device hostname.
- `\(`: Matches the literal opening parenthesis `(` character.
- **config-if**: Matches **config-if**, which is part of the interface configuration mode prompt.
- `\)#`: Matches the closing parenthesis `)` and `#` symbol at the end of the prompt.

Appendix C. Using Python Dictionaries as NAPALM Outputs

Creating a Dictionary to Simulate Output

When working through the examples in this book, the preferred approach is to retrieve live data directly from network devices by using NAPALM. However, in some situations, it may not be possible or practical to use real equipment. For example, you might be experimenting with Python code on a personal system, developing examples offline, or teaching and learning in environments that lack dedicated lab hardware. In these cases, Python dictionaries that mirror the structure of NAPALM's returned data provide a useful alternative.

By using predefined dictionaries that match the format of actual NAPALM method outputs, you can write, test, and understand Python programs exactly as if the data were from a live device. This allows you to focus on Python logic and data handling while maintaining consistency with real-world NAPALM workflows.

In [Example C-1](#), the dictionary **device_facts** is populated dynamically in a way that is similar to using live data from **device.get_facts()**.

Example C-1 NAPALM's *get_facts()* Method

```
from pprint import pprint

device_facts = {
    'fqdn': 'Router-R1.SSH-KEY.com',
    'hostname': 'Router-R1',
    'interface_list': [ 'GigabitEthernet0/0/0',
                        'GigabitEthernet0/0/1',
                        'GigabitEthernet0/0/2',
                        'GigabitEthernet0'],
    'model': 'ISR4331/K9',
    'os_version': 'ISR Software (X86_64_LINUX_IOSD-  
Version 16.6.3, RELEASE SOFTWARE  
'serial_number': 'FLM2229W1R6',
    'uptime': 2820.0,
    'vendor': 'Cisco'
}

pprint(device_facts)
```

Examples C-2 through C-7 provide other dictionary examples you can use.

Example C-2 NAPALM's *get_environment()* Method

```
{'cpu': {0: {'%usage': 1.0}},
 'fans': {'invalid': {'status': True}},
 'memory': {'available_ram': 1845474744, 'used_ram': 282094448},
 'power': {'invalid': {'capacity': -1.0, 'output': -1.0, 'status': True},
 'temperature': {'invalid': {'is_alert': False,
```

Example C-3 NAPALM's `get_interfaces()` Method

[illegible]

```
'mtu': 1500,  
'speed': 1000.0}}
```

Example C-4 *NAPALM's **get_interfaces_ip()** Method*

```
{'GigabitEthernet0/0/0': {'ipv4': {'192.168.1.1': {'prefix_length':  
                                '24'},  
                                'ipv6': {'2001:db8:c0de:1::1': {'prefix_length':  
                                                                '64'},  
                                'fe80::1:1': {'prefix_length':  
                                                '64'}}},  
'GigabitEthernet0/0/1': {'ipv4': {'192.168.2.1': {'prefix_length':  
                                '24'},  
                                'ipv6': {'2001:db8:c0de:2::1': {'prefix_length':  
                                                                '64'},  
                                'fe80::2:1': {'prefix_length':  
                                                '64'}}}}
```

Example C-5 *NAPALM's **get_users()** Method*

```
{'admin': {'level': 1, 'password': 'cisco', 'sshkeys': []},  
 'admin2': {'level': 1,  
            'password': '$1$NeqH$H823UIm16zVGclLX7ag71',  
            'sshkeys': []}}
```

Example C-6 *NAPALM's **get_arp_table()** Method*

```
[{'age': -1.0,  
  'interface': 'GigabitEthernet0/0/0',  
  'ip': '192.168.1.1',  
  'mac': '2C:73:A0:0A:6A:50'},  
 {'age': 0.0,  
  'interface': 'GigabitEthernet0/0/0',  
  'ip': '192.168.1.10',  
  'mac': '00:E0:4C:68:05:B6'},  
 {'age': -1.0,  
  'interface': 'GigabitEthernet0/0/1',
```

```
'ip': '192.168.2.1',  
'mac': '2C:73:A0:0A:6A:51'}]
```

