

Hands-On Cisco Automation with Python

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



**RICK GRAZIANI
ADRIAN ILIESIU**

ciscopress.com

*Includes endorsements by networking experts from Cisco Systems,
Cisco Networking Academy, and industry leaders*

FREE SAMPLE CHAPTER |



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
Adrian Iliesiu

Copyright© 2026 Pearson Education, 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 authors 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: 978-0-13-546319-2

ISBN-10: 0-13-546319-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: Timothy Wright

Proofreader: Barbara Mack



Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

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

Europe Headquarters
Cisco Systems International BV Amsterdam,
The Netherlands

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

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: www.cisco.com/go/trademarks. Third party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

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, have 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 scribbles, 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

	Introduction	xxiii
Chapter 1	Introducing Netmiko, NAPALM, and Nornir	1
Part 1	Netmiko	
Chapter 2	Getting Started with Netmiko	9
Chapter 3	Configuring Devices with Netmiko	35
Chapter 4	Accessing Multiple Devices with Netmiko	53
Part 2	NAPALM	
Chapter 5	Introducing NAPALM and Structured Data	85
Chapter 6	Understanding Python Dictionaries with NAPALM	109
Chapter 7	Iterating Through NAPALM Dictionaries	137
Chapter 8	Configuring Devices with NAPALM	179
Part 3	Nornir	
Chapter 9	Introducing Nornir: A Pythonic Framework for Network Orchestration	203
Chapter 10	Using Nornir with Netmiko	221
Chapter 11	Using Nornir with NAPALM	247
Chapter 12	Inventory Management with Nornir	277
Part 4	What's Next	
Chapter 13	What's Next	321
Appendix A	Python Virtual Environments	341
Appendix B	Understanding <code>expect_string</code> with <code>send_command()</code>	345
Appendix C	Using Python Dictionaries as NAPALM Outputs	347
Appendix D	The Relationship Between Python Dictionaries and JSON	353
Appendix E	Understanding Objects and Variables in Python	355
Appendix F	Using a Recursive Function to Handle Nested Dictionaries of Any Depth	357

Appendix G	Using Tabular Output	359
Appendix H	Using Public and Private Keys	361
Appendix I	Netmiko-Supported Network Operating Systems	365
	Index	367

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

	Introduction	xxiii
Chapter 1	Introducing Netmiko, NAPALM, and Nornir	1
	The 3 <i>N</i> s: Netmiko, NAPALM, and Nornir	1
	Part 1: Netmiko	1
	Part 2: NAPALM	2
	Part 3: Nornir	3
	Part 4: What's Next?	3
	Automation and Programmability for All Levels	4
	How Much Cisco IOS and Python Do I Need to Know?	4
	What Do I Need to Get Started?	5
	Using AI as an Alternative to Physical Equipment	6
	Summary	7
Part 1	Netmiko	
Chapter 2	Getting Started with Netmiko	9
	Why Start with Netmiko?	9
	A First Look at a Netmiko Program	10
	Installing the Netmiko Library	11
	The Basic Netmiko Framework	11
	Step 1: Import the Netmiko Library	13
	Step 2: Establish an SSH Connection to the Device	13
	Step 3: Execute an IOS Command and Display the Output	13
	The <code>send_command()</code> Method	14
	Step 4: Close the SSH Connection	15
	Optional: Exploring Python Classes, Objects, Instances, and Methods	15
	Your Turn	17
	Sending Command Output to a Variable	19
	<i>Including the IOS Command in the Output</i>	20
	<i>Including the IOS Prompt and the Command in the Output</i>	21
	The <code>send_command_expect()</code> Method	23
	The <code>save_config()</code> Method	26
	Debugging Netmiko by Using the Session Log	28
	Missing the Secret Parameter	28

	Requiring Privileged EXEC Mode Access	30
	Using Python Interactive Mode to Experiment with Netmiko	33
	Summary	34
Chapter 3	Configuring Devices with Netmiko	35
	The send_command() Method	35
	Using send_command() for Retrieving Information	36
	Using send_command() for Individual Configuration Commands	37
	Using send_command() for Multiple Configuration Commands	39
	Using send_config_set() for Configuration Commands	39
	Configuring and Displaying Commands with send_config_set()	41
	Sending Commands from a File with send_config_from_file()	44
	Ensuring That Python Uses the Current Working Directory	48
	Summary	50
Chapter 4	Accessing Multiple Devices with Netmiko	53
	Using a Variable to Store an IP Address	53
	Understanding Python for Loops	54
	How a for Loop Works, Step-by-Step	55
	Python F-Strings	56
	Iterating Through Multiple Devices	56
	Using a for Loop with Netmiko, Step-by-Step	59
	<i>Step 1: Define the List of Device IP Addresses</i>	59
	<i>Step 2: Start the for Loop to Iterate over Devices</i>	59
	<i>Step 3: Display the IPv4 Routing Table</i>	60
	Example: Using a for Loop to Configure IPv6 Addresses	60
	Using Dictionaries to Store Device Connection Parameters	65
	Understanding Dictionary Unpacking in Python (**device)	67
	Extracting and Displaying the Device IP Address	69
	An Alternative Way to Store Device Dictionaries	69
	Using Python's getpass() for Secure Password Input	71
	Using Netmiko Exceptions for Troubleshooting	73
	Maintaining Multiple SSH Connections Simultaneously	77
	Summary	81

Part 2 NAPALM

Chapter 5 Introducing NAPALM and Structured Data 85

What Is NAPALM, and How Is It Different from Netmiko?	85
Installing the NAPALM Library	88
Basic NAPALM Framework	88
How NAPALM Retrieves Data Without an API on Cisco IOS	91
Good News, Bad News: Understanding Structured Data	92
The Basics of a Python Dictionary	92
How Python Formats a Dictionary	94
get_facts(): Our First NAPALM Method and Dictionary	95
Assigning the Dictionary to a Variable	97
Understanding Different Types of Values in a Dictionary	99
Creating a NAPALM Dictionary Without a Device	100
NAPALM Methods	102
General Device Information	103
Interface and Networking Information	103
Routing and Network Instances	103
Layer 2 and Neighbor Discovery	104
Optical and NTP Information	104
Connectivity and Testing	104
Using the CLI	105
Configuration-Related Methods	105
Using Python Interactive Mode to Experiment with NAPALM	105
Summary	107

