Network Programmability and Automation Fundamentals

KHALED ABUELENAIN, CCIE® No. 27401
JEFF DOYLE, CCIE® No. 1919
ANTON KARNELIUK, CCIE® No. 49412
VINIT JAIN, CCIE® No. 22854
Network Programmability and Automation Fundamentals

Khaled Abuelenain, CCIE No. 27401
Jeff Doyle, CCIE No. 1919
Anton Karneliuk, CCIE No. 49412
Vinit Jain, CCIE No. 22854

Cisco Press
Hoboken, New Jersey
Network Programmability and Automation Fundamentals

Copyright© 2021 Cisco Systems, Inc.

Cisco Press logo is a trademark of Cisco Systems, Inc.

Published by:
Cisco Press

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 www.pearson.com/permissions.

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

Library of Congress Control Number: 2020922839

Warning and Disclaimer

This book is designed to provide information about network programmability and automation. 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.

Editor-in-Chief: Mark Taub
Technical Editors: Jeff Tantsura, Viktor Osipchuk
Director, ITP Product Management: Brett Bartow
Editorial Assistant: Cindy Teeters
Alliances Manager, Cisco Press: Arezou Gol
Designer: Chuti Prasertsith
Managing Editor: Sandra Schroeder
Composition: codeMantra
Development Editor: Ellie C. Bru
Indexer: Ken Johnson
Project Editor: Mandie Frank
Proofreader: Abigail Bass
Copy Editor: Kitty Wilson

Credits

Figure/Text Selection

“HTTP is not designed to be a transport protocol. It is a transfer protocol in which the messages reflect the semantics of the Web architecture by performing actions on resources through the transfer and manipulation of representations of those resources [Section 6.5.2]”

“This specification [HTTP/2.0] is an alternative to, but does not obsolete, the HTTP/1.1 message syntax. HTTP’s existing semantics remain unchanged.”

“a sequence of octets, along with representation metadata describing those octets that constitutes a record of the state of the resource at the time when the representation is generated.”

“While RFC 2396, section 1.2, attempts to address the distinction between URIs, URLs and URNs, it has not been successful in clearing up the confusion.”

1. Device state metrics;
2. Data from shared services such as DDI (DNS, DHCP and IPAM) and Active Directory;
3. Network flows from sources such as NetFlow; and
4. Configuration data normalized into key value pairs.

“Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale”.

“allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.”

“the most important security protocol on the internet”

“there was more TLS 1.3 use in the first five months after RFC 8446 was published than in the first five years after the last version of TLS was published as an RFC”

Attribution/Credit

© Roy Thomas Fielding, 2000

Hypertext Transfer Protocol Version 2

Uniform Resource Identifier (URI): Generic Syntax, Copyright © The Internet Society (2005)


©2020 Agile Alliance

Copyright (c) 2020 IETF

Copyright (c) 2020 IETF

Copyright (c) 2020 IETF
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Copyright/Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>CI/CD</td>
<td>©2021 Red Hat, Inc. <a href="http://www.redhat.com">www.redhat.com</a></td>
</tr>
<tr>
<td>6-2</td>
<td>Accessing a Django Application in a Web Browser</td>
<td>© 2005-2021 Django Software Foundation</td>
</tr>
<tr>
<td>7-1</td>
<td>The SupportApache-small.png Image, as It Appears in a Web Browser</td>
<td>Copyright © 2020 The Apache Software Foundation</td>
</tr>
<tr>
<td>7-6</td>
<td>The Postman Interface</td>
<td>©2020 Postman, Inc.</td>
</tr>
<tr>
<td>7-7</td>
<td>A GET Request Using Postman</td>
<td>©2020 Postman, Inc.</td>
</tr>
<tr>
<td>7-8</td>
<td>Viewing the Response Headers in Postman</td>
<td>©2020 Postman, Inc.</td>
</tr>
<tr>
<td>13-1</td>
<td>GitHub Website with YANG Modules</td>
<td>© 2020 GitHub, Inc.</td>
</tr>
<tr>
<td>18-3</td>
<td>Converting a username:password Tuple to Base64 Format</td>
<td>Screenshot of Converting username:password tuple to Base64 format © Cisco systems</td>
</tr>
<tr>
<td>18-4</td>
<td>Arista YANG Modules</td>
<td>Screenshot of Arista YANG modules © 2020 GitHub, Inc.</td>
</tr>
<tr>
<td>19-3</td>
<td>Available Ansible Module Categories</td>
<td>Copyright © 2020 Red Hat, Inc.</td>
</tr>
</tbody>
</table>
About the Authors

Khaled Abuelenain, CCIE No. 27401 (R&S, SP), is currently the Consulting Director at Acuative, a Cisco Managed Services Master Partner. Khaled has spent the past 18 years designing, implementing, operating, and automating networks and clouds. He specializes in service provider technologies, SD-WAN, data center technologies, programmability, automation, and cloud architectures. Khaled is especially interested in Linux and OpenStack.

Khaled is a contributing author of the best-selling Cisco Press book Routing TCP/IP, Volume II, 2nd edition, by Jeff Doyle. He also blogs frequently on network programmability and automation on blogs.cisco.com. Khaled is also a member of the DevNet500 group, being one of the first 500 individuals in the world to become DevNet certified.

Khaled lives in Riyadh, Saudi Arabia, and when not working or writing, he likes to run marathons and skydive. He can be reached at kabuelenain@gmail.com, on Twitter at @kabuelenain or on LinkedIn at linkedin.com/in/kabuelenain.

Jeff Doyle, CCIE No. 1919, is a Member of Technical Staff at Apstra. Specializing in IP routing protocols, complex BGP policy, SDN/NFV, data center fabrics, IBN, EVPN, MPLS, and IPv6, Jeff has designed or assisted in the design of large-scale IP and IPv6 service provider networks in 26 countries over 6 continents.

Jeff is the author of CCIE Professional Development: Routing TCP/IP, Volumes I and II and OSPF and IS-IS: Choosing an IGP for Large-Scale Networks; a co-author of Software-Defined Networking: Anatomy of OpenFlow; and an editor and contributing author of Juniper Networks Routers: The Complete Reference. Jeff is currently writing CCIE Professional Development: Switching TCP/IP. He also writes for Forbes and blogs for both Network World and Network Computing. Jeff is one of the founders of the Rocky Mountain IPv6 Task Force, is an IPv6 Forum Fellow, and serves on the executive board of the Colorado chapter of the Internet Society (ISOC).

Anton Karneliuk, CCIE No. 49412 (R&S, SP), is a Network Engineer and Manager at THG Hosting, responsible for the development, operation, and automation of networks in numerous data centers across the globe and the international backbone. Prior to joining THG, Anton was a team lead in Vodafone Group Network Engineering and Delivery, focusing on introduction of SDN and NFV projects in Germany. Anton has 15 years of extensive experience in design, rollout, operation, and optimization of large-scale service providers and converged networks, focusing on IP/MPLS, BGP, network security, and data center Clos fabrics built using EVPN/VXLAN. He also has several years of full-stack software development experience for network management and automation.

Anton holds a B.S. in telecommunications and an M.S. in information security from Belarusian State University of Informatics and Radio Electronics. You can find him actively blogging about network automation and running online training at Karneliuk.com. Anton lives with his wife in London.
Vinit Jain, CCIE No. 22854 (R&S, SP, Security & DC), is a Network Development Engineer at Amazon, managing the Amazon network backbone operations team. Previously, he worked as a technical leader with the Cisco Technical Assistance Center (TAC), providing escalation support in routing and data center technologies. Vinit is a speaker at various networking forums, including Cisco Live! events. He has co-authored several Cisco Press titles, such as Troubleshooting BGP, and Troubleshooting Cisco Nexus Switches and NX-OS, LISP Network Deployment and Troubleshooting, and has authored and co-authored several video courses, including BGP Troubleshooting, the CCNP DCCOR Complete video course, and the CCNP ENCOR Complete video course. In addition to his CCIEs, Vinit holds multiple certifications related to programming and databases. Vinit graduated from Delhi University in mathematics and earned a master's in information technology from Kuvempu University in India. Vinit can be found on Twitter as @VinuGenie.
About the Technical Reviewers

Jeff Tantsura, CCIE No. 11416 (R&S), has been in the networking space for over 25 years and has authored and contributed to many RFCs and patents and worked in both service provider and vendor environments.

He is co-chair of IETF Routing Working Group, chartered to work on new network architectures and technologies, including protocol-independent YANG models and next-generation routing protocols. He is also the co-chair of the RIFT (Routing in Fat Trees) Working Group, chartered to work on a new routing protocol that specifically addresses fat tree topologies typically seen in the data center environment.

Jeff serves on the Internet Architecture Board (IAB). His focus has been on 5G transport and integration with RAN, IoT, MEC, low-latency networking, and data modeling. He’s also a board member of San Francisco Bay Area ISOC chapter.

Jeff is Head of Networking Strategy at Apstra, a leader in intent networking, where he defines networking strategy and technologies.

Jeff also holds the certification Ericsson Certified Expert IP Networking.

Jeff lives in Palo Alto, California, with his wife and youngest child.

Viktor Osipchuk, CCIE No. 38256 (R&S, SP), is a Senior Network Engineer at Google, focusing on automation and improving one of the largest production networks in the world. Before joining Google, Viktor spent time at DigitalOcean and Equinix, helping to architect and run their worldwide infrastructures. Viktor spent many years at Cisco, supporting customers and focusing on automation, telemetry, data models, and APIs for large-scale web and service provider deployments. Viktor has around 15 years of diverse network experience, an M.S. in telecommunications, and associated industry certifications.
Dedications

Khaled Abuelenain: To my mother, the dearest person to my heart, who invested all the years of her life so I can be who I am today. I owe you more than any words can express. To my father, my role model, who always led by example and showed me the real meaning of work ethic. Nothing I do or say will ever be enough to thank you both.

And to the love of my life, my soulmate, and my better half, Mai, for letting me work and write while you take care of, literally, everything else. This book would not have happened if not for your phenomenal support, patience and love. I will forever be grateful for the blessing of having you in my life.

Jeff Doyle: I would like to dedicate this book to my large and growing herd of grandchildren: Claire, Samuel, Caroline, Elsie, and Amelia. While they are far too young to comprehend or care about the contents of this book, perhaps someday they will look at it and appreciate that Grampa is more than a nice old man and itinerant babysitter.

Anton Karneliuk: I dedicate this book to my family, which has tremendously supported me during the writing process. First of all, many thanks to my amazing wife, Julia, who took on the huge burden of sorting out many things for our lives, allowing me to concentrate on the book. You acted as a navigation star during this journey, and you are my beauty. I’d also like to thank my parents and brother for me helping me form the habit of working hard and completing the tasks I’ve committed to, no matter how badly I want to drop them.

Vinit Jain: I would like to dedicate this book to the woman who has been a great influence and inspiration in my life: Sonal Sethia (Sonpari). You are one of the most brilliant, talented, courageous, and humble people I have ever known. You have always inspired me to push myself beyond what I thought I was capable of. You have been there for me during difficult times and believed in me when even I did not. You are my rock. This is a small token of my appreciation, gratitude, and love for you. I am really glad to have found my best friend in you and know that I will always be there for you.
Acknowledgments

Khaled: First and foremost, I would like to thank Jeff Doyle, my co-author, mentor, and friend, for getting me started with writing, and for his continuous assistance and guidance. Jeff has played a fundamental role in my professional life as well as in the lives of many other network engineers; he probably doesn't realize the magnitude of this role! Despite all that he has done for this industry and the network engineering community, Jeff remains one of the most humble and amiable human beings I have ever come across. Thank you, Jeff, I owe you a lot!

I am grateful to Anton and Vinit for agreeing to work with me on this project. It has been challenging at times, but it has been seriously fun most of the time.

I would also like to thank Jeff Tantsura and Viktor Osipchuk for their thorough technical reviews and feedback. I bothered Viktor very frequently with discussions and questions over email, and never once did he fail to reply and add a ton of value!

I especially want to thank Brett Bartow and Eleanor Bru for their immense support and phenomenal patience. And I’m grateful to Mandie Frank, Kitty Wilson, and everyone else at Cisco Press who worked hard to get this book out to the light. Such an amazing team.

Jeff Doyle: I would like to express my thanks to my friend Khaled Abuelenain for bringing me into this project, and thanks to Anton and Vinit for letting me be a part of their excellent work. Thanks also to Brett Bartow and everyone at Pearson, whom I’ve worked with for many years and continue to tell everyone who will listen that this is the best publishing team any technical writer could hope to work for. Finally, thanks to my wife Sara who, as always, puts up with my obsessiveness. When she sees me sitting and staring into nothingness she knows there’s writing going on in my head.

Anton: Special thanks to Schalk Van Der Merwe, CTO, and Andrew Mutty, CIO, at The Hut Group for believing in me and giving me freedom and responsibility to implement my automation ideas in a high-scale data center environment. Thanks to all my brothers-in-arms from The Hut Group hosting networks for constantly sharing with me ideas about what use cases to focus on for automation. I want to thank my previous manager in Vodafone Group, Tamas Almasi, who supported me during my initial steps in network automation and helped me create an appropriate mindset during numerous testbeds and proofs of concept. Last but not least, I’m very grateful to Khaled Abuelenain for his invitation to co-author this book and the whole author and technical reviewer team; it was a pleasure to work with you.

Vinit: A special thanks to Khaled for asking me to co-author this book and for being amazingly patient and supportive of me as I faced challenges during this project. I would like to thank Jeff Doyle and Anton Karneliuk for their amazing collaboration on this project. I learned a lot from all of you guys and look forward to working with all of you in the future.

I would also like to thank our technical reviewers, Jeff Tantsura and Viktor Osipchuk, and our editor, Eleanor Bru, for your in-depth verification of the content and insightful input to make this project a successful one.

This project wouldn’t have been possible without the support of Brett Bartow and other members of the editorial team.
Contents at a Glance

Introduction xxix

Part I Introduction
Chapter 1 The Network Programmability and Automation Ecosystem 1

Part II Linux
Chapter 2 Linux Fundamentals 21
Chapter 3 Linux Storage, Security, and Networks 119
Chapter 4 Linux Scripting 183

Part III Python
Chapter 5 Python Fundamentals 249
Chapter 6 Python Applications 311

Part IV Transport
Chapter 7 HTTP and REST 387
Chapter 8 Advanced HTTP 469
Chapter 9 SSH 509

Part V Encoding
Chapter 10 XML 553
Chapter 11 JSON 591
Chapter 12 YAML 615

Part VI Modeling
Chapter 13 YANG 639

Part VII Protocols
Chapter 14 NETCONF and RESTCONF 689
Chapter 15 gRPC, Protobuf, and gNMI 781
Chapter 16 Service Provider Programmability 819
xii  Network Programmability and Automation Fundamentals

**Part VIII  Programmability Applications**
Chapter 17  Programming Cisco Platforms  881
Chapter 18  Programming Non-Cisco Platforms  957
Chapter 19  Ansible  989

**Part IX  Looking Ahead**
Chapter 20  Looking Ahead  1109

Index  1121
# Contents

Introduction   xxix

## Part I  Introduction

### Chapter 1  The Network Programmability and Automation Ecosystem  1

First, a Few Definitions  2
- Network Management  3
- Automation  5
- Orchestration  6
- Programmability  7
- Virtualization and Abstraction  8
- Software-Defined Networking  13
- Intent-Based Networking  13

Your Network Programmability and Automation Toolbox  14
- Python  15
- Ansible  15
- Linux  16
- Virtualization  17
- YANG  17
- Protocols  18
- Encoding the Protocols  18
- Transporting the Protocols  18

Software and Network Engineers: The New Era  19

## Part II  Linux

### Chapter 2  Linux Fundamentals  21

The Story of Linux  21
- History  21
- Linux Today  22
- Linux Development  22
- Linux Architecture  23
- Linux Distributions  26

The Linux Boot Process  26
- A Linux Command Shell Primer  28

Finding Help in Linux  31
<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Linux Scripting</th>
<th>183</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Regular Expressions and the <code>grep</code> Utility</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>The AWK Programming Language</td>
<td>193</td>
</tr>
<tr>
<td></td>
<td>The <code>sed</code> Utility</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>General Structure of Shell Scripts</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>Output and Input</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>Input</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>Variables</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>Integers and Strings</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>Indexed and Associative Arrays</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Conditional Statements</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>The if-then Construct</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>The case-in Construct</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>Loops</td>
<td>232</td>
</tr>
<tr>
<td></td>
<td>The for-do Loop</td>
<td>232</td>
</tr>
<tr>
<td></td>
<td>The while-do Loop</td>
<td>236</td>
</tr>
<tr>
<td></td>
<td>The until-do Loop</td>
<td>237</td>
</tr>
<tr>
<td></td>
<td>Functions</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td>Expect</td>
<td>242</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>246</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part III</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 5</td>
<td>Python Fundamentals</td>
</tr>
<tr>
<td></td>
<td>Scripting Languages Versus Programming Languages</td>
</tr>
<tr>
<td></td>
<td>Network Programmability</td>
</tr>
<tr>
<td></td>
<td>Computer Science Concepts</td>
</tr>
<tr>
<td></td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td></td>
<td>Algorithms</td>
</tr>
<tr>
<td></td>
<td>Python Fundamentals</td>
</tr>
<tr>
<td></td>
<td>Python Installation</td>
</tr>
<tr>
<td></td>
<td>Python Code Execution</td>
</tr>
<tr>
<td></td>
<td>Python Data Types</td>
</tr>
</tbody>
</table>
Variables 270
Numbers 273
Strings 276
Operators 281
Python Data Structures 286
List 286
Dictionaries 290
Tuples 292
Sets 294
Control Flow 295
if-else Statements 296
for Loops 301
while Loops 304
Functions 306
Summary 309
References 310