Example C-7 *NAPALM's `get_mac_address_table()` Method*

```
[{'active': True,  
  'interface': '',  
  'last_move': -1.0,  
  'mac': '01:00:0C:CC:CC:CC',  
  'moves': -1,  
  'static': True,  
  'vlan': 0},  
{'active': True,  
  'interface': '',  
  'last_move': -1.0,  
  'mac': '01:00:0C:CC:CC:CD',  
  'moves': -1,  
  'static': True,  
  'vlan': 0},  
{'active': True,  
  'interface': '',  
  'last_move': -1.0,  
  'mac': '01:00:0C:CC:CC:CE',  
  'moves': -1,  
  'static': True,  
  'vlan': 0},  
<output omitted>  
{'active': True,  
  'interface': '',  
  'last_move': -1.0,  
  'mac': '01:80:C2:00:00:10',  
  'moves': -1,
```

```
'static': True,  
'vlan': 0}]
```

How to Use These Dictionaries

The dictionaries shown in this appendix are intended to be used as direct stand-ins for data returned by NAPALM methods. In a Python program, you can assign one of these dictionaries to a variable (for example, **device_facts**) and then access its keys exactly as you would if the data had been retrieved from a live network device.

Because the structure of each dictionary matches the output format of the corresponding NAPALM method, any code that works with real NAPALM data will also work with these predefined examples. This makes them especially useful for learning, testing logic, and experimenting with Python without requiring access to physical or virtual network equipment.

Important Note About These Examples

The values shown in the dictionaries throughout this appendix are representative examples and are not meant to be complete or exhaustive. Actual NAPALM outputs may include additional keys, different values, or platform-specific variations, depending on the device type, operating system, and configuration.

When writing production code, you should always account for the possibility that certain keys may be missing or that values may differ. However, for learning and experimentation purposes, the dictionaries provided here

accurately reflect the structure and data types returned by common NAPALM methods.

Mini Example: Switching Between Live NAPALM Data and a Dictionary

[Example C-8](#) shows how a Python program can retrieve device facts either from a live network device using NAPALM or from a predefined dictionary, like the ones shown in this appendix. The only difference between the two methods is how the **device_facts** variable is populated.

Example C-8 *Using Live NAPALM Data or a Dictionary*

```
from pprint import pprint

# -----
# Option 1: Use a predefined dictionary
# (Uncomment this section to run offline)
# -----

device_facts = {
    'fqdn': 'Router-R1.SSH-KEY.com',
    'hostname': 'Router-R1',
    'interface_list': [
        'GigabitEthernet0/0/0',
        'GigabitEthernet0/0/1',
        'GigabitEthernet0/0/2',
        'GigabitEthernet0'
    ],
    'model': 'ISR4331/K9',
    'os_version': 'IOS XE 16.6.3',
    'serial_number': 'FLM2229W1R6',
    'uptime': 2820.0,
```

```

        'vendor': 'Cisco'
    }

# -----
# Option 2: Use live NAPALM data
# (Uncomment this section when hardware
# or a virtual device is available)
# -----

"""
from napalm import get_network_driver

driver = get_network_driver("ios")
device = driver(
    hostname="192.168.1.1",
    username="admin",
    password="cisco"
)

device.open()
device_facts = device.get_facts()
device.close()
"""

# -----
# Program logic (identical in both cases)
# -----

print(f"Hostname: {device_facts['hostname']}")
print(f"Model: {device_facts['model']}")
print(f"OS Version: {device_facts['os_version']}")
print(f"Uptime (seconds): {device_facts['uptime']}")

```



```
print("\nInterfaces:")  
pprint(device_facts['interface_list'])
```

Appendix D. The Relationship Between Python Dictionaries and JSON

If you want to continue your journey in network automation and programmability beyond this book, understanding Python dictionaries will also help you become familiar with JavaScript Object Notation (JSON). JSON is a widely used data format in APIs and automation that is used by network automation tools like RESTCONF and NETCONF.

Python dictionaries are structured in a way that is very similar to JSON. Both Python dictionaries and JSON use key/value pairs, curly braces (`{}`) to enclose data, and colons (`:`) to separate keys from values. One key difference is that Python dictionaries allow both single quotes (`'`) and double quotes (`"`) for keys and string values, whereas JSON requires double quotes (`"`). If you try to use single quotes in JSON, it will not be valid JSON syntax and will cause an error when parsing.

This is a good example of how understanding Python dictionaries in NAPALM helps when learning other structured data formats used in automation, such as JSON in RESTCONF and NETCONF.

Note

While it does not matter functionally, using double quotes is sometimes preferable for consistency with JSON and APIs. We have chosen to use single quotes in this book to make it clear that we are working with Python dictionaries and not JSON files. However, the key is consistency: Whichever style you choose, try to stick with it throughout your project.