Chapter 6 Understanding Python Dictionaries with NAPALM 109

How NAPALM Organizes Data	110
A Single Dictionary	110
get_facts(): A Single Dictionary	110
<i>Viewing get_facts() as a Table</i>	111
get_environment(): A Single Dictionary	112
Why Use Nested Dictionaries?	114
A Dictionary of Dictionaries	115
get_interfaces(): A Dictionary of Dictionaries	115
<i>Keys, Not Values</i>	117
<i>Viewing a Dictionary of Dictionaries as a Table</i>	118

Why Use a Dictionary of Dictionaries?	119
get_interfaces_ip(): A Dictionary of Dictionaries	120
<i>A Structured Breakdown of the get_interfaces_ip() Output</i>	121
A List of Dictionaries	122
get_mac_address_table(): A List of Dictionaries	122
<i>Viewing a List of Dictionaries as a Table</i>	123
<i>A Closer Look at the get_mac_address_table() Method</i>	123
get_arp_table(): A List of Dictionaries	125
Sample Program Using All Three Types of Dictionaries	127
Comparing the Three Types of Dictionaries	131
Methods by Type of Data Structure	133
Single Main Dictionary	133
Dictionary of Dictionaries	133
List of Dictionaries	134
Summary	135
Chapter 7 Iterating Through NAPALM Dictionaries	137
Live or Simulated Data	138
Working with .keys(), .values(), and .items() in NAPALM Dictionaries	140
Using .keys() to Retrieve All Keys	142
Using .values() to Retrieve All Values	142
Using .items() to Retrieve Key/Value Pairs	142
Using a for Loop with Dictionary Methods	143
Using a for Loop with .keys() to Retrieve Keys	143
Using a for Loop with .values() to Retrieve Values	144
Using a for Loop with .items() to Retrieve Values	145
Summarizing the Three Dictionary Methods in a for Loop	145
Determining the Type of Value	147
Using the type() Function (Less Preferred Approach)	148
Using the isinstance() Function (Preferred Approach)	151
Looping Through Key/Value Pairs with .items() and Processing Values with isinstance()	153
When the Dictionary Value Is a List	154
<i>A Step-by-Step Explanation</i>	156
<i>Summary of Loop Behavior</i>	158
When the Dictionary Value Is Another Dictionary	158

	<i>Working with a Three-Level Nested Dictionary</i>	162
	<i>Step-by-Step Explanation: Three-Level Nested Dictionary</i>	163
	<i>Accessing Specific Values in a Nested Dictionary</i>	166
	Iterating Through a Dictionary of Dictionaries	167
	Iterating Through a List of Dictionaries	173
	Summary	176
Chapter 8	Configuring Devices with NAPALM	179
	A Quick Overview of NAPALM Configuration Methods	179
	Introducing Our Example Scenario	180
	Configuring a Device with <code>load_merge_candidate()</code>	182
	Understanding the Relationships Between Configuration Methods	185
	Saving the Configuration	186
	Using the <code>cli()</code> method	188
	Understanding the Code and Output	190
	<i>Using the <code>pprint</code> Command</i>	190
	<i>Storing the Output to a Variable</i>	191
	<i>Accessing the Value of the Key “show ip interface brief”</i>	191
	<i>Accessing the Key “show ip interface brief”</i>	192
	The <code>cli()</code> Method is Read-Only	192
	Using <code>load_replace_candidate()</code> to Replace the Configuration	193
	Cisco IOS Archive Feature	193
	Using a New Configuration File to Change the Hostname	193
	Displaying Running and Startup Configuration Files	199
	Next Step: Nornir	200
	Summary	201
Part 3	Nornir	
Chapter 9	Introducing Nornir: A Pythonic Framework for Network Orchestration	203
	What Is Orchestration?	204
	How Does Nornir Compare to Netmiko and NAPALM?	205
	How Nornir Uses Netmiko and NAPALM	206
	Installing Nornir	207
	Basic Nornir Framework: Python and YAML	207
	Your First Nornir Program and YAML Files	208
	The Python File	209

	The config.yaml File	210
	The hosts.yaml File	211
	The groups.yaml File	213
	The defaults.yaml File	215
	How These YAML Files Work Together	216
	The Output	216
	What's Next?	218
	Summary	219
Chapter 10	Using Nornir with Netmiko	221
	Installing Support for Netmiko: <code>nornir_netmiko</code>	221
	Using Nornir and <code>netmiko_send_command()</code>	222
	Parallel Execution with Nornir	224
	Understanding Workers in Nornir	226
	The Python Code: A Closer Look	226
	Understanding How Nornir Displays the Output	228
	<i>What Is an AggregatedResult Object?</i>	229
	<i>Displaying the Results</i>	230
	Visualizing How Nornir Executes Tasks	233
	Sending Configuration Commands with <code>netmiko_send_config</code>	236
	Using <code>netmiko_send_config</code> with a Single Shared List of Commands	236
	<i>How It Works</i>	237
	<i>Changes to defaults.yaml</i>	238
	Using <code>netmiko_send_config</code> with <code>hosts.yaml</code> for per-Device Configuration	240
	<i>How It Works</i>	242
	<i>Understanding the User-Defined Function</i>	243
	Summary	246
Chapter 11	Using Nornir with NAPALM	247
	A Quick Review	247
	A Quick Review of NAPALM	247
	A Quick Review of Nornir	248
	Nornir and NAPALM: Why Combine Them?	248
	Installing Support for NAPALM: <code>nornir_napalm</code>	249
	<code>nornir_napalm</code> Tasks	250
	Advantages of Using These Tasks with Nornir	251
	Using Nornir and the <code>napalm_cli</code> Task	251

napalm_cli Task Example	252
The Output	255
High-Level Explanation of Example 11-1: What the Program Does	257
Breaking Down the Code	259
Using Nornir and the napalm_get Task	261
Introducing the napalm_get Task Example	263
The Output from Example 11-7	264
High-Level Explanation of Example 11-7: What the Program Does	266
Using Nornir and the napalm_configure Task	268
Understanding the napalm_configure Parameters	269
Introducing the napalm_configure Example	271
High-Level Explanation of Example 11-10: What the Program Does	272
The Output	274
Simplified Workflow of the napalm_configure Task (Optional)	275
Summary	276