Chapter 6 Python Applications 311
Organizing the Development Environment 311
Git 312
Docker 317
The virtualenv Tool 331
Python Modules 333
Python Applications 336
Web/API Development 336
Django 337
Flask 345
Network Automation 353
NAPALM 354
Nornir 359
Templating with Jinja2 363
Orchestration 375
Docker 376
Kubernetes 378
Machine Learning 382
Summary 385
Part IV    Transport

Chapter 7    HTTP and REST    387

HTTP Overview    387
The REST Framework    392
The HTTP Connection    394
  Client/Server Communication    394
HTTP/1.1 Connection Enhancements    395
    Persistent Connections    395
    Pipelining    396
    Compression    396
HTTP Transactions    397
  Client Requests    397
    GET    398
    HEAD    398
    POST    399
    PUT    402
    DELETE    405
    CONNECT    407
    OPTIONS    407
    TRACE    408
  Server Status Codes    408
    1xx: Informational Status Codes    411
    2xx: Successful Status Codes    411
    3xx: Redirection Status Codes    412
    4xx: Client Error Status Codes    413
    5xx: Server Error Status Codes    414
  Server Status Codes on Cisco Devices    414
HTTP Messages    415
  HTTP General Header Fields    418
  Cache Servers: Cache-Control and Pragma    418
    Connection    420
    Date    420
    Upgrade    420
    Via    421
    Transfer-Encoding    421
    Trailer    422
  Client Request Header Fields    422
Content Negotiation Header Fields: Accept, Accept-Charset, Accept-Encoding and Accept-Language 423
Client Authentication Credentials: Authorization, Proxy-Authorization and Cookie 423
Host 424
Expect 424
Max-Fowards 424
Request Context: From, Referer and User-Agent 424
TE 425
Server Response Header Fields 425
Age 425
Validator Header Fields: ETag and Last-Modified 425
Response Authentication Challenges: X-Authenticate and Set-Cookie 426
Response Control Header Fields: Location, Retry-After, and Vary 426
Response Context: Server 427
The HTTP Entity Header Fields 427
Control Header Fields: Allow 428
Representation Metadata Header Fields: Content-X 428
Content-Length 430
Expires 430
Resource Identification 431
URI, URL, and URN 431
URI Syntax 432
URI Components 432
Characters 435
Absolute and Relative References 436
Postman 436
Downloading and Installing Postman 438
The Postman Interface 438
Using Postman 441
HTTP and Bash 447
HTTP and Python 455
TCP Over Python: The socket Module 455
The urllib Package 458
The requests Package 464
Summary 467
## Chapter 8  Advanced HTTP  469

HTTP/1.1 Authentication  469
  Basic Authentication  472
  OAuth and Bearer Tokens  474
  Client Registration  476
  Authorization Grant  477
  Access Token  481
  API Call to the Resource Server  483
  State Management Using Cookies  483
Transport Layer Security (TLS) and HTTPS  487
  Cryptography Primer  488
  Key Generation and Exchange  488
  Stream and Block Data Encryption  492
  Message Integrity and Authenticity  493
  Encryption and Message Integrity and Authenticity Combined  495
  Digital Signatures and Peer Authentication  496
  TLS 1.3 Protocol Operation  498
  The TLS Version 1.3 Handshake  500
  0-RTT and Early Data  502
  The Record Protocol  503
  HTTP over TLS (HTTPS)  503
HTTP/2  503
  Streams, Messages, and Frames  504
  Frame Multiplexing  505
  Binary Message Framing  506
  Other HTTP/2 Optimizations  507
Summary  508

## Chapter 9  SSH  509

SSH Overview  509
  SSH1  510
  SSH2  512
  SSH Transport Layer Protocol  513
  SSH Authentication Protocol  514
  SSH Connection Protocol  518
Setting Up SSH 521
  Setting Up SSH on CentOS 521
Enabling SSH on Cisco Devices 526
  Configuring and Verifying SSH on Cisco IOS XE 526
  Configuring SSH on IOS XR 532
  Configuring SSH on NX-OS 537
Secure File Transfer 540
  Setting Up SFTP on Cisco Devices 545
Secure Copy Protocol 549
Summary 551
References 551

Part V  Encoding

Chapter 10  XML 553
  XML Overview, History, and Usage 553
  XML Syntax and Components 554
    XML Document Building Blocks 554
    XML Attributes, Comments, and Namespaces 558
    XML Formatting Rules 561
  Making XML Valid 562
    XML DTD 563
    XSD 565
    Brief Comparison of XSD and DTD 574
  Navigating XML Documents 574
    XPath 574
  XML Stylesheet Language Transformations (XSLT) 578
  Processing XML Files with Python 580
Summary 588

Chapter 11  JSON 591
  JavaScript Object Notation (JSON) 591
    JSON Data Format and Data Types 592
  JSON Schema Definition (JSD) 595
    Structure of the JSON Schema 595
    Repetitive Objects in the JSON Schema 598
    Referencing External JSON Schemas 602
    Using JSON Schemas for Data Validation 609
Summary 614
Chapter 12  YAML  615
YAML Structure  616
  Collections  618
  Scalars  620
  Tags  621
  Anchors  624
  YAML Example  625
Handling YAML Data Using Python  626
Summary  637

Part VI  Modeling

Chapter 13  YANG  639
A Data Modeling Primer  639
  What Is a Data Model?  639
  Why Data Modeling Matters  640
YANG Data Models  642
  Structure of a YANG Module  644
  Data Types in a YANG Module  646
  Built-in Data Types  647
  Derived Data Types  648
  Data Modeling Nodes  649
  Leaf Nodes  649
  Leaf-List Nodes  651
  Container Nodes  652
  List Nodes  653
  Grouping Nodes  654
  Aug augmentations in YANG Modules  656
  Deviations in YANG Modules  658
YANG 1.1  662
Types of YANG Modules  663
  The Home of YANG Modules  664
  Native (Vendor-Specific) YANG Modules  666
  IETF YANG Modules  670
  OpenConfig YANG Modules  671
YANG Tools  673
  Using pyang  673
Part VII  Protocols

Chapter 14  NETCONF and RESTCONF  689

NETCONF  689
  NETCONF Overview  689
  NETCONF Architecture  692
  The NETCONF Transport Layer  693
  NETCONF Transport Protocol Requirements  693
  NETCONF over SSH  694
  The NETCONF Messages Layer  695
  Hello Messages  696
  rpc Messages  698
  rpc-reply Messages  699
  The NETCONF Operations Layer  701
  Retrieving Data: <get> and <get-config>  702
  Changing Configuration: <edit-config>, <copy-config>, and
  <delete-config>  712
  Datastore Operations: <lock> and <unlock>  720
  Session Operations: <close-session> and <kill-session>  721
  Candidate Configuration Operations: <commit>, <discard-changes>,
  and <cancel-commit>  722
  Configuration Validation: <validate>  724
  The NETCONF Content Layer  725
  NETCONF Capabilities  731
  The Writable Running Capability  732
  The Candidate Configuration Capability  732
  The Confirmed Commit Capability  732
  The Rollback-on-Error Capability  732
  The Validate Capability  733
  The Distinct Startup Capability  733
  The URL Capability  733
  The XPath Capability  735
  NETCONF Using Python: ncclient  735

RESTCONF  739
  Protocol Overview  739
Protocol Architecture 742
The RESTCONF Transport Layer 743
The RESTCONF Messages Layer 743
Request Messages 743
Response Messages 744
Constructing RESTCONF Messages 745
RESTCONF HTTP Headers 745
RESTCONF Error Reporting 746
Resources 746
The API Resource 747
The Datastore Resource 749
The Schema Resource 750
The Data Resource 753
The Operations Resource 756
The YANG Library Version Resource 758
Methods and the RESTCONF Operations Layer 759
Retrieving Data: OPTIONS, GET, and HEAD 759
Editing Data: POST, PUT, PATCH, and DELETE 763
Query Parameters 771
RESTCONF and Python 777
Summary 779

Chapter 15  gRPC, Protobuf, and gNMI 781
Requirements for Efficient Transport 781
History and Principles of gRPC 782
gRPC as a Transport 784
The Protocol Buffers Data Format 786
Working with gRPC and Protobuf in Python 790
The gNMI Specification 798
The Anatomy of gNMI 799
The Get RPC 801
The Set RPC 807
The Capabilities RPC 810
The Subscribe RPC 811
Managing Network Elements with gNMI/gRPC 814
Summary 818
Chapter 16  Service Provider Programmability  819
The SDN Framework for Service Providers  819
  Requirements for Service Provider Networks of the Future  819
  SDN Controllers for Service Provider Networks  821
Segment Routing (SR)  823
  Segment Routing Basics  823
  Segment Routing Traffic Engineering  832
BGP Link State (BGP-LS)  843
  BGP-LS Basics  843
  BGP-LS Route Types  850
  Node NLRI  854
  Link NLRI  856
  Prefix NLRI  858
Path Computation Element Protocol (PCEP)  859
  Typical PCEP Call Flow  861
  PCEP Call Flow with Delegation  865
  Configuring PCEP in Cisco IOS XR  867
Summary  880

Part VIII  Programmability Applications

Chapter 17  Programming Cisco Platforms  881
API Classification  882
Network Platforms  883
  Networking APIs  884
  Open NX-OS Programmability  884
  IOS XE Programmability  885
  IOS XR Programmability  886
Use Cases  887
  Use Case 1: Linux Shells  887
  Use Case 2: NX-API CLI  893
  Use Case 3: NX-API REST  898
  Use Case 4: NETCONF  905
Meraki  922
  Meraki APIs  922
  Meraki Use Case: Dashboard API  923
DNA Center  931
  DNA Center APIs  933
  Intent API  934
Device Management 934
Event Notifications and Webhooks 935
Integration API 935
Use Case: Intent API 936

Collaboration Platforms 942
Cisco's Collaboration Portfolio 942
Collaboration APIs 944
Cisco Unified Communications Manager (CUCM) 944
Webex Meetings 945
Webex Teams 945
Webex Devices 946
Finesse 946
Use Case: Webex Teams 948

Chapter 18 Programming Non-Cisco Platforms 957
General Approaches to Programming Networks 957
The Vendor/API Matrix 957
Programmability via the CLI 958
Programmability via SNMP 959
Programmability via the Linux Shell 960
Programmability via NETCONF 960
Programmability via RESTCONF and REST APIs 961
Programmability via gRPC/gNMI 961
Implementation Examples 962
Converting the Traditional CLI to a Programmable One 962
Classical Linux-Based Programmability 967
Managing Network Devices with NETCONF/YANG 973
Managing Network Devices with RESTCONF/YANG 978
Summary 987

Chapter 19 Ansible 989
Ansible Basics 989
How Ansible Works 990
Ad Hoc Commands and Playbooks 996
The World of Ansible Modules 1000
Extending Ansible Capabilities 1003
Connection Plugins 1003
Variables and Facts 1005
Filters 1013
Conditionals 1016
Loops 1024
Jinja2 Templates 1034
The Need for Templates 1034
Variables, Loops, and Conditions 1040
Using Python Functions in Jinja2 1049
The join() Function 1050
The split() Function 1051
The map() Function 1054
Using Ansible for Cisco IOS XE 1055
Operational Data Verification Using the ios_command Module 1058
General Configuration Using the ios_config Module 1061
Configuration Using Various ios_* Modules 1069
Using Ansible for Cisco IOS XR 1073
Operational Data Verification Using the iosxr_command Module 1075
General Configuration Using the iosxr_config Module 1078
Configuration Using Various iosxr_* Modules 1083
Using Ansible for Cisco NX-OS 1084
Operational Data Verification Using the nxos_command Module 1086
General Configuration Using the nxos_config Module 1089
Configuration Using Various nxos_* Modules 1093
Using Ansible in Conjunction with NETCONF 1095
Operational Data Verification Using the netconf_get Module 1098
General Configuration Using the netconf_config Module 1103
Summary 1108

Part IX  Looking Ahead

Chapter 20  Looking Ahead  1109

Some Rules of Thumb 1109
Automate the Painful Stuff 1109
Don’t Automate a Broken Process 1110
Clean Up Your Network 1110
Find Your Sources of Truth 1110
Avoid Automation You Can’t Reuse 1111
Document What You Do  1111
Understand What Level of Complexity You're Willing to Handle  1111
Do a Cost/Benefit Analysis  1112
What Do You Study Next?  1112
  Model-Driven Telemetry  1113
  Containers: Docker and Kubernetes  1114
  Application Hosting  1115
  Software Development Methodologies  1116
  Miscellaneous Topics  1117
What Does All This Mean for Your Career?  1118

Index  1121
Icons Used in This Book

- Laptop
- Cisco Carrier Routing System
- Mobile Customer
- PC with software
- Router
- Database
- Cloud
- Wireless Connectivity
- Wireless Modem/Wireless Gateway
- Server
- Cisco Nexus 7000
- File Server

Command Syntax Conventions

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

- **Boldface** indicates commands and keywords that are entered literally as shown. In actual configuration examples and output (not general command syntax), boldface indicates commands that are manually input by the user (such as a `show` command).

- **Italics** indicate arguments for which you supply actual values.

- Vertical bars (|) separate alternative, mutually exclusive elements.

- Square brackets [ ] indicate optional elements.

- Braces { } indicate a required choice.

- Braces within brackets [{}] indicate a required choice within an optional element.

**Note** This book covers multiple operating systems, and in each example, icons and router names indicate the OS that is being used. IOS and IOS XE use router names like R1 and R2 and are referenced by the IOS router icon. IOS XR routers use router names like XR1 and XR2 are referenced by the IOS XR router icon.
Introduction

For more than three decades, network management has been entirely based on the command-line interface (CLI) and legacy protocols such as SNMP. These protocols and methods are severely limited. The CLI, for example, is vendor specific, lacks a unified data hierarchy (sometimes even for platforms from the same vendor), and was designed primarily as a human interface. SNMP suffers major scaling problems, is not fit for writing configuration to devices, and overall, is very complex to implement and customize.

In essence, automation aims at offloading as much work from humans as possible and delegating that work to machines. But with the aforementioned legacy interfaces and protocols, machine-to-machine communication is neither effective nor efficient; and at times, close to impossible.

Moreover, device configuration and operational data have traditionally lacked a proper hierarchy and failed to follow a data model. In addition, network management workflows have always been far from mature, compared to software development workflows in terms of versioning, collaboration, testing, and automated deployments.

Enter network programmability. Programmability revolves around programmable interfaces, commonly referred to as application programming interfaces (APIs). APIs are interfaces that are designed primarily to be used for machine-to-machine communication. A Python program accessing a network router to retrieve or push configuration, without human intervention, is an example of a machine-to-machine interaction. Contrast this with the CLI, where a human needs to manually enter commands on a device and then visually inspect the output.

Network equipment vendors (for both physical and virtual equipment) are placing ever-increasing emphasis on the importance of managing their equipment using programmable interfaces, and Cisco is at the forefront of this new world. This new approach to managing a network provides several benefits over legacy methods, including the following:

- Normalizing the interface for interaction with network platforms by abstracting communication with these platforms and breaking the dependency of this communication on specific network OS scripting languages (for example, NX-OS, IOS XR, and Junos OS)
- Providing new methods of interacting with network platforms and, in the process, enabling and aligning with new technologies and architectures, such as SDN, NFV, and cloud
- Harnessing the power of programming to automate manual tasks and perform repetitive tasks efficiently
- Enabling rapid infrastructure and service deployment by using workflows for service provisioning
- Increasing the reliability of the network configuration process by leveraging error checking, validation, and rollback and minimizing human involvement in the configuration process
Using common software development tools and techniques for network configuration management, such as software development methodologies, versioning, staging, collaboration, testing, and continuous integration/continuous delivery.

This book covers all the major programmable interfaces used in the market today for network management. The book discusses the protocols, tools, techniques, and technologies on which network programmability is based. Programming, operating systems, and APIs are not new technologies. However, programmable interfaces on network platforms, and using these programmable interfaces to fully operate and maintain a network, along with the culture accompanying these new methods and protocols, may be (relatively) new. This book explains, in detail, all the major components of this new ecosystem.

Goals and Methods of This Book

This is a “fundamentals” book aimed at transitioning network engineers from a legacy network-based mindset to a software-based (and associated technologies) mindset. A book covering fundamentals generally struggles to cover as many subjects as possible with just enough detail. The fine balance between breadth and depth is challenging, but this book handles this challenge very well.

This book introduces the emerging network programmability and automation ecosystem based on programmable interfaces. It covers each protocol individually, in some significant detail, using the relevant RFCs as guiding documents. Protocol workflows, messages, and other protocol nuances tend to be dry, and at times boring, so to keep things interesting, practical examples are given wherever possible and relevant. You, the reader, can follow and implement these examples on your machine, which can be as simple as a Linux virtual machine with Python Version 3.x installed, and free tools to work with APIs, such as Postman and cURL. This book makes heavy use of the Cisco DevNet sandboxes, so in the majority of cases, you do not need a home lab to test and experiment with physical equipment.