Appendix E. Understanding Objects and Variables in Python

This appendix is for readers who are familiar with object-oriented programming (OOP) or who want to deepen their understanding of Python objects and variables. While not required to use NAPALM, this knowledge can provide helpful insights into how Python manages data in memory. If you're already comfortable with these concepts or this is too much detail right now, feel free to skip this appendix.

In Python, an object is a data structure that represents a specific instance of a class or data type, and everything is an *object*—including variables of all types. Objects have *attributes* (data stored in them) and *methods* (functions that operate on object data). Even functions like **print()** are objects, which means they can be assigned to variables, passed as arguments, and called dynamically.

For example, say that **device** is an object created using the NAPALM driver. This object provides structured interaction with a network device and includes built-in methods such as **get_facts()**, **open()**, and **close()** to retrieve or modify device information.

A *variable* in Python, on the other hand, is simply a reference to an object that is stored in memory. A variable

does not store the object itself; instead, it acts as a label that points to the object. Consider the following line:

```
device_facts = device.get_facts()
```

Each time **device.get_facts()** is called, it generates a new dictionary that contains fresh data from the device. The variable **device_facts** stores a reference to this dictionary, allowing access to its key/value pairs. Modifications to **device_facts** affect only the local copy stored in memory, not the original device data. If **device.get_facts()** is called again and reassigned, **device_facts** will now reference the latest dictionary, replacing the previous one.

There are some differences between an object and a variable:

- **Object (for example, device):** The **device** object is an instance of the NAPALM driver that maintains a connection to the network device. It provides methods such as **open()**, **close()**, and **get_facts()** for retrieving or modifying device information. In Python, objects are stored in memory and have specific attributes and behaviors.
- **Variable (for example, device_facts):** A variable is a label that references an object in memory, not the object itself. In this case, **device_facts** is a variable that points to the dictionary returned by **device.get_facts()**. Because Python variables reference objects rather than contain them directly, assigning **device_facts = device.get_facts()** does not copy the dictionary; it simply makes **device_facts** point to it.

If all of this makes your head spin, don't worry about it. It is not necessary to understand this level of detail about object-oriented programming to use structured data

effectively. There are many excellent resources that can help you understanding OOP, including the excellent YouTube video *Python Object Oriented Programming (OOP) - For Beginners*, at https://www.youtube.com/watch?v=JeznW_7DIB0.

Appendix F. Using a Recursive Function to Handle Nested Dictionaries of Any Depth

The examples in [Part 2](#), “[NAPALM](#),” use simple, explicitly written loops to process structured data returned by NAPALM. This approach is intentional, to keep the logic easy to follow for readers who are still becoming comfortable with Python.

For readers who wish to explore a more flexible and reusable technique, this appendix provides an example of using a recursive function to process nested dictionaries and lists of arbitrary depth. *Recursion* is a programming technique in which a function calls itself in order to solve a problem by breaking it into smaller, similar pieces.

Rather than assuming a fixed structure, the function shown in [Example F-1](#) automatically walks through the data and prints each key/value pair, regardless of how deeply it is nested. This technique can be useful when working with complex or unfamiliar data structures or when you want a single function that adapts to many different NAPALM outputs.

Example F-1 *Using a Recursive Function for Nested Dictionaries*

```
import napalm
from pprint import pprint

# Recursive function to handle any depth of nested dictionaries/
def print_nested_data(data, indent=0):
    indent_str = "    " * indent # Indentation for formatting

    if isinstance(data, dict):
        for key, value in data.items():
            print(f"{indent_str}{key}:")
            print_nested_data(value, indent + 1)
    elif isinstance(data, list):
        for item in data:
            print_nested_data(item, indent + 1)
    else:
        print(f"{indent_str}{data}")

# --- Main Program ---
driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret': 'spot'}
)

device.open()

device_environment = device.get_environment()
```



```
pprint(device_environment) # Still useful for raw structured vi  
  
print("\nKey-Value pairs with nested dictionaries and lists:")  
print_nested_data(device_environment) # Recursive print  
  
device.close()
```

Appendix G. Using Tabular Output

When you're working with structured data returned by NAPALM, the raw dictionary format is useful for programmatic access but can be difficult to read when displayed directly. In some cases—such as with validation, troubleshooting, or presenting results to others—it can be helpful to format this data in a more human-friendly way.