Chapter 12 Inventory Management with Nornir 277

A Quick Overview with a Focus on Inventory	278
Inheritance	279
Where Inventory Data Comes From	280
A Note on Filtering	281
Inventory Management Core Architecture	282
Hosts	282
Setup: Accessing a Host in Python	283
Host Name (Nornir Device Identifier)	285
Hostname (IP Address or DNS Name)	285
Platform (Device Operating System)	286
Port (SSH Connection Port)	286
Username and Password (Credentials)	287
Groups (Group Membership)	287
Data (Custom Metadata)	288
Why Hosts Matter	288
Summary Example	289
Groups	289
How Groups Work in Nornir	289
The cisco_devices Group	290
The core_routers Group	291

The access_switches Group	292
Effective Values After Inheritance	292
Effective Values for router1	293
Effective Values for switch1	294
Why Effective Values Matter	294
Using Group Data for Filtering	295
Inspecting a Filtered Inventory	295
Task Logic	296
Defaults	296
Accessing Inventory Data	298
Accessing Built-in Host Attributes	299
Accessing Custom Inventory Data	299
Why This Matters	300
Inventory Plugins	300
Why We Need Inventory Plugins	300
Where Inventory Plugins Are Configured	301
Built-in Versus Third-Party Inventory Plugins	301
Inventory Plugin Options	301
The SimpleInventory Plugin	302
<i>Configuring SimpleInventory with config.yaml</i>	302
<i>Configuring SimpleInventory Directly in Python</i>	303
<i>When to Use SimpleInventory</i>	303
The DictInventory Plugin	304
<i>DictInventory Versus SimpleInventory</i>	305
<i>When DictInventory Is Useful</i>	305
The NetBox Plugin	306
<i>Why Use NetBox with Nornir?</i>	306
<i>How Nornir Uses NetBox</i>	307
<i>Installing and Configuring the NetBox Plugin</i>	307
<i>Hosts, Groups, and Defaults with NetBox</i>	308
<i>Accessing Inventory Data with NetBox</i>	309
<i>What Stays the Same</i>	309
<i>When to Use NetBox Inventory</i>	309
The Ansible Plugin	310
<i>Why Use Ansible Inventory with Nornir?</i>	310
<i>How Ansible Inventory Works with Nornir</i>	310

- Installing and Configuring the Ansible inventory Plugin* 311
- How Ansible Variables Map into Nornir* 311
- When to Use Ansible inventory* 312
- CSV and Excel (.xlsx) Plugin 312
- Why Use Spreadsheet-Based Inventory with Nornir?* 313
- How the CSV and Excel Plugin Works* 313
- How Spreadsheet Inventory Maps to Nornir* 313
- Installing and Configuring the Spreadsheet Plugins* 314
- When to Use CSV or Excel inventory* 315
- The InfraHub Plugin 316
- Why Use InfraHub with Nornir?* 316
- Installing and Configuring the InfraHub Plugin* 317
- How InfraHub Data Maps to Nornir Inventory* 317
- Accessing Inventory Data from InfraHub* 318
- When to Use InfraHub Inventory* 319

Summary 319

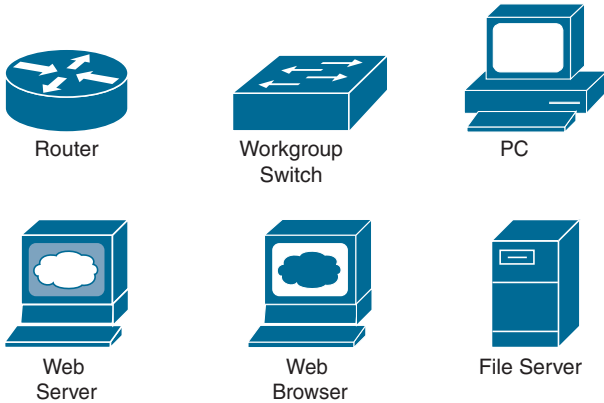
Part 4 What's Next

Chapter 13 What's Next 321

- Why Network Automation Became Necessary 321
 - What Changed 322
 - Automation as an Operational Requirement 322
 - SDN and IBN in Context 323
 - A Natural Evolution, Not a Replacement 323
- Netmiko, NAPALM, and Nornir in Context 324
- Ansible, NETCONF, and RESTCONF: Expanding the Automation Toolkit 325
 - Ansible 326
 - How Ansible Differs from Netmiko, NAPALM, and Nornir* 327
 - How Your Existing Knowledge Helps* 327
 - Why Someone Would Choose Ansible* 327
- RESTCONF 328
 - How RESTCONF Differs from Netmiko, NAPALM, and Nornir* 328
 - How Your Existing Knowledge Helps* 329
 - Why Someone Would Choose RESTCONF* 329
- NETCONF 329

<i>How NETCONF Differs from Netmiko, NAPALM, and Nornir</i>	330
<i>How Your Existing Knowledge Helps</i>	330
<i>Why Someone Would Choose NETCONF</i>	331
Comparisons	331
Understanding YANG Models	332
Which Tools Use YANG Models and Which Do Not	332
Types of YANG Models	333
Why YANG Models Matter	333
A Helpful Way to Think About YANG	334
Understanding APIs	334
APIs in Everyday Life	335
Why APIs Are Useful for Automation	335
How APIs Are Used in Practice	335
Which Tools in This Chapter Use APIs	335
What Is Required to Use an API	336
A Helpful Way to Think About APIs	336
Artificial Intelligence and Network Automation	336
AI as a Learning and Exploration Tool	337
How AI Is Changing Network Operations and Security	337
Using AI with Netmiko, NAPALM, and Nornir	338
A Final Thought on AI	338
Closing Thoughts	338
Appendix A Python Virtual Environments	341
Appendix B Understanding <code>expect_string</code> with <code>send_command()</code>	345
Appendix C Using Python Dictionaries as NAPALM Outputs	347
Appendix D The Relationship Between Python Dictionaries and JSON	353
Appendix E Understanding Objects and Variables in Python	355
Appendix F Using a Recursive Function to Handle Nested Dictionaries of Any Depth	357
Appendix G Using Tabular Output	359
Appendix H Using Public and Private Keys	361
Appendix I Netmiko-Supported Network Operating Systems	365
Index	367

Icons Used in This Book



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.