A whole section of the book is dedicated to putting the knowledge and skills learned throughout the book to good use. One chapter covers programming Cisco platforms and another covers programming non-Cisco platforms. A third chapter in that same section is dedicated exclusively to Ansible. This book provides an abundance of hands-on practice.

The last chapter provides a way forward, discussing tools and technologies that you might want to explore after you are done with this book.

Who This Book Is For

This book is meant for the following individuals and roles, among others:

- Network architects and engineers who want to integrate programmability into their network designs
- NOC engineers monitoring and operating programmable networks or those who rely on network management systems that utilize programmability protocols
Network engineers designing, implementing, and deploying new network services
Software engineers or programmers developing applications for network management systems
Network and software engineers working with networks or systems involving SDN, NFV, or cloud technologies
Network engineers pursuing their Cisco DevNet certifications

Whether you are an expert network engineer with no prior programming experience or knowledge, or a software engineer looking to utilize your expertise in the network automation domain, after reading this book, you will fully understand the most commonly used protocols, tools, technologies, and techniques related to the subject, and you will be capable of effectively using the newly learned material to design, implement, and operate full-fledged programmable networks and the associated network automation systems.

How This Book Is Organized

This book covers the information you need to transition from having a focus on networking technology to focusing on software and network programmability. This book covers six main focus areas:

- Operating systems: Linux
- Software development: Python
- Transport: HTTP, REST, and SSH
- Encoding: XML, JSON, and YAML
- Modeling: YANG
- Protocols: NETCONF, RESTCONF, gRPC, and service provider programmability
- Practical programmability: Cisco platforms, non-Cisco platforms, and Ansible

Each chapter in this book either explicitly covers one of these focus areas or prepares you for one of them. Special consideration has been given to the ordering of topics to minimize forward referencing. Following an introduction to the programmability landscape, Linux is covered first because to get anything done in network programmability, you will almost always find yourself working with Linux. The book next covers Python because the vast majority of the rest of the book includes coverage of Python in the context of working with various protocols. The following chapters present an organic flow of topics: transport, encoding, modeling, and the protocols that build on all the previous sections. For example, understanding NETCONF requires you to understand SSH, XML, and YANG, and understanding RESTCONF requires that you understand HTTP, XML/JSON, and YANG. Both NETCONF and RESTCONF require knowledge of Python, most likely running on a Linux machine.
How This Book Is Structured

The book is organized into nine parts, described in the following sections.

PART I, “Introduction”

Chapter 1, “The Network Programmability and Automation Ecosystem”: This chapter introduces the concepts and defines the terms that are necessary to understand the protocols and technologies covered in the following chapters. It also introduces the network programmability stack and explores the different components of the stack that constitute a typical network programmability and automation toolbox.

PART II, “Linux”

Chapter 2, “Linux Fundamentals”: Linux is the predominant operating system used for running software for network programmability and automation. Linux is also the underlying operating system for the vast majority of network device software, such as IOS XR, NX-OS, and Cumulus Linux. Therefore, to be able to effectively work with programmable devices, it is of paramount importance to master the fundamentals of Linux. This chapter introduces Linux, including its architecture and boot process, and covers the basics of working with Linux through the Bash shell, such as working with files and directories, redirecting input and output, performing system maintenance, and installing software.

Chapter 3, “Linux Storage, Security, and Networks”: This chapter builds on Chapter 2 and covers more advanced Linux topics. It starts with storage on Linux systems and the Linux Logical Volume Manager. It then covers Linux user, group, file, and system security. Finally, it explains three different methods to manage networking in Linux; the `ip` utility, the NetworkManager service, and network configuration files.

Chapter 4, “Linux Scripting”: This chapter builds on Chapters 2 and 3 and covers Linux scripting using the Bash shell. The chapter introduces the `grep`, `awk`, and `sed` utilities and covers the syntax and semantics of Bash scripting. The chapter covers comments, input and output, variables and arrays, expansion, operations and comparisons, how to execute system commands from a Bash script, conditional statements, loops, and functions. It also touches on the Expect programming language.

PART III, “Python”

Chapter 5, “Python Fundamentals”: This chapter assumes no prior knowledge of programming and starts with an introduction to programming, covering some very important software and computer science concepts, including algorithms and object-oriented programming. It also discusses why programming is a foundational skill for learning network programmability and covers the fundamentals of the Python programming language,
including installing Python Version 3.x, executing Python programs, input and output, data types, data structures, operators, conditional statements, loops, and functions.

Chapter 6, “Python Applications”: This chapter builds on Chapter 5 and covers the application of Python to different domains. The chapter illustrates the use of Python for creating web applications using Django and Flask, for network programmability using NAPALM and Nornir, and for orchestration and machine learning. The chapter also covers some very important tools and protocols used in software development in general, such as Git, containers, Docker and virtual environments.

PART IV, “Transport”

Chapter 7, “HTTP and REST”: This is one of the most important chapters in this book. It introduces the HTTP protocol and the REST architectural framework, as well as the relationship between them. This chapter covers HTTP connections based on TCP. It also covers the anatomy of HTTP messages and dives into the details of HTTP request methods and response status codes. It also provides a comprehensive explanation of the most common header fields. The chapter discusses the syntax rules that govern the use of URIs and then walks through working with HTTP, using tools such as Postman, cURL, and Python libraries, such as the requests library.

Chapter 8, “Advanced HTTP”: Building on Chapter 7, this chapter moves to more advanced HTTP topics, including HTTP authentication and how state can be maintained over HTTP connections by using cookies. This chapter provides a primer on cryptography for engineers who know nothing on the subject and builds on that to cover TLS, and HTTP over TLS (aka HTTPS). It also provides a glimpse into HTTP/2 and HTTP/3, and the enhancements introduced by these newer versions of HTTP.

Chapter 9, “SSH”: Despite being a rather traditional protocol, SSH is still an integral component of the programmability stack. SSH is still one of the most widely used protocols, and having a firm understanding of the protocol is crucial. This chapter discusses the three sub-protocols that constitute SSH and cover the lifecycle of an SSH connection: the SSH Transport Layer Protocol, User Authentication Protocol, and Connection Protocol. It also discusses how to set up SSH on Linux systems as well as how to work with SSH on the three major network operating system: IOS XR, IOS XE, and NX-OS. Finally, it covers SFTP, which is a version of FTP based on SSH.

PART V, “Encoding”

Chapter 10, “XML”: This chapter covers XML, the first of three encoding protocols covered in this book. XML is the oldest of the three protocols and is probably the most sophisticated. This chapter describes the general structure of an XML document as well as XML elements, attributes, comments, and namespaces. It also covers advanced XML topics such as creating document templates using DTD and XML-based schemas using XSD, and it compares the two. This chapter also covers XPath, XSLT, and working with XML using Python.
Chapter 11, “JSON”: JSON is less sophisticated, newer, and more human-readable than XML, and it is therefore a little more popular than XML. This chapter covers JSON data formats and data types, as well as the general format of a JSON-encoded document. The chapter also covers JSON Schema Definition (JSD) for data validation and how JSD coexists with YANG.

Chapter 12, “YAML”: YAML is frequently described as a superset of JSON. YAML is slightly more human-readable than JSON, but data encoded in YAML tends to be significantly lengthier than its JSON-encoded counterpart. YAML is a very popular encoding format and is required for effective use of tools such as Ansible. This chapter covers the differences between XML, JSON, and YAML and discusses the structure of a YAML document. It also explains collections, scalers, tags, and anchors. Finally, the chapter discusses working with YAML in Python.

PART VI, “Modeling”

Chapter 13, “YANG”: At the heart of the new paradigm of network programmability is data modeling. This is a very important chapter that covers both generic modeling and the YANG modeling language. This chapter starts with a data modeling primer, explaining what a data model is and why it is important to have data models. Then it explains the structure of a data model. This chapter describes the different node types in YANG and their place in a data model hierarchy. It also delves into more advanced topics, such as augmentations and deviations in YANG. It describes the difference between open-standard and vendor-specific YANG models and where to get each type. Finally, the chapter covers a number of tools for working with YANG modules, including pyang and pyangbind.

PART VII, “Protocols”

Chapter 14, “NETCONF and RESTCONF”: NETCONF was the first protocol developed to replace SNMP. RESTCONF was developed later and is commonly referred to as the RESTful version of NETCONF. Building on earlier chapters, this chapter takes a deep dive into both NETCONF and RESTCONF. The chapter covers the protocol architecture as well as the transport, message, operations, and content layers of each of the two protocols. It also covers working with these protocols using Python.

Chapter 15, “gRPC, Protobuf, and gNMI”: The gRPC protocol was initially developed by Google for network programmability that borrows its operational concepts from the communications models of distributed applications. This chapter provides an overview of the motivation that drove the development of gRPC. It covers the communication flow of gRPC and protocol buffers (Protobuf) used to serialize data for gRPC communications. The chapter also shows how to work with gRPC using Python. The chapter then takes a deep dive into gNMI, a gRPC-based specification. Finally, the chapter shows how gRPC and gNMI are used to manage a Cisco IOS XE device.
Chapter 16, “Service Provider Programmability”: Service providers face unique challenges due to the typical scale of their operations and the stringent KPIs that must be imposed on their networks, especially given the heated race to adopt 5G and associated technologies. This chapter discusses how such challenges influence the programmability and automation in service provider networks and provides in-depth coverage of Segment Routing, BGP-LS, and PCEP.

PART VIII, “Programmability Applications”

Chapter 17, “Programming Cisco Platforms”: This chapter explores the programmability capabilities of several Cisco platforms, covering a wide range of technology domains. In addition, this chapter provides several practical examples and makes heavy use of Cisco’s DevNet sandboxes. This chapter covers the programmability of IOS XE, IOS XR, NX-OS, Meraki, DNA Center, and Cisco’s collaboration platforms, with a use case covering Webex Teams.

Chapter 18, “Programming Non-Cisco Platforms”: This chapter covers the programmability of a number of non-Cisco platforms, such as the Cumulus Linux and Arista EOS platforms. This chapter shows that the knowledge and skills gained in the previous chapters are truly vendor neutral and global. In addition, this chapter shows that programmability using APIs does in fact abstract network configuration and management and breaks the dependency on vendor-specific CLIs.

Chapter 19, “Ansible”: This chapter covers a very popular tool that has become synonymous with network automation: Ansible. As a matter of fact, Ansible is used in the application and compute automation domains as well. Ansible is a very simple, yet extremely powerful, automation tool that provides a not-so-steep learning curve, and hence a quick and effective entry point into network automation. This is quite a lengthy chapter that takes you from zero to hero in Ansible.

PART IX, “Looking Ahead”

Chapter 20, “Looking Ahead”: This chapter builds on the foundation covered in the preceding chapters and discusses more advanced technologies and tools that you might want to explore to further your knowledge and skills related to network programmability and automation.
Chapter 2, “Linux Fundamentals,” covers Linux basics, and by now, you should be familiar with the Linux environment and feel comfortable performing general system maintenance tasks. This chapter takes you a step further in your Linux journey and covers storage, security, and networking.

Linux Storage

Many network engineers struggle with concepts such as what mounting a volume means and the relationship between physical and logical volumes. This section covers everything you need to know about storage to effectively manage a Linux-based environment, whether it is your development environment or the underlying Linux system on which a network operating system is based, such as IOS XR and NX-OS.

Physical Storage

The /dev directory contains device files, which are special files used to access the hardware on a system. A program trying to access a device uses a device file as an interface to the device driver of that device. Writing data to a device file is the same as sending data to the device represented by that device file, and reading data from a device file is the same as receiving data from that device. For example, writing data to the printer device file prints this data, and reading data from the device file of a hard disk partition is the same as reading data from that partition on the disk.

Example 3-1 shows the output of the ls -l command for the /dev directory. Notice that, unlike other directories, the first bit of the file permissions is one of five characters:

- for regular files
- d for directories
120 Chapter 3: Linux Storage, Security, and Networks

- l for links
- c for character device files
- b for block device files

You learned about the first three of these bits in Chapter 2, and the other two are covered here.

**Example 3-1 Contents of the /dev Directory**

```bash
[netdev@server1 dev]$ ls -l
total 0
-rw-r--r--. 1 root    root           0 Aug 10 00:28 any_regular_file
-crw-r--r--. 1 root    root     10, 235 Aug 10 00:19 autofs
-drwxr-xr-x. 2 root    root         140 Aug 10 00:18 block
-drwxr-xr-x. 2 root    root          60 Aug 10 00:18 bsg
-drwxr-xr-x. 2 root    root          60 Aug 10 00:19 bus
-drwxr-xr-x. 2 root    root        2940 Aug 10 00:20 char
-drwxr-xr-x. 2 root    root          80 Aug 10 00:18 cl
-crw-------. 1 root    root      5,   1 Aug 10 00:20 console
-lrwxrwxrwx. 1 root    root          11 Aug 10 00:18 core -> /proc/kcore
-drwxr-xr-x. 6 root    root         120 Aug 10 00:19 cpu
-crwx--------. 1 root    root          10 Aug 10 00:19 cpu_dma_latency
-drwxr-xr-x. 6 root    root         120 Aug 10 00:18 disk
-brw-rw----. 1 root    disk    253,   0 Aug 10 00:19 dm-0
-brw-rw----. 1 root    disk    253,   1 Aug 10 00:19 dm-1
```

--- OUTPUT TRUNCATED FOR BREVITY ---

Character device files provide *unbuffered* access to hardware. This means that what is written to the file is transmitted to the hardware device right away, byte by byte. The same applies to read operations. Think of data sent to the device file of an audio output device or data read from the device file representing your keyboard. This data should not be buffered.

On the other hand, block device files provide *buffered* access; that is, data written to a device file is buffered by the kernel before it is passed on to the hardware device. The same applies to read operations. Think of data written to or read from a partition on your hard disk. This is typically done in data blocks, not individual bytes.

However, note that the device file type (as seen in the /dev directory) is not necessarily the same as the device type. Storage devices such as hard disks are block devices, which means that data is read from and written to the device in fixed-size blocks. Although this may sound counterintuitive, block devices may be accessed using character device files on some operating systems, such as BSD. This is not the case with Linux, where block
devices are always associated with block device files. The difference between block
devices and block device files is sometimes a source of confusion.

The first step in analyzing a storage and file system is getting to know the hard disks.
Each hard disk and partition has a corresponding device file in the /dev directory. By
listing the contents of this directory, you find the sda file for the first hard disk, and, if
installed, sdb for the second hard disk, sdc for the third hard disk, and so forth. Partitions
are named after the hard disk that the partition belongs to, with the partition number
appended to the name. For example, the first partition on the second hard disk is
named sdb1. The hard disk naming convention follows the configuration in the /lib/udev/
rules.d/60-persistent-storage.rules file, and the configuration is per hard disk type (ATA,
USB, SCSI, SATA, and so on). Example 3-2 lists the relevant files in the /dev directory on
a CentOS 7 distro. As you can see, this system has two hard disks. The first hard disk is
named sda and has two partitions—sda1 and sda2—and the second is named sdb and has
three partitions—sdb1, sdb2, and sdb3.

Example 3-2  Hard Disks and Partitions in the /dev Directory

```
[root@localhost ~]# ls -1 /dev | grep sd
brw-rw----. 1 root    disk      8,   0 Jun  8 04:55 sda
brw-rw----. 1 root    disk      8,   1 Jun  8 04:55 sda1
brw-rw----. 1 root    disk      8,   2 Jun  8 04:55 sda2
brw-rw----. 1 root    disk      8,  16 Jun  8 04:55 sdb
brw-rw----. 1 root    disk      8,  17 Jun  8 04:55 sdb1
brw-rw----. 1 root    disk      8,  18 Jun  8 04:55 sdb2
brw-rw----. 1 root    disk      8,  19 Jun  8 04:55 sdb3
```

Notice the letter b at the beginning of each line of the output in Example 3-2. This indi-
cates a block device file. A character device file would have the letter c instead.

The command `fdisk -l` lists all the disks and partitions on a system, along with some use-
ful details. Example 3-3 shows the output of this command for the same system as in
Example 3-2.