[Example G-1](#) demonstrates how to format interface data returned by NAPALM into a table by using the `tabulate` library. As you can see in the output in [Example G-2](#), this approach allows you to quickly compare values across interfaces and produces clean, readable output without requiring manual alignment of columns.

In this example:

- Attribute names are extracted from one of the nested dictionaries to create column headings.
- Data rows are built by iterating through each interface and its associated attributes.
- The **`tabulate()`** function formats the data into a structured table using the grid style.

The internal details of how `tabulate` works are beyond the scope of this book. The purpose of this example is to show

one simple way to present nested dictionary data in a clear tabular format.

Example G-1 *Using the tabulate Function*

```
import napalm
from tabulate import tabulate # Import tabulate for table format

driver = napalm.get_network_driver('ios')

device = driver(
    hostname='192.168.1.1',
    username='admin',
    password='cisco',
    optional_args={'secret':'spot'}
)

device.open()

device_interfaces = device.get_interfaces ()

# Extract headers from the first dictionary (keys from nested dict)
headers = ["Interface"] + list(list(device_interfaces.values())[0].keys())

# Prepare table data
table_data = []
for interface, attributes in device_interfaces.items():
    # Append each row
    table_data.append([interface] + list(attributes.values()))

# Print the table
print(tabulate(table_data, headers=headers, tablefmt="grid"))
```

```
device.close()
```

Example G-2 Output from [Example G-1](#)

Interface	description	is_enabled	is_up	last_flapped	mac_address	mtu	speed
GigabitEthernet0		False	False	-1	DC:F7:19:94:A8:BF	1500	1000
GigabitEthernet0/0/0		True	True	-1	DC:F7:19:94:A8:30	1500	100
GigabitEthernet0/0/1		True	True	-1	DC:F7:19:94:A8:31	1500	1000
GigabitEthernet0/0/2		False	False	-1	DC:F7:19:94:A8:32	1500	1000

Appendix H. Using Public and Private Keys

Most of the examples in this book use a username and password to authenticate to network devices. While this approach is simple and effective for learning, many real-world environments use *public and private key authentication* instead of (or in addition to) passwords.

Public key authentication allows you to prove your identity to a device using a cryptographic key pair. The *private key* remains securely stored on your local system, while the *public key* is placed on the remote device. When you connect, the device verifies that you possess the matching private key—without requiring a password to be sent over the network.

This appendix provides a simple example of using an SSH private key with Netmiko. The goal is not to fully explain SSH key management but to show how Netmiko can be configured to use an existing key pair for authentication.

Public Versus Private Keys: A Brief Overview

Let's look at the roles of public and private keys and how they work together during SSH authentication:

- **Private key:** A private key is stored on your local machine, and it must be kept secure (meaning it must not be shared). You use the private key to prove your identity during SSH authentication.
- **Public key:** A public key is copied to the remote device (for example, a router) and can be shared freely. The device uses the public key to verify the client that is connecting.

Public and private keys are mathematically related, but the private key cannot be derived from the public key.

Basic Steps to Use SSH Keys with Netmiko

The high-level process for using public and private keys is as follows:

1. **Generate an SSH key pair.** On your local system, generate a public/private key pair (commonly stored in `~/.ssh/`). This step is usually done once.
2. **Install the public key on the device.** The public key must be configured on the network device and associated with a user account.
3. **Reference the private key in Netmiko.** Your Python program points Netmiko to the private key file when establishing the connection.
4. **Connect without a password prompt.** Authentication succeeds if the private key matches the public key stored on the device.

[Example H-1](#) demonstrates how Netmiko can authenticate using a private key file instead of a password.

Example H-1 *Using an SSH Private Key with Netmiko*

```
# Import Netmiko library
import netmiko

# Prompts and returns SSH username and password (not shown)
username_entered = input('Enter SSH username-key: ')

devices = ['2001:db8:c5e:1::1']

for device in devices:

    key_file_path = "/Users/myname/.ssh/id_rsa"

    # ip = current IP address in list of devices
    connection = netmiko.ConnectHandler(ip=device,
                                         device_type='cisco_ios',
                                         username=username_entered,
                                         key_file='/Users/myname/.ssh/id_rsa',
                                         disabled_algorithms = {
                                             'rsa-sha2-256', 'rsa-sha2-512'
                                         })

<Rest of code omitted>
```

These are the key parameters in the example:

- **key_file:** Specifies the path to the private key file used for authentication.
- **Username:** The username on the remote device that is associated with the public key.