“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

“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

“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

“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

“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

“If you’ve ever wondered how to start basic automation for Cisco devices, you have the answer now. This book guides you through installing and using three popular Python frameworks. It covers how to connect to Cisco devices and extract information from them. You’ll also learn essential Python topics, like the various uses of dictionaries. If you’re new to Cisco automation, this book is an excellent starting point.”

Russ White, Ph.D., Nokia Principle Architect, IETF Routing Area Directorate

“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

“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

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.

Figure Credits

Epigraph (Chapters 2 & 5) Oswald, Matt et al, *Network programmability and automation: Skills for the next-generation network engineer*. Sebastopol, CA: O’Reilly Media, Inc, 2023.

Cover credit: Halim Karya Art/Adobe Stock

This page intentionally left blank

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          Protocol
GigabitEthernet0/0/0 192.168.1.1    YES NVRAM  up              up
GigabitEthernet0/0/1 192.168.2.1    YES NVRAM  up              up
GigabitEthernet0/0/2 unassigned      YES NVRAM  administratively down down
GigabitEthernet0    unassigned      YES NVRAM  administratively down down
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 password

# 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 route'))

# 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, GigabitEthernet0/0/0
L     192.168.1.1/32 is directly connected, GigabitEthernet0/0/0
     192.168.2.0/24 is variably subnetted, 2 subnets, 2 masks
C     192.168.2.0/24 is directly connected, GigabitEthernet0/0/1
L     192.168.2.1/32 is directly connected, GigabitEthernet0/0/1
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 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 password)

# 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 changes)
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 GigabitEthernet0/0/0')