Example 3-3  Using the fdisk -l Command to Get Hard Disk and Partition Details

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 26.8 GB, 26843545600 bytes, 52428800 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x000b4fba
```
In addition to physical disks /dev/sda and /dev/sdb and their respective partitions, the command output in Example 3-3 lists two other disks: /dev/mapper/centos-root and /dev/mapper/centos-swap. These are two logical volumes. (Logical volumes are discussed in detail in the next section.) Notice that there is an asterisk (*) under the title Boot for partition /dev/sda1. As you may have guessed, this indicates that this is the partition on which the boot sector resides, containing the boot loader. The boot loader is the software that will eventually load the kernel image into memory during the system boot process, as you have read in Section “The Linux Boot Process” in Chapter 2.

In addition to displaying existing partition details, fdisk can create new partitions and delete existing ones. For example, after a third hard disk, sdc, is added to the system, the fdisk utility can be used to create two partitions, sdc1 and sdc2, as shown in Example 3-4.
Example 3-4  Creating New Hard Disk Partitions by Using the fdisk Utility

Current status of the sdc hard disk: no partitions exist
[root@localhost ~]# fdisk -l /dev/sdc

Disk /dev/sdc: 21.5 GB, 21474836480 bytes, 41943040 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

Using fdisk to create two new partitions on sdc
[root@localhost ~]# fdisk /dev/sdc
Welcome to fdisk (util-linux 2.23.2).

Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table
Building a new DOS disklabel with disk identifier 0x4cd00767.

Command (m for help): m
Command action
  a toggle a bootable flag
  b edit bsd disklabel
  c toggle the dos compatibility flag
  d delete a partition
  g create a new empty GPT partition table
  G create an IRIX (SGI) partition table
  l list known partition types
  m print this menu
  n add a new partition
  o create a new empty DOS partition table
  p print the partition table
  q quit without saving changes
  s create a new empty Sun disklabel
  t change a partition’s system id
  u change display/entry units
  v verify the partition table
  w write table to disk and exit
  x extra functionality (experts only)

Command (m for help): n
Partition type:
  p primary (0 primary, 0 extended, 4 free)
  e extended
The interactive dialogue of the `fdisk` utility is self-explanatory. After the `fdisk /dev/sdc` command is issued, you can enter `m` to see all available options. You can enter `n` to start the new partition dialogue. Note the different methods to specify the size of the
partition. If you go with the default option (by simply pressing Enter), the command uses all the remaining space on the disk to create that particular partition.

Before a hard disk partition can be used to store data, the partition needs to be formatted; that is, a file system has to be created. (File systems are discussed in some detail in Chapter 2.) At the time of writing, the two most common file systems used on Linux are ext4 and xfs. A partition is formatted using the `mkfs` utility. In Example 3-5, the sdc1 partition is formatted to use the ext4 file system, and sdc2 is formatted to use the xfs file system.

**Example 3-5  Creating File Systems by Using the `mkfs` Command**

```
[root@localhost ~]# mkfs -t ext4 /dev/sdc1
mke2fs 1.42.9 (28-Dec-2013)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
32768 inodes, 1310720 blocks
65536 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=1342177280
40 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
     32768, 98304, 163840, 229376, 294912, 819200, 884736
Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
[root@localhost ~]# mkfs -t xfs /dev/sdc2
meta-data=/dev/sdc2 isize=512 agcount=4, agsize=982976 blks
  = sectsz=512 attr=2, projid32bit=1
  = crc=1 finobt=0, sparse=0
data  = bsize=4096 blocks=3931904, imaxpct=25
  = sunit=0 swidth=0 blks
naming =version 2 bsize=4096 ascii-ci=0 ftype=1
log   =internal log bsize=4096 blocks=2560, version=2
  = sectsz=512 sunit=0 blks, lazy-count=1
realtime =none extsz=4096 blocks=0, rtextents=0
[root@localhost ~]#```
To specify a file system type, you use `mkfs` with the `-t` option. Keep in mind that the command output depends on the file system type used with the command.

The final step toward making a partition usable is to mount that partition or file system. Mounting is usually an ambiguous concept to engineers who are new to Linux. As discussed in Chapter 2, the Linux file hierarchy always starts at the root directory, represented by `/`, and branches down. For a file system to be accessible, it has to be mounted to a mount point—that is, attached (mounted) to the file hierarchy at a specific path in that hierarchy (mount point). The mount point is the path in the file hierarchy that the file system is attached to and through which the contents of that file system can be accessed. For example, mounting the `/dev/sdc1` partition to the `/Operations` directory maps the content of `/dev/sdc1` to, and makes it accessible through, the `/Operations` directory, for both read and write operations. Example 3-6 shows the `/Operations` directory being created and the sdc1 partition being mounted to it.

**Example 3-6  Mounting /dev/sdc1 to /Operations**

```
[root@localhost ~]# mkdir /Operations
[root@localhost ~]# mount /dev/sdc1 /Operations
```

To display all the mounted file systems, you use the `df` command, as shown in Example 3-7. The option `-h` displays the file system sizes in human-readable format.

**Example 3-7  Output of the df -h Command**

```
[root@localhost ~]# df -h
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   22G  5.3G   17G  25% /
devtmpfs                 3.9G  0.0G  3.9G   0% /dev
tmpfs                    3.9G  0.0G  3.9G   0% /dev/shm
tmpfs                    3.9G  9.4M  3.9G   1% /run
tmpfs                    3.9G  0.0G  3.9G   0% /sys/fs/cgroup
/dev/sda1               1014M  333M  682M  33% /boot
tmpfs                    783M  12K  783M   1% /run/user/1000
/dev/sdc1                4.8G  20M  4.6G   1% /Operations
[root@localhost ~]#
```

Each row in the output in Example 3-7 is a separate file system. The entry `/dev/mapper/centos-root` is a logical volume (and is discussed in detail in the next section). The following few entries are tmpfs file systems, which are temporary file systems created in memory (not on disk) for cache-like operations due to the high speed of RAM, as compared to the low speed of hard disks. An entry exists in the list for partition `/dev/sda1` that is mounted to directory `/boot`. Then the entry at the bottom is for `/dev/sdc1` that was mounted to directory `/Operations` as shown in Example 3-6.
To unmount the /dev/sdc1 file system, you use the `umount /dev/sdc1` command. You can also use the mount point, in which case the command is `umount /Operations`. Note that the command is `umount`, not `unmount`. Adding the letter `n` is a very common error.

The mounting done by using the `mount` command is not persistent. In other words, once the system is rebooted, the volumes mounted using the `mount` command are no longer mounted. For persistent mounting, an entry needs to be added to the /etc/fstab file. Example 3-8 shows the contents of the /etc/fstab file after the entry for /dev/sdc1 is added.

**Example 3-8  Editing the /etc/fstab File for Persistent Mounting**

```bash
# Adding an entry for /etc/sdc1 using the echo command
[root@localhost ~]# echo "/dev/sdc1 /Operations    ext4 defaults 0 0" >> /etc/fstab

# After adding an entry for /etc/sdc1
[root@localhost ~]# cat /etc/fstab

# /etc/fstab
# Created by anaconda on Sat May 26 04:28:54 2018
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/centos-root /            xfs     defaults        0 0
UUID=dfe65618-19ab-458d-b5e3-dafdb59b4e68 /boot      xfs    defaults   0 0
/dev/mapper/centos-swap swap            swap    defaults        0 0
/dev/sdc1 /Operations            ext4   defaults    0 0

# Command mount -a immediately mounts all file systems listed in fstab
[root@localhost ~]# mount -a
```

The command `mount -a` immediately mounts all file systems listed in /etc/fstab.

The /etc/fstab file has one entry for each file system that is to be mounted at system boot. It is important to understand the entries in the /etc/fstab file because this is the file that defines what file systems a system will have mounted right after it boots and the options that each of these file systems will be mounted with. Each line has the following fields:

- The first field can be either the file system path, the universal unique identifier (UUID), or the label. You can learn the UUID (and type) of all file systems by using the command `blkid`. You can show the label by using the command `tune2fs -l [file_system]` for ext2/3/4 file systems or `xfs_admin -l [file_system]` for xfs file systems. Using the file system path, which is /dev/sdc1 in this case, is pretty straightforward. However, when a system has tens of hard disks installed, it would be wiser to use the partition UUIDs. A UUID is a unique number that identifies a partition. The UUID does not change if the hard disk containing the partition is moved to another system, and hence it is universal. Using a UUID eliminates the possibility of errors in the /etc/fstab file.
The second field is the file system mount point, which is /Operations in this case.

The third field is the file system type, which is ext4 in this example.

The fourth field is the mounting options. In this example, defaults indicates that the default mounting options will be used. You can also add non-default mounting options such as acl for ACL support. You add options in a comma-separated list.

The fifth field indicates which file systems are to be backed up by the `dump` utility. The zero value in this case indicates that this file system will not be automatically backed up.

The sixth field is used by the `fsck` utility to determine whether to check the health of the file system. The `fsck` utility checks file systems with a nonzero value in this field, in order, starting with the file system that has the value one. A zero in this field tells the `fsck` utility not to check that file system.

`fdisk` is not the only Linux utility available to manipulate disk partitions. Two other popular utilities for disk partitioning are `gdisk` and `parted`. You can use the man pages for these utilities to explore them and use a non-production environment (ideally a virtual machine) to experiment with using them. You may run into a distro that has one of them implemented but not the other. The more utilities you are familiar with, the better.

### Logical Volume Manager

Linux generally uses the concept of logical volumes to provide storage to users. Logical volumes abstract the storage that is available to a user from the actual physical disks. Logical volumes on Linux are managed by system software called **Logical Volume Manager (LVM)**. LVM operates by grouping physical disks or disk partitions, each referred to as a *physical volume (PV)*, such as /dev/sda or /dev/sdb2, into a *volume group (VG)*. LVM then manages a VG as one pool of storage that is split by the LVM into one or more *logical volumes (LVs)*. Figure 3-1 illustrates these concepts.

![Figure 3-1](image-url)
To better understand the concept of logical volumes, keep in mind the following:

- The different PVs that constitute a VG do not have to be equal in size.
- The different PVs that constitute a VG may be different disks, or different partitions on the same disk, or different partitions on different disks.
- Two different partitions on the same disk may be members of two different VGs.
- The LVs that are created from a VG do not correlate to the PVs that constitute the VG in either size or number.

Using LVs created by LVM provides several advantages over using physical storage directly. The most significant benefit is the disassociation between user data and specific physical storage volumes. From a capacity perspective, capacity can be added to and removed from a logical volume without having to repartition a physical disk to create a bigger or smaller partition, and a file system is not limited by the size of the physical disk that it resides on. From a performance perspective, data may be striped across several physical volumes (for added throughput) transparently from the user. These are just a few of the advantages.

The following steps are involved in creating a logical volume that is ready to use:

**Step 1.** Using the command `pvcreate physical_disk/partition`, label the physical volumes that will constitute the volume group as LVM physical volumes.

**Step 2.** Using the command `vgcreate vg_name pv1 pv2 .. pvN`, create the VG by using the physical volumes `pv1`, `pv2`, ..., `pvN`.

**Step 3.** Using the command `lvcreate -n lv_name -L lv_size vg_name`, create the logical volume named `lv_name` from the volume group named `vg_name`.

**Step 4.** Create the file system of choice on the new logical volume by using the `mkfs` command, exactly as you would on a physical partition.

**Step 5.** Mount the new file system by using the `mount` command exactly as you would mount a file system created on a physical partition.

In Example 3-9, two disks, sdb and sdc, are each divided into two partitions as follows:

- sdb1: 12 GB
- sdb2: 8 GB
- sdc1: 15 GB
- sdc2: 10 GB
Example 3-9  The Four Partitions That Will Be Used to Create a Volume Group

<table>
<thead>
<tr>
<th>[root@server1 ~]# fdisk -l</th>
<th>grep -E sd[b,c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk /dev/sdc: 26.8 GB, 26843545600 bytes, 52428800 sectors</td>
<td></td>
</tr>
<tr>
<td>/dev/sdc1            2048    31459327    15728640   83  Linux</td>
<td></td>
</tr>
<tr>
<td>/dev/sdc2        31459328    52428799    1048736   83  Linux</td>
<td></td>
</tr>
<tr>
<td>Disk /dev/sdb: 21.5 GB, 21474836480 bytes, 41943040 sectors</td>
<td></td>
</tr>
<tr>
<td>/dev/sdb1            2048    25167871    12582912   83  Linux</td>
<td></td>
</tr>
<tr>
<td>/dev/sdb2        25167872    41943039    8387584   83  Linux</td>
<td></td>
</tr>
<tr>
<td>[root@server1 ~]#</td>
<td></td>
</tr>
</tbody>
</table>

After each of the four partitions is labeled as a PV, all four partitions are added to the VG VGNetProg, which has a total capacity of 40 GB. The volume group capacity is then used to create two logical volumes—LVNetAutom with a capacity of 10 GB and LVNetDev with a capacity of 30 GB—as shown in Example 3-10.

Example 3-10  Creating Physical Volumes, Volume Groups, and Logical Volumes

<table>
<thead>
<tr>
<th>Label the physical volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@server1 ~]# pvcreate /dev/sdb1 /dev/sdb2 /dev/sdc1 /dev/sdc2</td>
</tr>
<tr>
<td>Physical volume &quot;/dev/sdb1&quot; successfully created.</td>
</tr>
<tr>
<td>Physical volume &quot;/dev/sdb2&quot; successfully created.</td>
</tr>
<tr>
<td>Physical volume &quot;/dev/sdc1&quot; successfully created.</td>
</tr>
<tr>
<td>Physical volume &quot;/dev/sdc2&quot; successfully created.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Create the volume group</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@server1 ~]# vgcreate VGNetProg /dev/sdb1 /dev/sdb2 /dev/sdc1 /dev/sdc2</td>
</tr>
<tr>
<td>Volume group &quot;VGNetProg&quot; successfully created</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Create the two logical volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@server1 ~]# lvcreate -n LVNetAutom -L 10G VGNetProg</td>
</tr>
<tr>
<td>Logical volume &quot;LVNetAutom&quot; created.</td>
</tr>
<tr>
<td>[root@server1 ~]# lvcreate -n LVNetDev -L 30G VGNetProg</td>
</tr>
<tr>
<td>Logical volume &quot;LVNetDev&quot; created.</td>
</tr>
<tr>
<td>[root@server1 ~]#</td>
</tr>
</tbody>
</table>

Example 3-11 shows the pvdisplay command being used to display the details of the physical volumes.
Example 3-11  Displaying Physical Volume Details by Using the pvdisplay Command

<table>
<thead>
<tr>
<th>PV Name</th>
<th>VG Name</th>
<th>PV Size</th>
<th>Allocatable</th>
<th>PE Size</th>
<th>Total PE</th>
<th>Free PE</th>
<th>Allocated PE</th>
<th>PV UUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/sdb1</td>
<td>VGNetProg</td>
<td>12.00 GiB /</td>
<td>yes</td>
<td>4.00 MiB</td>
<td>3071</td>
<td>511</td>
<td>2560</td>
<td>dPYPj6-Wv1i-iX7H-3iH0-oChE-0zkA-2LcJlx</td>
</tr>
<tr>
<td>/dev/sdb2</td>
<td>VGNetProg</td>
<td>&lt;8.00 GiB /</td>
<td>yes</td>
<td>4.00 MiB</td>
<td>2047</td>
<td>765</td>
<td>1282</td>
<td>ftOYQo-a19G-0Gs6-01ir-i6M5-Yj1N-TRREDR</td>
</tr>
<tr>
<td>/dev/sdc1</td>
<td>VGNetProg</td>
<td>15.00 GiB /</td>
<td>yes (but full)</td>
<td>4.00 MiB</td>
<td>3839</td>
<td>0</td>
<td>3839</td>
<td>DYW0TD-vXGl-8Sr-BCyQ-SLQQ-mkfi-rvQFVd</td>
</tr>
<tr>
<td>/dev/sdc2</td>
<td>VGNetProg</td>
<td>&lt;10.00 GiB /</td>
<td>yes (but full)</td>
<td>4.00 MiB</td>
<td>2559</td>
<td>0</td>
<td>2559</td>
<td>n1snhx-aevL-X5ay-1a43-1jlo-83uC-LIkIT7</td>
</tr>
</tbody>
</table>
Example 3-12 shows the `vgdisplay` command being used to display the volume group that has been created.

**Example 3-12 Displaying Volume Group Details by Using the `vgdisplay` Command**

```
[root@server1 ~]# vgdisplay VGNetProg
--- Volume group ---
VG Name               VGNetProg
System ID
Format                lvm2
Metadata Areas        4
Metadata Sequence No  3
VG Access             read/write
VG Status             resizable
MAX LV                0
Cur LV                2
Open LV               0
Max PV                0
Cur PV                4
Act PV                4
VG Size               44.98 GiB
PE Size               4.00 MiB
Total PE              11516
Alloc PE / Size       10240 / 40.00 GiB
Free PE / Size        1276 / 4.98 GiB
VG UUID               PSi3RJ-91kc-1ZFE-oCVA-RaXC-HDh5-K0VuV3
[root@server1 ~]#
```

Example 3-13 shows the `lvdisplay` command being used to display the logical volumes that have been created. A logical volume is addressed using its full path in the `/dev` directory, as shown in the example.

**Example 3-13 Displaying Logical Volume Details by Using the `lvdisplay` Command**

```
[root@server1 ~]# lvdisplay /dev/VGNetProg/LVNetAutom
--- Logical volume ---
LV Path                /dev/VGNetProg/LVNetAutom
LV Name                LVNetAutom
VG Name                VGNetProg
LV UUID                Y09QdN-J3Fw-s3NB-RB84-b8Ps-lUSv-tzMFaw
LV Write Access        read/write
LV Creation host, time server1, 2018-08-05 21:57:42 +0300
LV Status              available
# open                 0
LV Size                10.00 GiB
```
Current LE: 2560
Segments: 1
Allocation: inherit
Read ahead sectors: auto
- currently set to: 8192
Block device: 253:2

[root@server1 ~]# lvdisplay /dev/VGNetProg/LVNetDev
--- Logical volume ---
LV Path: /dev/VGNetProg/LVNetDev
LV Name: LVNetDev
VG Name: VGNetProg
LV UUID: Z9VRTv-CUE6-uSa8-8821-}GY5-ymKh-zsKfHZ
LV Write Access: read/write
LV Creation host, time: server1, 2018-08-05 21:58:17 +0300
LV Status: available
# open: 0
LV Size: 30.00 GiB
Current LE: 7680
Segments: 3
Allocation: inherit
Read ahead sectors: auto
- currently set to: 8192
Block device: 253:3

Note that you can issue the `pvdisplay`, `vgdisplay`, and `lvdisplay` commands without any arguments to display all physical volumes, all volume groups, and all logical volumes, respectively, that are configured on the system.