- **disabled_algorithms:** Provides compatibility with certain devices and SSH implementations that may not support newer RSA signature algorithms.

Why Use SSH Keys?

Using public and private keys provides several advantages:

- Eliminates the need to store plaintext passwords in code
- Reduces exposure to password-based attacks
- Aligns more closely with production security practices
- Works well with automation tools like Netmiko and Nornir

For learning purposes, passwords are often sufficient. As you move toward more secure or production-like environments, it becomes more important to understand SSH keys.

Public Key Configuration Example (Cisco IOS)

On Cisco IOS and IOS XE devices, an SSH public key is associated with a *local user account*. The public key is stored directly in the device configuration and is used to authenticate incoming SSH connections.

The following example shows a simplified Cisco IOS configuration where a public key is installed for a user named admin:

```
username admin privilege 15
username admin sshkey ssh-rsa
    AAAAB3NzaC1yc2EAAAADAQABAAQ3C3...
```



In this example:

- **username (admin)** must match the username used by Netmiko.
- The public key (which begins with **ssh-rsa**) is pasted directly into the device configuration.
- The device uses this public key to verify the private key presented by the client during SSH authentication.

Once the public key is installed, the device no longer needs a password for SSH access when the correct private key is used.

Keep in mind these important notes:

- The private key never leaves your local system.
- Only the public key is stored on the router.
- Multiple public keys can be associated with different users, if needed.
- File permissions on the private key (for example, `~/.ssh/id_rsa`) should restrict access to the owner only.

This configuration is what allows Netmiko to authenticate using the `key_file` parameter shown earlier in this appendix. As long as the username and key pair match, the SSH connection can be established without requiring that a password be sent over the network.

This appendix is intended as a practical reference. If you already have SSH keys configured on your system and devices, Netmiko can use them with only a small change to your connection parameters. The rest of your automation program remains unchanged.

Appendix I. Netmiko-Supported Network Operating Systems

Netmiko is designed to support a wide range of network devices across many different vendors and platforms. You specify the device type in the **device_type** parameter when creating a connection using **ConnectHandler**. This appendix lists commonly supported network operating systems, organized by vendor.

Netmiko supports a number of popular platforms that you can specify in the **device_type** parameter:

- Cisco:
 - **cisco_ios**: Cisco IOS (routers/switches)
 - **cisco_xe**: IOS XE (same as IOS, but newer codebase)
 - **cisco_xr**: IOS XR (service provider routers)
 - **cisco_nxos**: NX-OS (data center switches like Nexus)
 - **cisco_asa**: ASA firewalls
 - **cisco_wlc**: Wireless LAN Controllers (older AireOS)
 - **cisco_s300**: Small business switches (SG series)
- Arista:

- **arista_eos:** Arista EOS (Extensible Operating System)
- Juniper:
 - **juniper_junos:** Junos OS (used on routers and switches)
- HP / Aruba:
 - **hp_procurve:** HP ProCurve switches
 - **aruba_os:** Aruba Mobility Controllers
- Dell:
 - **dell_os10:** OS10
 - **dell_force10:** Force10
- Ubiquiti:
 - **ubiquiti_edge:** EdgeOS (EdgeRouters)
- F5:
 - **f5_ltm:** BIG-IP Local Traffic Manager (via SSH)
- Palo Alto:
 - **paloalto_panos:** PAN-OS (firewalls)
- Checkpoint:
 - **checkpoint_gaia:** Gaia OS
- Fortinet:
 - **fortinet:** FortiOS (via SSH, not API)
- Linux/Unix (for automation of generic devices):
 - **linux:** Generic Linux server
 - **generic:** When all else fails (basic SSH)

[Example I-1](#) shows how to get a complete list of network operating systems supported by the version of Netmiko you are using from Python interactive mode.

Example I-1 *Obtaining the Current List of Netmiko-Supported Network Operating Systems*

```
MyPrompt% python3
Python 3.10.3 (v3.10.3:a342a49189, Mar 16 2022, 09:34:18) [Clang
darwin
Type "help", "copyright", "credits" or "license" for more inform
>>>
from netmiko import ConnectHandler
from netmiko.ssh_dispatcher import CLASS_MAPPER_BASE
print(CLASS_MAPPER_BASE.keys())

<List of supported network operating systems>

>>> ^D
MyPrompt%
```