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

# Close the SSH connection
connection.disconnect()

```

A list is a collection of items stored in a specific order and enclosed within square brackets `[]`. 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 GigabitEthernet0/0/0
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 password)

# 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 changes)
connection.enable()

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

# 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 file
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0/0')
print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0/0'))

# 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 ex3-10_send_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 GigabitEthernet0/0/0
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 changes)
connection.enable()

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

# 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 file
print('\nIOS command: '
      'show running-config | begin interface GigabitEthernet0/0/0')
print(connection.send_command(
      'show running-config | begin interface GigabitEthernet0/0/0'))

# 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.

Index

A

accessing inventory data, 298–299

AggregatedResult object, 229–230, 234–235

AI, 337–338

as a learning and exploration tool, 337

and network automation, 336–337

using as alternative to physical equipment, 6–7

using with Netmiko, NAPALM, and Nornir, 338

Ansible, 326, 327

comparing with Netmiko, NAPALM, and Nornir, 327

reasons for using, 327

Ansible plugin, 310–312

installing and configuring, 311

when to use, 312

Apache 2.0 License, 2–3

API/s, 3, 334–336

abstraction, 91–92

unified, 85–86

architecture, Nornir inventory system, 282

archive feature, Cisco IOS, 193

arguments, user-defined function, 244

assigning a dictionary to a variable, 97–98

automation, 4, 322, 323. *See also* devices; dictionaries; for loops

AI, 336–338

Ansible, 326

APIs, 334–336

comparison of tools, 331–332

need for, 321–322

as operational requirement, 322–323

orchestration, 204–205

structured data, 86–87

YANG models, 332–334

B

Barroso, David, 85, 203

built-in plugins, 301

Byers, Kirk, 2, 10, 85, 203

C

changing device hostname, 193–199

ChatGPT, using as alternative to physical equipment, 6–7

Cisco IOS

archive feature, 193

commands, 4

crypto key generate rsa general-keys modulus 1024, 5–6

executing using Netmiko, 13–14

ip domain-name, 5–6

ip scp server enable, 181

line vty, 6

login local, 6

show ip interface brief, 10, 12, 71

transport input ssh, 6

username admin password cisco, 5–6

username admin privilege 15 password 0 cisco, 181

configuring SSH, 5–6

public key configuration, 363–364

classes, Netmiko, ConnectHandler(), 15, 16, 81–82

cli() method, 105, 188–190, 192

accessing the key, 192

accessing the value of the key, 191–192

storing the output to a variable, 191

closing an SSH connection using Netmiko, 15

commands, 2. *See also* configuration commands; functions; output

Cisco IOS, 4

crypto key generate rsa general-keys modulus 1024, 5–6

executing using Netmiko, 13–14

ip domain-name, 5–6

ip scp server enable, 181

line vty, 6

login local, 6

sending to devices, 222–224

show ip interface brief, 10, 12, 71

transport input ssh, 6

username admin password cisco, 5–6

username admin privilege 15 password 0 cisco, 181

Netmiko. *See also* methods

ConnectHandler, 13

print, 12

variables, 13–14

pip3, 11

python3, 11

vendor-agnostic, 2

commit_config() method, 105, 180, 185

compare_config() method, 105, 180, 183–185

configuration commands

applying to a device, 39–41

sending with netmiko_send_config, 236

configuration-related methods, NAPALM, 105, 185

configuring

devices

IPv6, 182–185

netmiko_send_config, 236–243

InfraHub plugin, 317

NetBox plugin, 307–308

public keys, 363–364

SSH on Cisco IOS, 5–6

config.yaml, 210–211, 216, 284

- inventory section, 211
- nornir_napalm_cli.py, 254
- nornir_netmiko_send_command.py, 223
- num_workers option, 226
- runner section, 211

ConnectHandler() class, 15, 16, 81–82

- dictionary unpacking, 65–68
- session_log parameter, 28–33

connection variable, 38

connections

- SSH
 - maintaining multiple*, 77–81
 - troubleshooting using Netmiko exceptions*, 73–77
- testing, 104

credentials, hiding, 71–73

crypto key generate rsa general-keys modulus 1024 command, 5–6

CSV and Excel plugin, 312–314

- installing and configuring, 314–315
- when to use, 315–316

curly braces ({}), 94. See also dictionaries

D

data structure, 92

defaults.yaml, 215–216, 279, 285, 296–298

- nornir_napalm_cli.py, 255
- nornir_netmiko_send_command.py, 224

device_facts variable, 102

device.get_facts() method, 138–140

devices. See also IP addresses

- changing the hostname, 193–199
- configuration-related NAPALM methods, 105
- credentials, hiding, 71–73
- displaying running and startup configuration files, 199–200
- IPv6 configuration, 182–185
- Layer 2 and neighbor discovery, NAPALM methods, 104
- replacing a configuration, 193
- retrieving general information, NAPALM methods, 103
- retrieving interface and networking information, NAPALM methods, 103
- retrieving live data, 138–140
- retrieving optical and NTP information, NAPALM methods, 104
- retrieving routing and network instances, NAPALM methods, 103
- saving configuration changes, 186–188
- sending CLI commands to, 222–224

DictInventory plugin, 304–306

dictionaries, 2–3, 4, 79, 85. *See also key/value pairs; list of dictionaries*

- assigning to a variable, 97–98
- creating to simulate output, 347–352
- curly braces ({}), 94
- format, 94–95
- heterogenous, 153
- homogenous, 116
- JSON and, 353
- keys, 98
- key/value pairs, 92–93

NAPALM, 110–111, 137–138
 comparing, 131–133
 displaying as a table, 111–112
 nested, 112–115
 sample program, 127–130
 static structure, 110
 nested, 158
 accessing specific values,
 166–167
 three-level, 162–166
 predefined, 100–102, 138
 square brackets ([]), 97–99
 storing device connection details,
 69–70
 three-level, key/value pair iteration,
 162–166
 unpacking, 65–68
 values, 99–100
 dictionary of dictionaries, 115–116
 iterating, 167–173
 reasons for using, 119–120
 viewing as a table, 118–119
 discard_config() method, 105,
 180, 185
 disconnect() method, 15
 dot notation, 90
 dual-stack environment, 44

E

effective values, 292–294
 enable() method, 38
 ex2_1_my_first_netmiko_program.
 py, 10
 exceptions, Netmiko, 73–77
 exit_config_mode() method, 38
 expect_string parameter, 345–346

F

F-strings, 56
 filtering, Nornir inventory, 295
 for loops, 54–56, 79, 85, 145–147
 configuring IPv6 addresses, 60–65
 if-elif-else statements, 63–64