To delete a physical volume, volume group, or logical volume, you use the commands `pvremove`, `vgremove`, or `lvremove`, respectively.

After logical volumes are created, you use the `mkfs` command to format the LVNetAutom LV as an ext4 file system and the LVNetDev LV as an xfs file system, as shown in Example 3-14.

**Example 3-14  Creating File Systems on the new Logical Volumes by Using the `mkfs` Command**

```
[root@server1 ~]# mkfs -t ext4 /dev/VGNetProg/LVNetAutom
mke2fs 1.42.9 (28-Dec-2013)
Filesystem label= 
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
```
Finally, in Example 3-15, both logical volumes are mounted, which means they are usable for storing and retrieving data.

**Example 3-15  Mounting Both Logical Volumes by Using the `mount` Command**

```bash
[root@server1 ~]# mkdir /Automation
[root@server1 ~]# mkdir /Development
[root@server1 ~]# ls /

Automation  dev  hd3  lib64  opt  root  srv  usr
bin  Development  home  media  proc  run  sys  var
boot  etc  lib  mnt  Programming  sbin  tmp
```

```bash
[root@server1 ~]# mount /dev/VGNetProg/LVNetAutom /Automation
[root@server1 ~]# mount /dev/VGNetProg/LVNetDev /Development/
[root@server1 ~]# df -h

Filesystem               Size  Used  Avail Use% Mounted on
/dev/mapper/centos-root  44G   6.7G   38G  16% /
devtmpfs                 3.9G    0   3.9G   0% /dev
```
Of course, the mounting done in Example 3-15 is not persistent. To mount both logical volumes during system boot, two entries need to be added to the /etc/fstab file—one entry for each LV.

You may have noticed in the output of the `df -h` command in Example 3-15 that each LV appears as a subdirectory to the directory `/dev/mapper`. The device mapper is a kernel space driver that provides the generic function of creating mappings between different storage volumes. The term *generic* is used here because the mapper is not particularly aware of the constructs used by LVM to implement logical volumes. LVM uses the device mapper to create the mappings between a volume group and its constituent logical volumes, without the device mapper explicitly knowing that the latter is a logical volume (rather than a physical one).

The examples in this section show only the very basic functionality of LVM—that is, creating the basic building blocks for having and using logical volumes on a system. However, the real power of LVM becomes clear when you use advanced features such as increasing or decreasing the size of a logical volume, without having to delete the volume and re-create it, or the several options for high availability of logical volumes. Red Hat has a 147-page document titled “Logical Volume Manager Administration” on managing logical volumes. You can check out the document for RHEL 8 at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_logical_volumes/index.

---

**Linux Security**

Linux security is a massive and complex topic so it is important to establish the intended scope of this section early on. The purpose of this section is two-fold. The first purpose is to familiarize you with basic Linux security operations that would enable you to effectively manage your development environment without being stumped. For example, you can’t execute a script unless your user on the system has the privileges to execute that script, based on the script’s file permissions and your group memberships. The second purpose of this section is to show you how to accomplish a minimal level of hardening for your development environment. Using an unsecured device to run scripts that access network devices—and possibly push configuration to those devices—is not a wise thing to do. Accordingly, this section covers user, group, file, and directory security, including access control lists. This chapter also covers the Linux firewall.
User and Group Management

Linux is a multiuser operating system, which means that more than one user can access a single Linux system at a time.

For a user to access a Linux system, the user’s account must be configured on the system. The user will then have a username and user ID (UID). A user with an account on the system is a member of one or more groups. Each group has a group name and a group ID (GID). By default, when a user is created on the system, a new group is also created; it has the same name as the username, and this becomes the primary group of the user. A user typically has a password, and each group also has a password.

Each user has a home directory that contains that user's files. One way that Linux maintains user segregation and security is by maintaining permissions on files and directories and allowing users with the appropriate authorization level to set those permissions. File permissions are classified into permissions for the owner of the file, the group of the file, and everyone else. The root user and any other user with root privileges can access all resources on the system, including other users' files and directories. The root user and users with root privileges are members of a group named wheel.

You can find user information by using the command `id {username}`, as shown in Example 3-16 for user NetProg.

Example 3-16  Getting User Information by Using the id Command

```
[root@localhost ~]# id NetProg
uid=1001(NetProg) gid=1002(NetProg) groups=1002(NetProg),10(wheel)
[root@localhost ~]#
```

User NetProg's UID is 1001. The output in Example 3-16 shows that the user's default (primary) group has the same name as the username. User NetProg in the example is also a member of the wheel group and therefore has root privileges that can be invoked by using the `sudo {command}` command, where `command` requires root privileges to be executed. The number to the left of each group name is the group ID.

User information is also stored in the /etc/passwd file, and group information is stored in the /etc/group file. Hashed user passwords are stored in the file /etc/shadow, and hashed group passwords are stored in the file /etc/gshadow. Example 3-17 displays the last five entries of each of the files.

Example 3-17  Last Five Entries from the /etc/passwd, /etc/group, /etc/shadow, and /etc/gshadow Files

```
! Sample entries from the /etc/passwd file
[netdev@server1 ~]$ tail -n 5 /etc/passwd
netdev:x:1000:1000:Network Developer:/home/netdev:/bin/bash
vboxadd:x:970:1::/var/run/vboxadd:/bin/false
cockpit-wsinstance:x:969:969:User for cockpit-ws instances:/nonexisting:/sbin/nologin
```
flatpak:x:968:968:User for flatpak system helper:/sbin/nologin
rngd:x:967:967:Random Number Generator Daemon:/var/lib/rngd:/sbin/nologin

[netdev@server1 ~]$ ! Sample entries from the /etc/group file
[netdev@server1 ~]$ tail -n 5 /etc/group
netdev:x:1000:
vboxsf:x:970:
cockpit-wsinstance:x:969:
flatpak:x:968:
rngd:x:967
[netdev@server1 ~]$ !

[netdev@server1 ~]$ file /etc/shadow requires root privileges to be read
[netdev@server1 ~]$ tail -n 5 /etc/shadow
tail: cannot open '/etc/shadow' for reading: Permission denied
[netdev@server1 ~]$ !

[netdev@server1 ~]$ file /etc/gshadow requires root privileges to be read
[netdev@server1 ~]$ tail -n 5 /etc/gshadow
tail: cannot open '/etc/gshadow' for reading: Permission denied
[netdev@server1 ~]$ !

[netdev@server1 ~]$ sudo tail -n 5 /etc/shadow
[sudo] password for netdev:
netdev:$6$JUG9NvdC/NzqiYq$zpCkMR3eENFgk906PjFVL5S26qFRI9L2n13rFApiyPS01gb2F1CTjJv6c1dqvEvhX9lq2fK.p3hv1EYKciiD2..18489:0:99999:7:::
vboxadd:!!!:18473::::::
cockpit-wsinstance:!!!:18473::::::
flatpak:!!!:18473::::::
rngd:!!!:18473::::::
[netdev@server1 ~]$ !

[netdev@server1 ~]$ sudo tail -n 5 /etc/gshadow
[netdev@server1 ~]$ sudo tail -n 5 /etc/gshadow
[netdev:!!!:
vboxsf:!!!:
cockpit-wsinstance:!!!:
flatpak:!!!:
rngd:!!!:
[netdev@server1 ~]$ !
Each line in the /etc/passwd file is a record containing the information for one user account. Each record is formatted as follows: *username:*x:*user_id:*primary_group_id:*user_extra_information:*user_home_directory:*user_default_shell.*

The /etc/passwd and /etc/group files can be read by any user on the system but can only be edited by a user with root privileges. For this reason, as a security measure, the second field in the record, which historically contained the user password hash, now shows only the letter x. The user password hashes are now maintained in the /etc/shadow file, which can only be read by users with root privileges. The same arrangement is true for the /etc/group and the /etc/gshadow files. Whenever a user does not have a password, the x is omitted. Two consecutive colons in any record indicate missing information for the respective field.

Each line in the /etc/group file is a record containing information for one group. Each record is formatted as follows: *groupname:*x:*group_id:*group_members.* The last field is a comma-separated list of non-default users in the group. For example, the record for the netdev group shows all users who are members of the group netdev except the user netdev itself.

Each line in the /etc/shadow file is a record containing the password information for one user. Each record is formatted as follows: *username:*password_hash:*last_changed:*min:*max:*warn:*expired:*disabled:*reserved.*

The field *last_changed* is the number of days between January 1, 1970, and the date the password was last changed. The field *min* is the minimum number of days to wait before the password can be changed. The value 0 indicates that it may be changed at any time. The field *max* is the number of days after which the password must be changed. The value 99999 means that the user can keep the same password practically forever. The field *warn* is the number of days to send a warning to the user prior to the password expiring. The field *expired* is the number of days after the password expires before the account should be disabled. The field *disabled* is the number of days since January 1, 1970, that an account has been disabled. The last field is reserved.

Finally, each line in the /etc/gshadow file is a record that contains the password information for one group. Each record is formatted as follows: *groupname:*group_password_hash:*group_admins:*group_members.* The *group_password_hash* field contains an exclamation symbol (!) if no user is allowed to access the group by using the *newgrp* command. (This command is covered later in this section.)

You use the command *useradd* [*username*] to create a new user, and the command *passwd* [*username*] to set or change the password for a user. After switching to user root by using the *su* command in Example 3-18, the *id* NetDev command is used to verify that user NetDev does not already exist. The new user NetDev is then created by issuing the command *useradd* NetDev.

Next, the example shows the *su* command being used to attempt to log in as user NetDev. Notice that although a password was requested, no password will actually work. This is because, by default, when a new user is created, a password entry is created in...
the /etc/shadow file, but until this password is actually set by using the `passwd` command, you cannot log in as the user because the `default` password hash in the shadow file is an invalid hash. The example shows the password being removed altogether with the command `passwd -d NetDev`. Only at this point are you able to log in without getting a password prompt. The password is then set using the command `passwd NetDev`, and a warning is displayed because the password entered was `Cisco123`. Once the password is set, it is possible to log in as the user in question. Note that creating a user also creates a home directory—in this case /home/NetDev—as shown in the output of the `pwd` command. The files /etc/passwd, /etc/group, and /etc/shadow are also updated to reflect the new user details, as shown in the example.

**Example 3-18  Creating a New User and Setting the Password**

```bash
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:26:29 +03 2018 on pts/1
[root@localhost ~]#

! Verify whether the user NetDev exists
[root@localhost ~]# id NetDev
id: NetDev: no such user
[root@localhost ~]#

! Add user NetDev and log in to it
[root@localhost ~]# useradd NetDev
[root@localhost ~]# exit
Logout
[NetProg@localhost ~]$

! Authentication will fail due to invalid "default" hash
[NetProg@localhost ~]$ su NetDev
Password:
`su: Authentication failure`
[NetProg@localhost ~]$

! Switch back to user root and remove the password
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:27:07 +03 2018 on pts/1
[root@localhost ~]# passwd -d NetDev
Removing password for user NetDev.
passwd: Success
[root@localhost ~]# exit
logout
```
Switch to user root and set the password manually then test

```
[NetProg@localhost ~]$
$ su -
Password:
Last login: Sun Apr 15 14:28:12 +03 2018 on pts/1
[root@localhost ~]# passwd NetDev
Changing password for user NetDev.
New password:
BAD PASSWORD: The password fails the dictionary check - it is based on a dictionary word
Retype new password:
```

```
Password: all authentication tokens updated successfully.
```

```
[NetProg@localhost ~]$
$ su -
Password:
Last login: Sun Apr 15 14:50:37 +03 2018 on pts/0
[root@localhost ~]# tail -n 1 /etc/shadow
NetDev:$6$y27JA0id$i8Wze1ShSptxy5w68S8f77f0eAeazo2cayjD/eqikRRKyp2V6etXNNrzqDqXqWeApzMs2zd/..j5fm05PK.kL/17636:99999:7:::
```

Check the home directory and other details for user NetDev

```
[NetDev@localhost ~]$
$ cd
```

```
[NetDev@localhost ~]$
$ id NetDev
uid=1002(NetDev) gid=1003(NetDev) groups=1003(NetDev)
```

```
[NetDev@localhost ~]$
$ tail -n 1 /etc/group
NetDev:x:1003:
```

Switch to user root and check file /etc/shadow

```
[root@localhost ~]# tail -n 1 /etc/shadow
NetDev:365y27JA0id$19Kre1ShoPtxyS668S8f77f0eAeazo2cayjD/eqikRRKyp2V6etXNNrzqDqXqWeApzMs2zd/..j5fm05PK.KL/17636:99999:7:::
```

logout

```
[NetDev@localhost ~]$
A user can change her own password by simply typing `passwd` without any arguments. The user is then prompted to enter the current password and then the new password and then to confirm the new password.

To delete a user, you use the command `userdel {username}`. This command deletes the user from the system; to delete that user's home directory and print spool as well, you use the option `-r` with the command. You use the option `-f` to force the delete action even if the user is still logged in.

You can add groups separately from users by using the command `groupadd {group_name}`. You can use the option `-g` to set the GID manually instead of allowing automatic assignment of the next available GID. You delete groups by using the command `groupdel {group_name}`. Example 3-19 shows how to create a new group called engineers and set its GID to 1111.

**Example 3-19  Creating a New Group engineers**

```
[root@localhost ~]# tail -n 2 /etc/group
NetProg:x:1002:
NetDev:x:1003:
[root@localhost ~]# groupadd -g 1111 engineers
[root@localhost ~]# tail -n 3 /etc/group
NetProg:x:1002:
NetDev:x:1003:
engineers:x:1111:
[root@localhost ~]#
```

To delete a group, you use the command `groupdel {group_name}`. You change a group's details by using the command `groupmod`. The command `groupmod -g {new_gid} {group_name}` changes the group gid to `new_gid`, and the command `groupmod -n {new_name} {old_name}` changes the group's name from `old_name` to `new_name`. Finally, you change the group password by using the command `gpasswd {group_name}`. In Example 3-20, the group engineers is changed to NetDevOps, and its GID is changed to 2222. Then its password is modified to Cisco123.

**Example 3-20  Modifying Group Details**

```
[root@localhost ~]# tail -n -1 /etc/group
engineers:x:1111:
[root@localhost ~]#

! Change the group name to NetDevOps
[root@localhost ~]# groupmod -n NetDevOps engineers
[root@localhost ~]# tail -n -1 /etc/group
NetDevOps:x:1111:
[root@localhost ~]#
```
Chapter 3: Linux Storage, Security, and Networks

A user has one primary group and one or more secondary groups. A user's primary group is the group that the user is placed in when logging in. You modify user group membership by using the command `usermod`. To change a user's primary group, you use the syntax `usermod -g [primary_group] [username]`. To change a user's secondary group, you use the syntax `usermod -G [secondary_group] [username]`; note that this command removes all secondary group memberships for this user and adds the group `secondary_group` specified in the command. To add a user to a secondary group while maintaining his current group memberships, you use the syntax `usermod -aG [new_secondary_group] [username]`. To lock a user account, you use the option `-L` with the `usermod` command, and to unlock an account, you use the `-U` option with this command.

Example 3-21 shows how to change the primary group of user NetDev from NetDev to NetOps and add the wheel group to the list of secondary groups to give the user root privileges through the `sudo` command.

**Example 3-21  Modifying User Details**

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@localhost ~]# id NetDev</td>
<td>uid=1002(NetDev)  gid=1003(NetDev)  groups=1003(NetDev)</td>
</tr>
<tr>
<td>[root@localhost ~]# usermod -g NetOps NetDev</td>
<td></td>
</tr>
<tr>
<td>[root@localhost ~]# id NetDev</td>
<td>uid=1002(NetDev)  gid=2222(NetOps)  groups=2222(NetOps)</td>
</tr>
<tr>
<td>[root@localhost ~]# usermod -aG wheel NetDev</td>
<td></td>
</tr>
<tr>
<td>[root@localhost ~]# id NetDev</td>
<td>uid=1002(NetDev)  gid=2222(NetOps)  groups=2222(NetOps),10(wheel)</td>
</tr>
</tbody>
</table>

Notice that when the `-g` option is used to change the primary group, the secondary group is also changed. This is because user NetDev was only a member of a single group, NetDev, and that group was both the user's primary group and secondary group. When the primary and secondary groups are different, the `-g` option changes only the primary group of the user.
File Security Management

Chapter 2 describes the output of the `ls -l` command and introduces file permissions, also known as the file mode bits. This section builds on that introduction and expands on how to manage access to files and directories by modifying their permissions. It also discusses changing the file owner (user) and group. Keep in mind that in Linux, everything is represented by a file. Therefore, the concepts discussed here have a wider scope than what seems to be obvious. Also, whenever a reference is made to a file, the same concept applies to a directory, unless explicitly stated otherwise.

Example 3-22 shows the output of `ls -l` for the NetProg home directory.

**Example 3-22  Output of the ls -l Command**