 iterating key-value pairs
 dictionary values, 158–162
 list values, 154–158
 three-level nested dictionary,
 162–166
 iterating through a dictionary of
 dictionaries, 167–173
 iterating through a list of dictionar-
 ies, 173–176
 iterating through dictionaries,
 158–162
 iterating through facts, configura-
 tions, or interface data, 145
 iterating through keys, 143–144
 iterating through lists, 99–100,
 156–157
 iterating through multiple devices,
 56–59
 define the list of device IP
 addresses, 59
 display the IPv4 routing
 table, 60
 start the loop to iterate over
 devices, 59
 iterating through values, 144
 nornir_netmiko_send_command.py,
 230-HOST
 format, dictionaries, 94–95
 framework
 NAPALM, 88–91
 Netmiko, 11–13

Nornir, 207–208

functions

NAPALM

get_network_driver(), 89–90

pprint(), 90–91, 154, 190–191

Python

getpass(), 71–73

isinstance(), 151–154

os.getcwd(), 49

print(), 153, 200

type(), 148–151

recursive, 357–358

user-defined, 243–245

argument, 244

parameter, 244

G

generative AI. *See* ChatGPT

get_arp_table() method, 104, 125–127, 134, 173–174

get_bgp_config() method, 103, 134

get_bgp_neighbors() method, 134

get_bgp_neighbors_detail() method, 104, 134

get_config() method, 103, 133, 199–200

get_environment() method, 103, 112–114, 133, 158

get_facts() method, 95–97, 103, 110–112, 132, 133, 147–148, 154–157

get_interfaces() method, 103, 115–116, 132–133, 134

get_interfaces_counters() method, 103, 134

get_interfaces_ip() method, 86–87, 103, 120–122, 134, 184

get_ipv6_neighbors_table() method, 104, 135

get_lldp_neighbors() method, 103, 134

get_lldp_neighbors_detail() method, 103

get_mac_address_table() method, 104, 122–125, 135

get_network_driver() function, 89–90

get_network_instances() method, 104, 134

get_ntp_peers() method, 104, 134

get_ntp_servers() method, 104, 134

get_ntp_stats() method, 104, 135

get_optics() method, 104, 135

get_route_to(destination) method, 104, 134

get_snmp_information() method, 103, 133

get_users() method, 103, 133

get_vlans() method, 104, 134

getpass() function, 71–73

getting started

equipment, 5

using AI as an alternative to physical equipment, 6–7

groups.yaml, 213, 216, 279, 284, 289–290

access_switches group, 292

cisco_device group, 290–291

core_routers group, 291–292

nornir_napalm_cli.py, 254

nornir_netmiko_send_command.py, 224

relationship to *hosts.yaml*, 214–215

subsections, 213–214

H

heterogenous dictionaries, 153
 hiding credentials, 71–73
 homogenous dictionary, 116, 117–118
 hosts.yaml, 211–212, 216, 279, 283–284, 288–289
 credentials, 287
 data, 288
 groups, 287
 host name, 285
 hostname, 285–286
 nornir_napalm_cli.py, 254
 nornir_netmiko_send_command.py, 224
 platform, 286
 port, 286

I

IBN (intent-based networking), 323
 if-elif-else statements, 63–64
 importing, Netmiko library, 13
 InfraHub plugin, 316, 317–318
 accessing inventory, 318–319
 installing and configuring, 317
 when to use, 319
 installing
 CSV and Excel plugin, 314–315
 InfraHub plugin, 317
 NAPALM, 88
 NetBox plugin, 307–308
 Netmiko, 11
 Nornir, 207
 nornir_napalm, 249–250
 nornir_netmiko, 221–222

interactive mode, Python, 30–33, 105–107
 inventory system, Nornir, 204. *See also* defaults.yaml; groups.yaml; hosts.yaml
 accessing built-in host attributes, 299
 accessing custom inventory data, 299–300
 accessing inventory, 298–299
 core architecture, 282
 effective values, 292–294
 filtering, 281–282, 295
 hosts, 282–283
 inheritance, 279–280
 inspecting a filtered inventory, 295–296
 plugins, 300–301
 Ansible, 310–312
 built-in versus third-party, 301
 DictInventory, 304–306
 InfraHub, 316–319
 NetBox, 306–310
 nornir_table_inventory, 312–316
 options, 301
 SimpleInventory, 302–304
 sources, 280–281
 task logic, 296
 IP addresses
 storing, 69–70
 storing in a variable, 53–54
 ip domain-name command, 5–6
 ip scp server enable command, 181
 IPv6 addresses, configuring with for loops, 60–65
 is_alive() method, 103, 133
 isinstance() function, 151–154

`.items()` method, 142–143, 145, 153–154
 iterating key-value pairs
 dictionary values, 158–162
 list values, 154–158

J-K

Jasinska, Elisa, 85

JSON (JavaScript Object Notation), 353

keys

dictionary, 98
 iterating through, 143–144
 public and private, 361
 SSH, 362–363

`.keys()` method, 140–142, 143–144

key/value pairs, 92–93, 115–116

homogenous dictionary, 117–118
 iterating, 153–154
 dictionary values, 158–162
 list values, 154–158
 three-level nested dictionary, 162–166

nested dictionary, 112–114

L

Layer 2 discovery, NAPALM methods, 104

libraries. *See also* NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support); Netmiko; Nornir

commands. *See* commands
 Python, Paramiko, 10

line vty command, 6

list of dictionaries, 122–123

 ARP table, 125–127

 iterating, 173–176

 MAC address table, 123–125

 viewing as a table, 123

lists, 40, 44, 99–100, 156–157. *See also* dictionaries

`load_merge_candidate()` method, 105, 179, 182–183

`load_replace_candidate()` method, 105, 180, 193

login local command, 6

M

MAC address table, 123–125

methods

 comparing Netmiko and NAPALM, 86–87

NAPALM

cli(), 105, 188–190, 191–192

commit_config(), 105, 180, 185

compare_config(), 105, 180, 183–185

configuration-related, 185

device.get_facts(), 138–140

discard_config(), 105, 180, 185
 get_arp_table(), 104, 125–127, 134, 173–174

get_bgp_config(), 103, 134

get_bgp_neighbors(), 134

get_bgp_neighbors_detail(), 104, 134

get_config(), 103, 133, 199–200

get_environment(), 103, 112–114, 133, 158

get_facts(), 95–98, 103,
 110–112, 132, 133, 147–148,
 154–157
get_interfaces(), 103, 115–116,
 132–133, 134
get_interfaces_counters(),
 103, 134
get_interfaces_ip(), 86–87, 103,
 120–123, 134, 184
get_ipv6_neighbors_table(),
 104, 135
get_lldp_neighbors(), 103, 134
get_lldp_neighbors_detail(),
 103
get_mac_address_table(), 104,
 122–125, 135
get_network_instances(),
 104, 134
get_ntp_peers(), 104, 134
get_ntp_servers(), 104, 134
get_ntp_stats(), 104, 135
get_optics(), 104, 135
get_route_ro(destination),
 104, 134
get_snmp_information(),
 103, 133
get_users(), 103, 133
get_vlans(), 104, 134
is_alive(), 103, 133
*Layer 2 and neighbor discov-
 ery*, 104
load_merge_candidate(), 105,
 179, 182–183
load_replace_candidate(), 105,
 180, 193
ping(destination), 104, 135
rollback(), 105, 180
traceroute(destination),
 104, 135

Netmiko, 15–16. *See also* Netmiko
disconnect(), 15
enable(), 38
exit_config_mode(), 38
save_config(), 26–28, 186–188
send_command(), 14–15, 17–18,
 20, 35–36, 35–39, 43
send_command_expect(), 23–25
send_config_from_file(), 44–48
send_config_set(), 39–41

Python

.items(), 142–143, 145,
 153–154
.keys(), 140–142, 143–144
.values(), 142, 144

relationship with objects and classes,
 16

MIT License, 2

MultiResult object, 230–HOST

multi-threading, 204

my_first_netmiko_program.py,
 output, 10

N

**NAPALM (Network Automation
 and Programmability Abstraction
 Layer with Multivendor support)**,
 2–3, 4, 85–86, 247–248, 324–325

API abstraction, 91–92

cli() method, 105, 188–190, 192

accessing the key, 192

accessing the value of the key,
 191–192

*storing the output to a
 variable*, 191

commit_config() method, 105,
 180, 185

- compare_config() method, 105, 180, 183–185
- comparing with Nornir, 205
- configuration-related methods, 105
- device.get_facts() method, 138–140
- dictionary of dictionaries, 115–116
 - reasons for using*, 119–120
 - viewing as a table*, 118–119
- discard_config() method, 105, 180, 185
- framework, 88–91
- general device information retrieval methods, 103
- get_arp_table() method, 104, 125–127, 134, 173–174
- get_bgp_config() method, 103, 134
- get_bgp_neighbors() method, 134
- get_bgp_neighbors_detail() method, 104, 134
- get_config() method, 103, 133, 199–200
- get_environment() method, 103, 112–114, 133, 158
- get_facts() method, 95–97, 103, 110–112, 132, 133, 147–148, 154–157
- get_interfaces() method, 103, 115–116, 132–133, 134
- get_interfaces_counters() method, 103, 134
- get_interfaces_ip() method, 86–87, 103, 120, 121–122, 134, 184
- get_ipv6_neighbors_table() method, 104, 135
- get_lldp_neighbors() method, 103, 134
- get_lldp_neighbors_detail() method, 103
- get_mac_address_table() method, 104, 122–125, 135
- get_network_driver() function, 89–90
- get_network_instances() method, 104, 134
- get_ntp_peers() method, 104, 134
- get_ntp_servers() method, 104, 134
- get_ntp_stats() method, 104, 135
- get_optics() method, 104, 135
- get_route_to(destination) method, 104, 134
- get_snmp_information() method, 103, 133
- get_users() method, 103, 133
- get_vlans() method, 104, 134
- installing, 88
- integration with Nornir. *See* Nornir, NAPALM integration
- interface and networking information retrieval methods, 103
- is_alive() method, 103, 133
- Layer 2 and neighbor discovery methods, 104
- list of dictionaries, 122–123
 - ARP table*, 125–127
 - MAC address table*, 123–125
 - viewing as a table*, 123
- load_merge_candidate() method, 105, 179, 182–183
- load_replace_candidate() method, 105, 180, 193
- optical and NTP information retrieval methods, 104
- ping(destination) method, 104, 135
- pprint() function, 90–91, 154, 190–191
- predefined dictionary, 100–102, 138
- retrieving live device data, 138–140

- rollback() method, 105, 180
- routing and network instance retrieval methods, 103
- single dictionary, 110–111, 132–133
 - comparing*, 131–133
 - displaying as a table*, 111–112
 - homogenous*, 116–118
 - nested*, 112–115
 - sample program*, 127–130
 - static structure*, 110
- structured data, 86–87, 92
- supported network operating systems, 87
- traceroute(destination) method, 104, 135
- napalm_cli**, 251–252. *See also* `nornir_napalm_cli.py`
- napalm_configure**, 268–269. *See also* `nornir_napalm_configure.py`
- parameters, 269–271
- workflow, 275–276
- napalm_get**, 261–265
- neighbor discovery, NAPALM methods, 104
- nested dictionaries, 158–162
 - accessing specific values, 166–167
 - NAPALM, 112–115
 - three-level, 162–166
- NetBox plugin**, 306–307, 309–310
 - accessing inventory, 309
 - hosts, groups, and defaults, 308–309
 - installing and configuring, 307–308
- NETCONF**, 3–4, 87, 329–331
- Netmiko**, 1–2, 4, 324–325
 - closing an SSH connection, 15
 - comparing with Nornir, 205
 - ConnectHandler() class, 13, 15, 16, 81–82
 - dictionary unpacking*, 65–68
 - session_log parameter*, 28–33
- disconnect() method, 15
- enable() method, 38
- establishing an SSH connection to a device, 13
- exceptions, troubleshooting SSH connections, 73–77
- executing an IOS command and displaying the output, 13–14
- exit_config_mode() method, 38
- for loops, 79. *See also* for loops
 - configuring IPv6 addresses*, 60–65
 - define the list of device IP addresses*, 59
 - display the IPv4 routing table*, 60
 - if-elif-else statements*, 63–64
 - start the loop to iterate over devices*, 59
- framework, 11–13
- importing the library, 13
- installing, 11
- interactive mode, 30–33
- maintaining multiple SSH connections, 77–81
- methods, 15–16, 86–87
- `my_first_netmiko_program.py`, 10
- print command, 12
- reasons for using, 9–10
- requiring privileged EXEC mode access, 30–33
- save_config() method, 26–28, 186–188
- secret parameter, 28–30
- send_command() method, 14–15, 17–18, 20, 35–36, 43

- for multiple configuration commands*, 39
 - output*, 86
 - retrieving information using*, 36
 - sending configuration commands to a device*, 37–39
- send_command_expect() method, 23–25
- send_config_from_file() method, 44–48
- send_config_set() method, applying configuration commands to a device, 39–41
- sending output to a variable, 19–20
 - including the IOS command*, 20–21
 - including the IOS prompt and command*, 21–23
- storing an IP address in a variable, 53–54
- strip_command parameter, 20–21
- supported network operating systems, 34, 365–366
- troubleshooting, 29
- variables, 13–14, 38
- netmiko_send_command(), 222–224
- netmiko_send_config, 236
 - changes to defaults.yaml, 238–240
 - configuring devices with, 236–238
 - for per-device configuration, 240–243
- netmiko.ConnectHandler command, 13
- network automation, 9. *See also* automation
- network operating systems
 - NAPALM-supported, 87
 - Netmiko-supported, 34, 365–366
- Nornir, 3, 4, 200–201, 203–204, 230-HOST, 248, 324–325
 - AggregatedResult object, 229–230, 234–235
 - comparison with Netmiko and Napalm, 205
 - config.yaml, 210–211, 216
 - inventory section*, 211
 - runner section*, 211
 - core, 234
 - defaults.yaml, 215–216, 296–298
 - framework, 207–208
 - groups.yaml, 213, 216, 289–290
 - access_switches group*, 292
 - cisco_device group*, 290–291
 - core_routers group*, 291–292
 - relationship to hosts.yaml*, 214–215
 - subsections*, 213–214
 - hosts.yaml, 211–212, 213, 216, 288–289
 - credentials*, 287
 - data*, 288
 - groups*, 287
 - host name*, 285
 - hostname*, 285–286
 - platform*, 286
 - port*, 286
 - subsections*, 212–213
 - installing, 207
 - integration with Netmiko and NAPALM, 206
 - inventory check program, 208–209
 - output*, 216–218
 - Python file*, 209–210
 - inventory system, 204, 248, 277, 278–279

- accessing built-in host attributes*, 299
 - accessing custom inventory data*, 299–300
 - accessing inventory*, 298–299
 - Ansible plugin*, 310–312
 - core architecture*, 282
 - DictInventory plugin*, 304–306
 - effective values*, 292–294
 - filtering*, 281–282, 295
 - hosts*, 282–283
 - InfraHub plugin*, 316–319
 - inheritance*, 279–280
 - inspecting a filtered inventory*, 295–296
 - NetBox plugin*, 306–310
 - nornir_table_inventory plugin*, 312–316
 - plugins*, 300–301
 - SimpleInventory plugin*, 302–304