```
[NetProg@localhost ~]$ ls -l
total 0
drwxr-xr-x. 2 NetProg NetProg  40 Apr  9 09:41 Desktop
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Documents
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Downloads
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Music
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Pictures
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Public
drwxrwxr-x. 2 NetProg NetProg 183 Apr  7 22:53 Scripts
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Templates
-rw-rw-r--. 1 NetProg NetProg   0 Apr  9 17:51 Testfile.txt
drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Videos
[NetProg@localhost ~]$
```

Here is a quick recap on the file permissions: The very first bit indicates whether this is a file (-), a directory (d), or a link (l). Then the following 3 bits define the permissions for the file owner. By default, the owner is the user who created the file. The following 3 bits define the permissions for the users who are members of the file group. By default, this is the primary group of the user who created the file. The last 3 bits define the permissions for everyone else, referred to as *other*. The letter r stands for read permission, w for write permission, and x for execute permission.

The dot right after the mode bits indicates that this file has an SELinux context. SELinux is a kernel security module that defines the access rights of every user, application, process, and file on the system. SELinux then governs the interactions of these entities using a security policy, where an entity referred to as a subject attempts to access another entity referred to as an object. SELinux is an important component of Linux security but is beyond the scope of this book. When a file or a directory has a + symbol in place of the dot (.), it means the file has an access control list (ACL) applied to it. ACLs, which are covered later in this chapter, provide more granular access control to files on a per-user basis.
The output of the `ls -l` command also displays the file owner (more formally referred to as user) and the file group.

File permissions can be represented (and modified) by either using *symbolic* notation or *octal* notation.

Symbolic notation is the type of notation described so far, where user, group, and others are represented by u, g, and o, respectively, and the access permissions are write, read, and execute, represented by w, r, and x, respectively. The following syntax is used to set the file permissions: `chmod [u=]permissions[g=]permissions[o=]permissions [file_name].`

Example 3-23 shows how to modify the file permissions for file `TestFile.txt` to the following:

- **User**: Read, write, and execute
- **Group**: Read and write
- **Other**: No access

**Example 3-23  Setting File Permissions by Using Symbolic Notation**

```
$ Current file permissions
[NetProg@localhost ~]$ ls -l Testfile.txt
-rw-rw-r--. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$

$ Change the file permissions as listed
[NetProg@localhost ~]$ chmod u=rwx,g=rw,o= Testfile.txt

$ New file permissions
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxrw----. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$
```

Notice that in order to remove all permissions for one of the categories, you just leave the right side of the `=` symbol blank.

One of the challenges with the symbolic notation syntax as used in Example 3-23 is that you have to know beforehand what permissions the file already has and make sure to align the current permissions with the new permissions you are trying to set. For example, if a file already has read and write permissions set for the file group and you would like to add the execute permission, you have to know this fact prior to the change, and then you need to make sure you do not delete the already existing write or read permissions while setting the execute permission. In order to just add or remove permissions for
a specific category, without explicitly knowing or considering the existing permissions, you replace the `=` symbol in the previous syntax with either a `+` or `-` symbol, as follows:

`chmod [u|g|o][+|-][permissions] [file_name]`.

In Example 3-24 the permissions for the file TestFile.txt are modified as follows:

- **User**: Unchanged
- **Group**: Write permission removed and execute permission added
- **Other**: Execute permission added

Notice that when using this syntax, you do not need to know what permissions the file already has. You only need to consider the changes that you want to implement.

**Example 3-24**  Adding and Removing File Permissions by Using Symbolic Notation

```
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxrw----. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$ chmod g-w,g+x,o+x Testfile.txt
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxr-x--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
```

Notice that you can mix the `+` and `-` symbols in the same command and for the same category, as shown in Example 3-24 for the file group, where `g-w` is used to remove the write permission for the group, and `g+x` is used to add the execute permission for the group.

When a certain permission has to be granted or revoked from all categories, the letter `a` is used to collectively mean `u`, `g`, and `o`. The letter `a` in this case stands for *all*. The letter `a` may be dropped altogether, and the command then applies to all categories. For example, the command `chmod +w Example.py` adds the write permission to all categories for the file `Example.py`.

Octal notation, on the other hand, uses the following syntax: `chmod [user_permission] [group_permission] [other_permission] [file_name]`. The `user`, `group`, and `other` categories are represented by their positions in the command. The permission granted to each category is represented as a numeric value that is equal to the summation of each permission's individual value. To elaborate, note the following permission values:

- **Read**=4
- **Write**=2
- **Execute**=1
To set the read permission only, you need to use the value 4; for write permission only, you use the value 2; and for execute permission only, you use the value 1. To set all permissions for any category, you need to use 4+2+1=7. To set the read and write permissions only, you need to use 4+2=6, and so forth. Example 3-25 illustrates this concept and uses octal notation to set the read, write, and execute permissions for both user and group, and set only the execute permission for the category other for file Testfile.txt.

**Example 3-25  Setting File Permissions Using Octal Notation**

```
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxr-x--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
[NetProg@localhost ~]$ chmod 771 Testfile.txt
[NetProg@localhost ~]$ ls -l Testfile.txt
-rwxrwx--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt
```

The number 7 in each of the first two positions in the command `chmod 771 Testfile.txt` represents the sum of 4, 2, and 1 and is used to set all permissions for user and group. The number 1 in the last position sets the execute only permission for other.

While octal notation looks snappier than symbolic notation, it does not provide the option of adding or removing permissions without considering the existing file permissions, as provided by the + and - symbols used with symbolic notation.

Besides modifying file and directory permissions, you can control access to a file or directory by changing the file's user and/or group through the `chown` command. The command syntax is `chown {user}:{group} {file}`. Example 3-26 shows how to change the `user` and `group` of file TestFile.txt to NetDev and networks, respectively.

**Example 3-26  Changing File User and Group by Using the chown Command**

```
[root@localhost ~]# ls -l /home/NetProg/Testfile.txt
-rwxrwx-x. 1 NetDev networks 0 Apr  9 17:51 /home/NetProg/Testfile.txt
[root@localhost ~]# chown NetDev:networks /home/NetProg/Testfile.txt
[root@localhost ~]# ls -l /home/NetProg/Testfile.txt
-rwxrwx-x. 1 NetDev networks 0 Apr  9 17:51 /home/NetProg/Testfile.txt
```

You use the -R option (which stands for recursive) with both the `chmod` and the `chown` commands if the operation is being performed on a directory, and you want the changes to also be made to all subdirectories and files in that directory.

By default, any file or directory created by a user is assigned to the primary group of that user. For example, if user NetDev is in the NetOps group, any file created by user NetDev has NetDev as the file user and NetOps as the file group. You can change this
default behavior by either using the sg command when creating the file or by logging in to another group by using the command newgrp. If that other group is one of the user's secondary groups, no password is required. If that other group is not one of the user's secondary groups, the user is prompted for a password.

Example 3-27 shows the default behavior when creating a file. In this case, a new file named NewFile is created by user NetDev. As expected, the file user is NetDev, and the file group is NetOps.

**Example 3-27**  Default User and Group of a Newly Created File

```
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)
[NetDev@localhost ~]$ touch NewFile
[NetDev@localhost ~]$ ls -l NewFile
-rw-r--r--. 1 NetDev NetOps 0 Apr 17 00:59 NewFile
[NetDev@localhost ~]$
```

Example 3-28 shows how to use the sg command to create file NewFile_1 but under the group networks.

**Example 3-28**  Using the sg Command to Create a File Under a Different Group

```
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)
[NetDev@localhost ~]$ sg networks 'touch NewFile_1'
Password:
[NetDev@localhost ~]$ ls -l NewFile_1
-rw-r--r--. 1 NetDev networks 0 Apr 17 01:03 NewFile_1
[NetDev@localhost ~]$
```

Notice that the command `touch [file_name]`, which itself is an argument to the sg command, has to be enclosed in quotes because it is a multi-word command. Notice also that because the user NetDev is not a member in the networks group, as you can see from the output of the id command, the user is prompted for the group password, which was set earlier by using the command gpasswd networks.

Alternatively, the user can log in to another group by using the command newgrp and create a file or directory under that group. Example 3-29 shows the user NetProg logging in to group systems and not being prompted for a password since this is one of NetProg's secondary groups. When the file NewFile_2 is created, the user of the file is NetProg, and the group is systems, not NetProg.

```
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)
[NetDev@localhost ~]$ newgrp systems
Password:
[NetDev@localhost ~]$ touch NewFile_2
[NetDev@localhost ~]$ id
uid=1003(NetProg) gid=0(wheel) groups=0(wheel),20(groups),10(wheel)
[NetDev@localhost ~]$ ls -l NewFile_2
-rw-r--r--. 1 NetDev NetOps 0 Apr 17 01:03 NewFile_2
[NetDev@localhost ~]$`

Alternatively, the user can log in to another group by using the command newgrp and create a file or directory under that group. Example 3-29 shows the user NetProg logging in to group systems and not being prompted for a password since this is one of NetProg's secondary groups. When the file NewFile_2 is created, the user of the file is NetProg, and the group is systems, not NetProg.
Example 3-29  Using the newgrp Command to Log In to a Different Group

```
[NetProg@localhost ~]$ id NetProg
uid=1001(NetProg) gid=1002(NetProg) groups=1002(NetProg),10(wheel),2224(systems)
[NetProg@localhost ~]$ newgrp systems
[NetProg@localhost ~]$ touch NewFile_2
[NetProg@localhost ~]$ ls -l NewFile_2
-rw-r--r--. 1 NetProg systems 0 Apr 17 01:15 NewFile_2
[NetProg@localhost ~]$
```

Access Control Lists

So far in this chapter, you have seen how to set file and directory access permissions for either user, or collectively for group, or other. What if you want to set those permissions individually for a specific user who is not the file owner or for a group of users who belong to a group other than the file group? File mode bits do not help in such situations. Using the file mode bits, the only user whose permissions can be changed individually is the file or directory owner (user) and the only group of users whose permissions can be changed collectively are the users who are members of the file or directory group.

Access control lists (ACLs) provide more granular control over file and directory access. ACLs allow a system administrator (or any other user who has root privileges) to set file and directory permissions for any user or group on the system.

Before you can configure ACLs, three prerequisites must be met:

- The kernel must support ACLs for the file system type on which ACLs will be applied.
- The file system on which ACLs will be used must be mounted with the ACL option.
- The ACL package must be installed.

Most common distros today—including CentOS 7 and Red Hat Enterprise Linux (RHEL) 7 and later versions—have these prerequisites configured by default, and you do not need to do any further configuration.

If you are running a different distro or an older version of CentOS, you can check the first prerequisite by using either the `findmnt` or `blkid` command to determine the file system type on your system. The command `findmnt` works only if the file system has been mounted, and `blkid` works whether it is mounted or not. Then you need to inspect the kernel configuration file `/boot/conf-<version.architecture>` to determine whether ACLs have been enabled for this file system type. Example 3-30 shows the relevant output for the file system on the sda1 partition.
Example 3-30  ACL Support for the sda1 File System

```
[&root@server1 ~]# findmnt /dev/sda1
TARGET SOURCE    FSTYPE OPTIONS
/boot  /dev/sda1 xfs    rw,relatime,seclabel,attr2,inode64,noquota
[&root@server1 ~]# cat /boot/config-3.10.0-693.el7.x86_64 | grep ACL
CONFIG_EXT4_FS_POSIX_ACL=y
CONFIG_XFS_POSIX_ACL=y
CONFIG_BTRFS_FS_POSIX_ACL=y
CONFIG_FSS_POSIX_ACL=y
CONFIG_GENERIC_ACL=y
CONFIG_TMPFS_POSIX_ACL=y
CONFIG_NFS_V3_ACL=y
CONFIG_NFSD_V2_ACL=y
CONFIG_NFSD_V3_ACL=y
CONFIG_NFS_ACL_SUPPORT=m
CONFIG_CEPH_FS_POSIX_ACL=y
CONFIG_CIFS_ACL=y
[&root@server1 ~]#
```

The kernel configuration file lists different configuration options, each followed by an = symbol and then the letter y, n, or m. The letter y means that this option (module) was configured as part of the kernel when the kernel was first compiled. In this example, CONFIG_XFS_POSIX_ACL=y means that the kernel supports ACLs for the xfs file system. The letter n indicates that this module was not compiled into the kernel, and the letter m means that this module was compiled as a loadable kernel module (introduced in Chapter 2).

The second prerequisite is that the partition on which the ACLs will be used has to be mounted with the ACL option. By default, on ext3/4 and xfs file systems, ACL support is enabled. In older CentOS versions and other distros where the ACL option is not enabled by default, the file system can be mounted with the ACL option by using the syntax `mount -o acl [partition] [mount_point]`. On the other hand, if the ACL option is enabled by default, and you want to disable ACL support while mounting the file system, you can use the `noacl` option with the `mount` command. As discussed in the previous section, mounting using the `mount` command is non-persistent. For persistent mounting with the ACL option, you can add an entry to the `/dev/fstab` file (or amend an existing entry) and add the `acl` option (right after the `defaults` keyword). The `/dev/fstab` file is discussed in detail earlier in this chapter.

Finally, by using the `yum info acl` command, you can confirm whether the ACL package has been installed. The `yum` command is covered in detail in Chapter 2.

When ACL support has been established, you can use the command `getfacl [filename|directory]` to display the ACL configuration for a file or directory. Example 3-31 shows the output of the `getfacl` command for the directory `/Programming` and then for the file `NewFile.txt`. 
Example 3-31  Output of the getfacl Command

As you can see from the output in Example 3-31, both the directory and file are owned by the user root, and the group of both is also root. So far, there is no additional information provided by the getfacl command beyond what is already displayed by ls -l; the format is the only difference.

For the file NewFile.txt, the user NetProg is not the file owner and is not a member of the file group. As per the permissions for other, the user NetProg should be able to only read the file but not write to it or execute it. In Example 3-32, the user NetProg attempts to write to the file NewFile.txt by using the echo command, but a “Permission denied” error message is displayed. The setfacl -m u:NetProg:rw /Programming/NewFile.txt command grants write permission to the user NetProg. When the write operation is attempted again, it is successful due to the new elevated permissions.

Example 3-32  Changing the Permissions for the User NetProg by Using setfacl
Notice the + symbol that now replaces the dot to the right of file permission bits at the end of Example 3-32. This indicates that an ACL has been applied to this file. The new write permission has been granted to the user NetProg only, and not to any other user. This was done without amending the file permissions for the user, group, or other categories. It was also done without modifying the group memberships of the user NetProg. The same permission could also be applied to a group instead of an individual user. The level of granularity provided by ACLs should be clear by now.

The `setfacl` command used in Example 3-32 was issued with the option `-m`, which is short for modify and is used to apply a new ACL or modify an existing ACL. To remove an ACL, you use the option `-x` instead of `-m`; the remainder of the command remains the same, except that the ACL in the command is an existing ACL that is now being removed.

In Example 3-32 you can see the three-field argument `u:NetProg:rw`. When setting an ACL for a user, the first field is `u`, as in the example. For a group, the first field would be `g`, and for other, the first field would be `o`. The second field is the user or group name, which is NetProg in this example. If the ACL is for other, this field remains empty. The third field is the permissions you wish to grant to the user or group.

Finally, after the three-field argument is the name of the directory or file to which the ACL is applied. Note that whether a full path or only a relative path is required depends on the current working directory relative to the location of the file or directory to which the ACL is being applied. The same rules apply here as with any other Linux command that operates on a file or directory.
Therefore, the general syntax of the `setfacl` command to add, modify, or remove an ACL is:

```
setfacl [-m|-x] [uglo]:[username/group]:[permissions] [file/directory]
```

To remove all ACL entries applied to a file, you use the option `-b` followed by the filename, omitting the three-field argument.

In Example 3-32, notice the text mask::rw- in the output of the `getfacl` command, after the ACL has been applied. The mask provides one more level of control over the permissions granted by the ACL. Say that after granting several users different permissions to a file, you decide to remove a specific permission, such as the write permission, from all named users. The ACL mask then comes in handy. The permissions in the mask override the permissions for all named users and the file group. For example, if the mask permissions are r-x and the user NetProg has been granted rwx permissions, that user’s effective permissions are r-x after the mask is set. The effective mask permissions are applied using the command:

```
setfacl -m m:[permissions] [filename]
```

In Example 3-33, the user NetProg has permissions rw-, and so does the mask. The mask is modified to r--. Notice the effective permissions that appear on the right side of the output of the `getfacl` command after the mask has been modified. After you remove the write permission from the mask, NetProg’s write attempt to the file fails.

**Example 3-33  Changing the Mask Permissions by Using `setfacl`**

```
# Set the effective rights mask
[root@localhost /]# setfacl -m m:r /Programming/NewFile.txt
[root@localhost /]# getfacl /Programming/NewFile.txt
getfacl: Removing leading '/' from absolute path names
# file: Programming/NewFile.txt
  # owner: root
  # group: root
  user::rw-
  user:NetProg:rw-        #effective:r--
  group::r--
  mask::r--
  other::r--
```

```
# Write operation to file by user NetProg now fails
[root@localhost /]# su NetProg
[NetProg@localhost /]# echo "Testing mask permissions" > /Programming/NewFile.txt
bash: /Programming/NewFile.txt: Permission denied
```

When ACLs are applied to directories, by default, these ACLs are not inherited by files and subdirectories in that directory. In order to achieve inheritance, the option `-R` has to
be added to the same `setfacl` command used earlier. In Example 3-34, an ACL setting rwx permissions for the user NetProg is applied to the directory Programming. Attempting to write to file NewFile.txt under the directory by user NetProg fails because the write permission has not been inherited by the file.