 - sources*, 280–281
 - task logic*, 296
 - MultiResult object, 230–HOST
 - NAPALM integration, 248–249, 277–278
 - `netmiko_send_command()`, 222–224.
See also `nornir_netmiko_send_command.py`
 - `nornir_napalm` plugin, installing, 249–250
 - `nornir_netmiko` plugin, 221–222, 234
 - parallel execution, 204, 224–225, 246
 - plugins, 204, 206, 248. *See also* `nornir_netmiko`
 - Result object, 230, 231
 - runners, 248
 - tasks, 248, 250–251
 - execution*, 233–236
 - napalm_cli*, 251–252
 - napalm_configure*, 268–276
 - napalm_get*, 261–265
 - workers, 226
 - YAML files, 207–208
 - `nornir_napalm_cli.py`**, 252–253, 257–259
 - `config.yaml`, 254
 - `defaults.yaml`, 255
 - `groups.yaml`, 254
 - `hosts.yaml`, 254
 - line-by-line breakdown, 259–261
 - output, 255–257
 - `nornir_napalm_configure.py`**, 271–272
 - output, 274–275
 - step-by-step explanation, 272–274
 - `nornir_napalm_get.py`**, 264
 - output, 264–265
 - step-by-step explanation, 266–268
 - `nornir_netmiko_send_command.py`**, 223, 226–228
 - `config.yaml`, 223
 - `defaults.yaml`, 224
 - displaying the results, 230–233
 - `groups.yaml`, 224
 - `hosts.yaml`, 224
 - output, 225, 228–229
 - `nornir_netmiko_send_config_host.py`**, 242, 245–246
-
- ## O
- objects, Python, 16, 355–356
 - OOP (object-oriented programming), 89–90
 - orchestration, 204–205

os.getcwd function, 49
 os.path module, 48–50
output
 get_interfaces_ip() method, 121–122
 inventory check program, 216–218
 my_first_netmiko_program.py, 10
 nornir_napalm_cli.py, 255–257
 nornir_napalm_configure.py, 274–275
 nornir_napalm_get.py, 264–265
 nornir_netmiko_send_command.py, 225
 print() function, 90
 send_command() method, 86
 sending to a variable, 19–20
 including the IOS command,
 20–21
 including the IOS prompt and
 command, 21–23
 send_config_set(), 41–44
 tabular, 359–360

P

parallel execution, 204, 224–225, 246
parameters
 expect_string, 345–346
 napalm_configure, 269–271
 secret, 28–30
 session_log, 28–33
 strip_command, 20–21
 user-defined function, 244
Paramiko, 10
passwords, hiding, 71–73
permissive software license, 2
ping(destination) method, 104, 135
pip3 command, 11
plugins, 204, 300–301
 Inventory, 300–301
 Ansible, 310–312
 built-in versus third-party, 301
 DictInventory, 304–306
 InfraHub, 316–319
 NetBox, 306–310
 nornir_table_inventory,
 312–316
 options, 301
 SimpleInventory, 302–304
 Nornir, 206, 248
pprint() function, 90–91, 154, 190–191
predefined dictionaries, 100–102, 138
print command, 12
print() function, 90, 153, 200
private keys, 361
public keys, 361, 363–364
Python
 dictionaries, 2–3, 4. *See also*
 dictionaries
 curly braces ({}), 94
 format, 94–95
 keys, 98
 key/value pairs, 92–93
 square brackets ({}), 97–99
 values, 99–100
 F-strings, 56
 getpass() function, 71–73
 interactive mode, 30–33, 105–107
 isinstance() function, 151–153
 .items() method, 142–143, 145, 153–154
 .keys() method, 140–142, 143–144

libraries, 10. *See also* NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support); Netmiko; Nornir

for loops, 54–56, 79. *See also* for loops

configuring IPv6 addresses, 60–65

if-elif-else statements, 63–64

iterating through lists, 99–100

iterating through multiple devices, 56–60

lists, 99

objects, 16, 355–356

os.getcwd function, 49

os.path module, 48–50

print() function, 90, 153, 200

try-except structure, 75

tuple, 140

type() function, 148–151

user-defined functions, 243–245

argument, 244

parameter, 244

.values() method, 142, 144

variables, 355–356

virtual environments, 341–343

python3 command, 11

Q-R

recursive function, 357–358

replacing a device configuration, 193

RESTCONF, 3–4, 87, 328–329

Result object, 230, 231

rollback() method, 105, 180

runners, Nornir, 248

running configuration file, displaying, 199–200

S

save_config() method, 26–28, 186–188

saving device configuration changes, 186–188

SCP (Secure Copy Protocol), 181

SDN (software-defined networking), 323

secret parameter, 28–30

send_command() method, 14–15, 17–18, 20–21, 35–36, 43

expect_string parameter, 345–346

for multiple configuration commands, 39

output, 86

retrieving information using, 36

sending configuration commands to a device, 37–39

send_command_expect() method, 23–25

send_config_from_file() method, 44–48

send_config_set() method

applying configuration commands to a device, 39–41

configuring and displaying commands, 41–44

session log, Netmiko, 28–33

show ip interface brief command, 10, 12, 71

SimpleInventory plugin, 302–304

spreadsheet plugins, 312–314

installing and configuring, 314–315

when to use, 315–316

square brackets ([]), 97–99

SSH

- closing a connection, 15
- configuring on Cisco IOS, 5–6
- connecting to a device using Netmiko, 13
- connections
 - maintaining multiple*, 77–81
 - troubleshooting using Netmiko exceptions*, 73–77
- hiding credentials, 71–73
- keys, 362–363
- startup configuration file, displaying, 199–200
- static structure, NAPALM dictionaries, 110
- storing device dictionaries, 69–70
- strip_command parameter, 20–21
- structured data, 2, 3–4, 86–87, 92. *See also* dictionaries
- supported network operating systems, Netmiko, 34

T

- tabular output, 359–360
- task execution, Nornir, 233–236
- task logic, 296
- tasks, Nornir, 248, 250–251
 - napalm_cli, 251–252
 - napalm_configure, 268–269
 - parameters*, 269–271
 - workflow*, 275–276
 - napalm_get, 261–265
- Terraform, 204
- testing connectivity, NAPALM methods, 104
- third-party plugins, 301

- three-level nested dictionaries, iterating, 162–166
- traceroute(destination) method, 104, 135
- transport input ssh command, 6
- troubleshooting, 71
 - connectivity, 73–77
 - Netmiko, 29
- try-except structure, Python, 75
- tuple, 140
- type() function, 148–151

U

- unified API, 85–86
- user-defined functions, 243–245
 - argument, 244
 - parameter, 244
- username admin password cisco command, 5–6
- username admin privilege 15 password 0 cisco command, 181

V

- values. *See also* dictionaries; key/value pairs
 - checking
 - isinstance() function*, 151–153
 - type() function*, 148–151
 - determining the type of, 147–148
 - dictionary, 99–100, 158–162
 - effective, 292–294
 - iterating through, 144
 - lists, iterating through, 154–158
 - tuple, 140
- .values() method, 142, 144

variable/s. *See also* dictionaries

assigning to a dictionary, 97–98

connection, 38

device_facts, 102

Netmiko command, 13–14

Python, 355–356

storing an IP address, 53–54

storing output in a, 19–20, 191

*including the IOS command,
20–21*

*including the IOS prompt and
command, 21–23*

send_config_set(), 41–44

vendor-agnostic commands, 2

virtual environments, Python,
341–343

W-X-Y-Z

workers, 226

YAML files, 204, 207–208. *See also*
config.yaml; defaults.yaml; groups.
yaml; hosts.yaml

YANG models, 332–334