**Example 3-34**  **ACLs Are Not Inherited by Default by Subdirectories and Files Under a Directory**

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setfacl -m u:NetProg:rwx /Programming</code></td>
<td></td>
</tr>
<tr>
<td><code>getfacl /Programming</code></td>
<td></td>
</tr>
<tr>
<td><code>getfacl: Removing leading '/' from absolute path names</code></td>
<td></td>
</tr>
<tr>
<td><code># file: Programming</code></td>
<td></td>
</tr>
<tr>
<td><code># owner: root</code></td>
<td></td>
</tr>
<tr>
<td><code># group: root</code></td>
<td></td>
</tr>
<tr>
<td><code>user::rwx</code></td>
<td></td>
</tr>
<tr>
<td><code>user:NetProg:rwx</code></td>
<td></td>
</tr>
<tr>
<td><code>group::r-x</code></td>
<td></td>
</tr>
<tr>
<td><code>mask::rwx</code></td>
<td></td>
</tr>
<tr>
<td><code>other::r-x</code></td>
<td></td>
</tr>
<tr>
<td>! The acl is not applied to NewFile.txt under the directory</td>
<td></td>
</tr>
<tr>
<td><code>getfacl /Programming/NewFile.txt</code></td>
<td></td>
</tr>
<tr>
<td><code>getfacl: Removing leading '/' from absolute path names</code></td>
<td></td>
</tr>
<tr>
<td><code># file: Programming/NewFile.txt</code></td>
<td></td>
</tr>
<tr>
<td><code># owner: root</code></td>
<td></td>
</tr>
<tr>
<td><code># group: root</code></td>
<td></td>
</tr>
<tr>
<td><code>user::rw-</code></td>
<td></td>
</tr>
<tr>
<td><code>group::r-</code></td>
<td></td>
</tr>
<tr>
<td><code>other::r-</code></td>
<td></td>
</tr>
<tr>
<td>! And the write operation fails as expected</td>
<td></td>
</tr>
<tr>
<td><code>su - NetProg</code></td>
<td></td>
</tr>
<tr>
<td><code>echo &quot;This is a write test&quot; &gt; /Programming/NewFile.txt</code></td>
<td></td>
</tr>
<tr>
<td><code>bash: /Programming/NewFile.txt: Permission denied</code></td>
<td></td>
</tr>
</tbody>
</table>

After the ACL has been removed and then reapplied in Example 3-35 using the `-R` option, the user NetProg can write to the file successfully. The `getfacl` command also shows that the ACL has been applied to the file as if the `setfacl` command had been applied to the file directly.
Example 3-35  ACL Inheritance by Subdirectories and Files Under a Directory Using the -R Option

<table>
<thead>
<tr>
<th>Clear the acl from the /Programming directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@localhost ~]# setfacl -b /Programming</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Apply the acl to directory /Programming using the -R option</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@localhost ~]# setfacl -R -m u:NetProg:rwx /Programming</td>
</tr>
<tr>
<td>[root@localhost ~]# getfacl /Programming</td>
</tr>
<tr>
<td>getfacl: Removing leading '/' from absolute path names</td>
</tr>
<tr>
<td># file: Programming</td>
</tr>
<tr>
<td># owner: root</td>
</tr>
<tr>
<td># group: root</td>
</tr>
<tr>
<td>user::rwx</td>
</tr>
<tr>
<td>user:NetProg:rwx</td>
</tr>
<tr>
<td>group::r-x</td>
</tr>
<tr>
<td>mask::rwx</td>
</tr>
<tr>
<td>other::r-x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The acl is inherited by the file NewFile.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@localhost ~]# getfacl /Programming/NewFile.txt</td>
</tr>
<tr>
<td>getfacl: Removing leading '/' from absolute path names</td>
</tr>
<tr>
<td># file: Programming/NewFile.txt</td>
</tr>
<tr>
<td># owner: root</td>
</tr>
<tr>
<td># group: root</td>
</tr>
<tr>
<td>user::rw-</td>
</tr>
<tr>
<td>user:NetProg:rwx</td>
</tr>
<tr>
<td>group::r--</td>
</tr>
<tr>
<td>mask::rw-</td>
</tr>
<tr>
<td>other::r--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>And the write operation is successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@localhost ~]# su - NetProg</td>
</tr>
<tr>
<td>[NetProg@localhost ~]$ echo &quot;This is to test inheritance&quot; &gt; /Programming/NewFile.txt</td>
</tr>
<tr>
<td>[NetProg@localhost ~]$ cat /Programming/NewFile.txt</td>
</tr>
<tr>
<td>This is to test inheritance</td>
</tr>
</tbody>
</table>

It is important to remember that the ACL applied to a directory and inherited by all subdirectories and files will not be applied to any files created after the ACL has been applied. Only the files that existed before the ACL was applied will be affected.

The ACLs described so far are called access ACLs. Another type of ACLs, called default ACLs, may be used with directories (only) if the requirement is that all files and subdirectories, when created, should inherit the parent directory ACLs. The syntax for applying a default ACL is `setfacl -m d:u|g|o:[username|group]:[permissions] {directory}`. Try to
experiment with default ACLs and note how newly created files inherit the directory ACL without your having to explicitly issue the `setfacl` command after the file or subdirectory has been created.

The same concepts discussed previously for a single user apply to a group when you set the ACL for a group of users other than the file or directory group by using the letter `g` along with the group name in the `setfacl` command instead of a `u` with the username.

In addition to using the `setfacl` command to set permissions for a specific user or group, you can use this command to set permissions for the file user, group, or other categories, similar to what can be accomplished using the `chmod` command as shown in the previous section. Note that if the `setfacl` command is used to apply an ACL to a file or directory, it is recommended that you *not* use `chmod`.

When a file or directory is copied or moved, ACLs are moved along with the file or directory.

**Linux System Security**

CentOS 7 and later versions come with a default built-in firewall service named `firewalld`. This service functions in a similar manner to a regular firewall in terms of providing *security zones* with different *trust levels*. Each zone constitutes a group of permit/deny rules for incoming traffic. Each physical interface on the server is bound to one of the firewall zones. However, `firewalld` provides only a subset of the services provided by a full-fledged firewall.

You can check the status of the `firewalld` service and start, stop, enable, and disable the service just as you would any other service on Linux by using the `systemctl` command. Example 3-36 shows the status of the `firewalld` service: In this example, you can see that it is active and enabled.

**Example 3-36 The `firewalld` Service Status**

```
[NetProg@localhost ~]$ systemctl status firewalld
● firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled; vendor preset: enabled)
   Active: active (running) since Sat 2018-04-21 21:37:06 +03; 30min ago
     Docs: man:firewalld(1)
   Main PID: 787 (firewalld)
   CGroup: /system.slice/firewalld.service
           └─787 /usr/bin/python -Es /usr/sbin/firewalld --nofork --nopid

Apr 21 21:37:05 localhost.localdomain systemd[1]: firewall-daemon: Starting firewall-daemon...
Apr 21 21:37:06 localhost.localdomain systemd[1]: firewall-daemon: Starting firewall-daemon...

--------- OUTPUT TRUNCATED FOR BREVITY ---------
```
The `firewalld` service has a set of zones created by default when the service is first installed; these zones are sometimes referred to as the *base* or *predefined* zones. Custom zones can also be created and deleted. However, base zones cannot be deleted. One zone is designated as the default zone and is the zone to which all interfaces are bound, by default, unless the interface is explicitly moved to another zone. By default, the default zone is the public zone. Each zone has a set of rules attached to it and a list of interfaces bound to it. Rules and interfaces can be added to or removed from a zone.

Example 3-37 shows how to list the base zones of `firewalld` by using the command `firewall-cmd --get-zones` and how to identify the default zone by using the command `firewall-cmd --get-default-zone`.

**Example 3-37  Listing the Base and Default Zones of a Firewall**

```
[root@localhost ~]# firewall-cmd --get-zones
block dmz drop external home internal public trusted work
[root@localhost ~]# firewall-cmd --get-default-zone
public
[root@localhost ~]#
```

You can change the default zone by using the command `firewall-cmd --set-default-zone={zone_name}`.

You can list the details of a zone by using the command `firewall-cmd --list-all --zone={zone_name}`, as shown in Example 3-38. To list the details of the default zone, you omit the `--zone={zone_name}` option.

**Example 3-38  Listing Zone Details**

```
[root@localhost ~]# firewall-cmd --list-all --zone=internal
internal
  target: default
  icmp-block-inversion: no
  interfaces:
  sources:
  services: ssh mdns samba-client dhcpv6-client
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:

[root@localhost ~]# firewall-cmd --list-all
public (active)
```

Example 3-39 shows how to add rules to the zone dmz to permit specific incoming traffic on interfaces bound to this zone. The first rule added permits traffic from the source IP address 10.10.1.0/24 by using a source-based rule. Then BGP traffic on TCP port 179 is permitted by using a port-based rule. HTTP service is then permitted by defining a service-based rule. Finally, interface enp0s9 is removed from the public zone and bound to the dmz zone. Notice how the rules appear when the details of the zone are listed at the end of the example.

Example 3-39  Adding Rules to Zone dmz

```
[root@localhost ~]# firewall-cmd --list-all --zone=dmz

success
```

```
[root@localhost ~]# firewall-cmd --zone=dmz --add-source=10.10.1.0/24

success
```

```
[root@localhost ~]# firewall-cmd --zone=dmz --add-port=179/tcp

success
```
```bash
[root@localhost ~]# firewall-cmd --zone=dmz --add-service=http
success
[root@localhost ~]# firewall-cmd --zone=dmz --add-interface=enp0s9
The interface is under control of NetworkManager, setting zone to 'dmz'.
success
[root@localhost ~]# firewall-cmd --zone=dmz --list-all
```

Note that in order to remove a rule, instead of using the `--add` option, you use the `--remove` option. For example, to remove the rule for TCP port 179, you use the command `firewall-cmd --zone=dmz --remove-port=179/tcp`.

Much like running and startup configurations on routers and switches, `firewalld` supports both runtime and permanent configurations. A runtime configuration is not persistent and is lost after a reload. A permanent configuration is persistent but takes effect only after a reload when the configuration has been changed. Any configuration commands that have been executed are reflected in the runtime configuration. To make a configuration permanent, you use the option `--permanent` with the command. You reload the `firewalld` service by using the command `firewall-cmd --reload`.

**Linux Networking**

Linux provides several methods for managing network devices and interfaces on a system. Usually, a system administrator can accomplish the same task using several different methods. A network device or an interface is managed by the kernel, and each method accesses the Linux kernel via a different path. There are three popular methods for managing Linux networking:

- Using the command-line `ip` utility
- Using the NetworkManager service
- Using network configuration files
This section covers these three methods listed. It should be fairly easy to use the help resources on your Linux distro, such as the man and info pages, to learn about any utility not covered here.

**Note** Keep in mind that some commands and utilities for managing Linux networking, such as `ifconfig`, `netstat`, `arp`, and `route`, are considered legacy utilities. These utilities have not been updated for years and have been deprecated on some distros but are still available on others. Even if any of these commands are available in the distro you are using, we do not recommend using them; instead, use the methods described in this section. Basically, the way legacy utilities function, particularly how these utilities speak with the kernel, is not very efficient. You will probably run into these legacy utilities at some point while working on Linux. For example, at the time of this writing, all four legacy utilities mentioned here are still supported on the Bash shells exposed by IOS XR and NX-OS.

### The ip Utility

`ip` is a command-line utility that is part of the `iproute2` group of utilities. It is invoked using the command `ip [options] [object] [action]`. This syntax is quite intuitive in that the **action** in the command indicates what action you would want to apply to an **object**. For example, the command `ip link show` applies the action **show** to the object **link**. As you may have guessed, this command displays the state of all network interfaces (links) on the system, as shown in Example 3-40. To limit the output to one specific interface, you can add `dev {intf}` to the end of the command, as also shown in the example.

**Note** The man pages for the `ip` command refer to the **action** part in the previous syntax as **command**. We took the liberty to call it **action** in the upcoming few paragraphs in order to avoid the obvious confusion that will result from calling it **command**.

#### Example 3-40  Output of the Command ip link show

```
[NetProg@localhost ~]$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
     link/ether 08:00:27:a7:32:f7 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
     link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
     link/ether 08:00:27:b4:ce:55 brd ff:ff:ff:ff:ff:ff
```
Table 3-1 lists some of the objects that are commonly used with the ip command.

**Table 3-1  Objects That Are Commonly Used with the ip Command**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>IPv4 or IPv6 protocol address</td>
</tr>
<tr>
<td>link</td>
<td>Network interface</td>
</tr>
<tr>
<td>route</td>
<td>Routing table entry</td>
</tr>
<tr>
<td>maddress</td>
<td>Multicast address</td>
</tr>
<tr>
<td>neigh</td>
<td>ARP entry</td>
</tr>
</tbody>
</table>

As of this writing, there are 19 objects that can be acted upon by using the ip command. A full list of objects can be found in the man pages for the ip command. Objects can be written in full or in abbreviated form, such as address or addr. The actions that can be used with the ip command are limited to three options listed in Table 3-2.

**Table 3-2  Actions That Can Be Used with the ip Command**

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Adds the object</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes the object</td>
</tr>
<tr>
<td>show (or list)</td>
<td>Displays information about the object</td>
</tr>
</tbody>
</table>
The keyword `show` or `list` can be dropped from a command, and the command will still be interpreted as a `show` action. For example, the command `ip link show` is equivalent to just `ip link`.

The `ip addr` command lists all interfaces on the system, each with its IP address information, and the `ip maddr` command displays the multicast information for each and every interface. The `ip neigh` command displays the ARP table. The ARP table consists of a list of neighbors on each interface on the local network. The examples in this section show how to use these `show` commands.

You can bring an interface on Linux up or down by using the command `ip link set {intf} {up|down}`. The `set` action is only applicable to the link object and therefore was not listed in Table 3-2. Example 3-41 shows how to bring interface enp0s8 down and then up again. Note that changing networking configuration on Linux, including toggling an interface’s state, requires root privileges. The `show` commands, however, do not. To keep Example 3-41 short and avoid the frequent password prompt, all commands in the example are issued by the root user. However, running commands as root in general is not a recommended practice. On a production network, make sure to avoid logging in as root. It is best practice to log in with your regular user account and use the `sudo` command whenever a command requires root privileges to execute, as explained in Chapter 2.

Example 3-41  Toggling Interface State

```bash
[<root@localhost ~]> ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
   link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
[<root@localhost ~]> ip link set enp0s8 down
[<root@localhost ~]> ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN mode DEFAULT qlen 1000
   link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
[<root@localhost ~]> ip link set enp0s8 up
[<root@localhost ~]> ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
   link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
[<root@localhost ~]#
```

You can add an IP address to an interface by using the command `ip addr add {IP_address} dev {intf}`. By replacing the action `add` with `del`, you remove the IP address. In Example 3-42, IP address 10.1.0.10/24 is added to interface enp0s8, and then the original IP address, 10.1.0.1/24, is removed. The `ip addr show enp0s8` command is used to inspect the interface IP address before and after the change.
Example 3-42  Adding and Removing IP Addresses on Interfaces

Notice that the IP address 10.1.0.10/24 is added as a secondary address, as long as another IP address is configured on the interface. When the original IP address is removed, the new IP address becomes the primary address.

Notice the mtu value in the output of the `ip addr show` command in Example 3-42. By default the mtu is set to 1500 bytes. To change that value, you use the command `ip link set {intf} mtu {mtu_value}`.

A very useful feature that any network engineer would truly appreciate is interface promiscuous mode. By default, when an Ethernet frame is received on an interface, that frame is passed on to the upper layers for processing only if the destination MAC address of the frame matches the MAC address of the interface (or if the destination MAC address is a broadcast address). If the MAC addresses do not match, the frame is ignored. This renders packet sniffing applications such as Wireshark and features such as port mirroring unusable. In promiscuous mode, an interface accepts any and all incoming
packets, whether the packets are addressed to that interface or not. You can enable promiscuous mode by using the command `ip link set {interface} promisc on`.

In the routing table, the list of routes on the system can be displayed by using the command `ip route`. Example 3-43 shows that the routing table is empty when no IP addresses are configured on any of the interfaces. When the IP address 10.2.0.30/24 is configured on interface enp0s3, one entry, corresponding to that interface, is added to the routing table.

**Example 3-43  Viewing a Routing Table by Using the `ip route` Command**

```
[NetProg@server4 ~]$
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:2c:61:d0 brd ff:ff:ff:ff:ff:ff
    inet 10.2.0.30/24 scope global enp0s3
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe2c:61d0/64 scope link
        valid_lft forever preferred_lft forever
[NetProg@server4 ~]$
```

Routing tables on Linux systems are very similar to routing tables on routers. In fact, a Linux server could easily function as a router. In order to display routing table functionality in Linux, server1 in the topology in Figure 3-2 is used as a router to route traffic between server2 and server3. server2 is connected to network 10.1.0.0/24, and server3 is connected to network 10.2.0.0/24. All three servers are configured such that server1 routes between the two networks, and eventually server2 should be able to ping server3.
IP addressing needs to be configured first. server1 is configured with IP addresses ending with .10, server2 with an IP address ending in .20, and server3 with an IP address ending in .30, as shown in Example 3-44.

**Example 3-44  Configuring IP Addresses on the Interfaces Connecting The Three Servers**

```bash
# server1
[root@server1 ~]# ip addr add 10.1.0.10/24 dev enp0s8
[root@server1 ~]# ip addr add 10.2.0.10/24 dev enp0s9
[root@server1 ~]# ip addr show enp0s8 | grep "inet "
    inet 10.1.0.10/24 scope global enp0s8
[root@server1 ~]# ip addr show enp0s9 | grep "inet "
    inet 10.2.0.10/24 scope global enp0s9

# server2
[root@server2 ~]# ip addr add 10.1.0.20/24 dev enp0s3
[root@server2 ~]# ip addr show enp0s3 | grep "inet "
    inet 10.1.0.20/24 scope global enp0s3

# server3
[root@server3 ~]# ip addr add 10.2.0.30/24 dev enp0s3
[root@server3 ~]# ip addr show enp0s3 | grep "inet "
    inet 10.2.0.30/24 scope global enp0s3
```

A ping to the directly connected server is successful on all three servers. However, when server2 attempts to ping server3, the ping fails, as shown in Example 3-45.
Example 3-45   Pinging the Directly Connected Interfaces Is Successful but Pinging server3 From server2 Is Not

![server2 to server1](root@server2 ~) # ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.796 ms
--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.796/0.796/0.796/0.000 ms
[root@server2 ~]#

![server3 to server1](root@server3 ~) # ping -c 1 10.2.0.10
PING 10.2.0.10 (10.2.0.10) 56(84) bytes of data.
64 bytes from 10.2.0.10: icmp_seq=1 ttl=64 time=1.139 ms
--- 10.2.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.139/1.139/1.139/0.000 ms
[root@server3 ~]#

![Pinging server2 to server3 and vice versa is not successful](root@server2 ~) # ping 10.2.0.10
connect: Network is unreachable
[root@server2 ~] # ping 10.2.0.30
connect: Network is unreachable
[root@server2 ~]#

![Pinging server3 to subnet 10.1.0.0/24](root@server3 ~) # ping -c 1 10.1.0.10
connect: Network is unreachable
[root@server3 ~] # ping -c 1 10.1.0.20
connect: Network is unreachable
[root@server3 ~]#

You are probably very familiar with the ping command. ping works on Linux exactly as it does on network devices: by sending one or more ICMP packets to the destination and either receiving an ICMP reply if the ping is successful (one reply per packet sent).
or receiving an ICMP unreachable packet or no response at all if the ping is not. The command in Example 3-45 uses the -c 1 option to send a single ICMP packet, which is enough to test the reachability of the destination.

Example 3-46 shows how to use the command `ip route add 10.2.0.0/24 via 10.1.0.10` on server2 and the command `ip route add 10.1.0.0/24 via 10.2.0.10` on server3 to add routes to the routing tables of each server. The general syntax for adding a route to the routing table is `ip route add [destination]/[mask] via [nexthop]`. The routes instruct each server to use server1 as the next hop to reach the remote network. After the routes are added, server2 and server3 are able to ping server1’s interface on the remote network, but they are still not able to ping each other.

**Example 3-46**  Adding Routing Table Entries for Remote Subnets on server2 and server3. server2 and server3 Can Ping the Remote Subnets on server1, But Still Cannot Ping Each Other

```
server2
[root@server2 ~]# ip route add 10.2.0.0/24 via 10.1.0.10
[root@server2 ~]# ip route add 10.1.0.0/24 via 10.2.0.10
[10.2.0.0/24 via 10.1.0.10 dev enp0s3]
[10.1.0.0/24 dev enp0s3]
[root@server2 ~]# ping -c 1 10.2.0.10
PING 10.2.0.10 (10.2.0.10) 56(84) bytes of data.
64 bytes from 10.2.0.10: icmp_seq=1 ttl=64 time=0.822 ms
--- 10.2.0.10 ping statistics ---
 1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.822/0.822/0.822/0.000 ms
[root@server2 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
--- 10.1.0.10 ping statistics ---
 1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

```
server3
[root@server3 ~]# ip route add 10.1.0.0/24 via 10.2.0.10
[root@server3 ~]# ip route add 10.2.0.0/24 via 10.1.0.10
[10.2.0.0/24 dev enp0s3]
[10.1.0.0/24 dev enp0s3]
[root@server3 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.865 ms
--- 10.1.0.10 ping statistics ---
 1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.865/0.865/0.865/0.000 ms
[root@server3 ~]# ping -c 1 10.1.0.20
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
--- 10.1.0.20 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
[root@server3 ~]#

Forwarding between the interfaces on server1 is disabled by default for security reasons. Therefore, the remaining step is to enable forwarding in the kernel of server1 by toggling the default value of 0 in file /proc/sys/net/ipv4/ip_forward to 1 by using either the command `echo 1 > /proc/sys/net/ipv4/ip_forward` or the command `/sbin/sysctl -w net.ipv4.ip_forward=1`. After either command is used, forwarding is enabled, and both servers can ping each other successfully, as shown in Example 3-47.

**Example 3-47  Enabling Routing on Server1 Resulting in Successful ping Between server2 and server3**

! Enabling routing on server1
[root@server1 ~]# echo 1 > /proc/sys/net/ipv4/ip_forward
[root@server1 ~]# cat /proc/sys/net/ipv4/ip_forward
1

! server2 to server3 ping is successful
[root@server2 ~]# ping -c 1 10.2.0.30
PING 10.2.0.30 (10.2.0.30) 56(84) bytes of data.
64 bytes from 10.2.0.30: icmp_seq=1 ttl=63 time=0.953 ms
--- 10.2.0.30 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.953/0.953/0.953/0.000 ms
[root@server2 ~]#

! server3 to server2 ping is successful
[root@server3 ~]# ping -c 1 10.1.0.20
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=63 time=1.39 ms
--- 10.1.0.20 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.394/1.394/1.394/0.000 ms
[root@server3 ~]#
Note that two commands to achieve the same result are mentioned here. The first method gets the job done by editing a file, and the second gets the same job done by using the command `sysctl`. Which one you should use depends on several factors, the first of which is personal preference. Another issue is whether you know which file in the `/proc/sys` directory contains the kernel setting (sometimes referred to as a `kernel tunable`) that you need to change. If you do not know the file, you can simply use the `sysctl` command to target the parameter directly, regardless of where it is located. You can list all kernel tunables by using the command `/sbin/sysctl -a.

Note You in this case does not necessarily have to literally mean you. It may refer to the automation script or tool that you are using to get the job done. Using a particular tool to amend a file may be more efficient than issuing a command; the reverse may be the case for another tool. Always choose the method that is most efficient and effective for your specific environment.

To remove a routing table entry, you use the syntax `ip route delete {destination}[/mask] via {nexthop}`. You can also have routes point to exit interfaces rather than next hops by using the syntax `ip route add {destination}[/mask] dev {intf}`. You can add a default route by using the syntax `ip route add default via {next_hop} dev {intf}`.

One final note on the `ip` utility is that any configuration performed using the commands discussed in this section is not persistent. Any changes to the configuration disappear after a system reboot. Persistent configuration is discussed in the following sections.

The NetworkManager Service

NetworkManager is the default network management service on several Linux distros, including Red Hat and Fedora. Because NetworkManager is a service, you can check its status, and you can start, stop, enable, or disable it as you can any other service on Linux by using the `systemctl` command. For example, the command `systemctl status network-manager` displays the current status of the service. To poll NetworkManager for information or push configuration to it, you can use one of several user interfaces:

- **Graphical user interfaces (GUIs):** There are two main graphical user interface tools that interact with NetworkManager. The first is the Network Control Center, which is accessible via the Settings menu. The Settings window has an icon labeled Network that opens the network control center, which provides basic network configuration. The other GUI tool is the Connection Editor and is used to configure more advanced settings. You can start the Connection Editor from the terminal by entering the command `nm-connection-editor`.

- **NetworkManager command-line interface (nmcli):** The NetworkManager CLI is a command-line utility that you can use to control NetworkManager. You can use this interface to NetworkManager via the `nmcli` command in the Bash shell.
NetworkManager text user interface (nmtui): Similar to the interface used to configure a computer’s BIOS settings or old DOS-based programs, the nmtui provides an interface to NetworkManager that displays graphics in text mode. You start the text user interface by issuing the nmtui command in the shell.

API: NetworkManager provides an API that can be used by applications for programmatic access to NetworkManager.

Because the majority of automation is typically performed through CLI tools (and API calls) and not the GUI, this section covers NetworkManager configuration via the nmcli interface.

NetworkManager deals with objects called connections. A connection is a representation of a link to the outside world and may represent, for example, a wired connection, a wireless connection, or a VPN connection. To display the current status of all network connections on a system, use the command `nmcli con show`, as shown in Example 3-48.

```
Example 3-48  Listing All Connections on a System

[root@server1 ~]# nmcli con show
<table>
<thead>
<tr>
<th>NAME</th>
<th>UUID</th>
<th>TYPE</th>
<th>DEVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wired connection 1</td>
<td>d8323782-5cf2-3afc-abcd-e603605ac4f8</td>
<td>802-3-ethernet</td>
<td>--</td>
</tr>
<tr>
<td>Wired connection 2</td>
<td>669f6f4-6c57-3d1b-b832-2b2125e0036b</td>
<td>802-3-ethernet</td>
<td>--</td>
</tr>
</tbody>
</table>
```

The output in Example 3-48 indicates that there are two connections, named Wired connection 1 and Wired connection 2. These connections are not bound (applied) to any interfaces, as indicated by the -- in the last column. Both connections are of type Ethernet. A connection is uniquely identified by its universally unique identifier (UUID). Although not shown in the command output, a connection can either be active or inactive. To activate an inactive connection, you use the command `nmcli con up [connection_name]`. To deactivate a connection, you replace the keyword `up` with the keyword `down`.

Each connection is known as a connection profile and contains several attributes or properties that you can set. These properties are known as settings. Connection profile settings are created and then applied to a device or device type. Settings are represented in a dot notation. For example, a connection’s IPv4 addresses are represented by the setting `ipv4.addresses`. To drill down on the details for a specific connection and list its settings and their values, you can use the command `nmcli con show [connection_name]`. Example 3-49 lists the connection profile settings for Wired connection 1. The output is truncated due to the length of the list. A full list of settings and their meanings can be found in the man pages for the nmcli command.

```
Example 3-49  Connection Profile Settings

[root@server1 ~]# nmcli con show Wired connection 1
```

Note: Use of the terms “master” and “slave” is ONLY in association with the official terminology used in industry specifications and standards, and in no way diminishes Pearson’s commitment to promoting diversity, equity, and inclusion, and challenging, countering and/or combating bias and stereotyping in the global population of the learners we serve.
### Example 3-49  Connection Attributes for Wired Connection 1

```bash
[root@server1 ~]# nmcli con show "Wired connection 1"
connection.id:                          Wired connection 1
connection.uuid:                        d8323782-5cf2-3afc-abcd-e603605ac4f8
connection.stable-id:                   --
connection.interface-name:              --
connection.type:                        802-3-ethernet
connection.autoconnect:                 yes
connection.autoconnect-priority:        -999
connection.autoconnect-retries:         -1 (default)
connection.timestamp:                   1525512827
connection.read-only:                   no
connection.permissions:                 --
connection.zone:                        --
connection.master:                      --
connection.slave-type:                  --
connection.autoconnect-slaves:          -1 (default)
connection.secondaries:                 --
connection.gateway-ping-timeout:        0
connection.metered:                     unknown
connection.11dp:                        -1 (default)
802-3-ethernet.port:                    --
802-3-ethernet.speed:                   0
802-3-ethernet.duplex:                  --
802-3-ethernet.auto-negotiate:          no
802-3-ethernet.mac-address:             08:00:27:83:40:75
802-3-ethernet.cloned-mac-address:      --
802-3-ethernet.generate-mac-address-mask:--
802-3-ethernet.mac-address-blacklist:   --
802-3-ethernet.mtu:                     auto
802-3-ethernet.s390-subchannels:        --
802-3-ethernet.s390-nettype:            --
802-3-ethernet.s390-options:            --
802-3-ethernet.wake-on-lan:             1 (default)
802-3-ethernet.wake-on-lan-password:    --
ipv4.method:                            auto
ipv4.dns:                               --
ipv4.dns-search:                       --
ipv4.dns-options:                      (default)
ipv4.dns-priority:                     0
ipv4.addresses:                         --
ipv4.gateway:                           --
ipv4.routes:                            --
--------- OUTPUT TRUNCATED FOR BREVITY ---------
```
To list the devices (aka interfaces) on the system and the status of each one, you use the command `nmcli dev status` for all devices or the command `nmcli dev show {device_name}` for a specific device, as shown in Example 3-50.

**Example 3-50  Device Status Using the nmcli dev status and nmcli dev show Commands**

```
[root@server1 ~]# nmcli dev status
DEVICE    TYPE      STATE         CONNECTION
enp0s8    ethernet  disconnected  --
enp0s9    ethernet  disconnected  --
lo        loopback  unmanaged     --
[root@server1 ~]# nmcli dev show enp0s8
GENERAL.DEVICE:                           enp0s8
GENERAL.TYPE:                            ethernet
GENERAL.HWADDR:                          08:00:27:83:40:75
GENERAL.MTU:                             1500
GENERAL.STATE:                           30 (disconnected)
GENERAL.CONNECTION:                     --
GENERAL.CON-PATH:                       --
WIRED-PROPERTIES.CARRIER:                on
```

As you can see from the outputs in Examples 3-49 and 3-50, connections and devices are mutually exclusive. A connection profile may or may not be applied to a device after it is created.

In Example 3-51, both of the wired connections are deleted, and one new connection named NetDev_1 is created. NetDev_1 is of type ethernet and is applied to device enp0s8. Connections are deleted using the command `nmcli con del {connection_name}`. You create new connections and configure their settings by using the command `nmcli con add {connection_name} {setting} {value}`. In Example 3-51, the type, ifname, ip4, and gw4 settings are set to Ethernet, enp0s8, 10.1.0.10/24, and 10.1.0.254, respectively. Note that in this command, `setting` can either be entered in the full dot format or in abbreviated format. For example, the IP address can be set using either `ip4` or `ipv4.address`.

**Example 3-51  Deleting and Creating Connections**

```
[root@server1 ~]# nmcli con show
NAME                 UUID                                  TYPE            DEVICE
Wired connection 1   d8323782-5cf2-3af2-abc-abcd-e6030554acf8  802-3-ethernet  --
Wired connection 2   669efb4-3c57-3d19-b832-b2125e0036b       802-3-ethernet  --
[root@server1 ~]# nmcli con del "Wired connection 1"
Connection 'Wired connection 1' (d8323782-5cf2-3af2-abc-abcd-e6030554acf8) successfully deleted.
[root@server1 ~]# nmcli con del "Wired connection 2"
```
Connection 'Wired connection 2' (669fefb4-bc57-3d19-b83b-2b2125e0036b) successfully deleted.

[root@server1 ~]# nmcli con show
NAME UUID TYPE DEVICE
[root@server1 ~]# nmcli con add con-name NetDev_1 type ethernet ifname enp0s8 ip4 10.1.0.10/24 gw4 10.1.0.254
Connection 'NetDev_1' (a8ac9116-697a-4a0a-85a2-63428d6e75a3) successfully added.

[root@server1 ~]# nmcli con show
NAME UUID TYPE DEVICE
NetDev_1 a8ac9116-697a-4a0a-85a2-63428d6e75a3 802-3-ethernet enp0s8

[root@server1 ~]# nmcli con show --active
NAME UUID TYPE DEVICE
NetDev_1 a8ac9116-697a-4a0a-85a2-63428d6e75a3 802-3-ethernet enp0s8

[root@server1 ~]# nmcli con show
NAME UUID TYPE DEVICE
NetDev_1 a8ac9116-697a-4a0a-85a2-63428d6e75a3 802-3-ethernet enp0s8

[root@server1 ~]# nmcli con show
NAME UUID TYPE DEVICE
NetDev_1 a8ac9116-697a-4a0a-85a2-63428d6e75a3 802-3-ethernet enp0s8

[root@server1 ~]# nmcli dev status
DEVICE TYPE STATE CONNECTION
enp0s8 ethernet connected NetDev_1
enp0s9 ethernet disconnected --
lo loopback unmanaged --

[root@server1 ~]# nmcli dev show enp0s8
GENERAL.DEVICE: enp0s8
GENERAL.TYPE: ethernet
GENERAL.HWADDR: 08:00:27:83:40:75
GENERAL.MTU: 1500
GENERAL.STATE: 100 [connected]
GENERAL.CONNECTION: NetDev_1
GENERAL.CON-PATH: /org/freedesktop/NetworkManager/ActiveConnection/359
WIRED-PROPERTIES.CARRIER: on
IP4.ADDRESS[1]: 10.1.0.10/24
IP4.GATEWAY: 10.1.0.254
IP6 ADDRESS[1]: fe80::8c1f:4c4a:51a5:6423/64
IP6.GATEWAY: --

[root@server1 ~]# ping 10.1.0.20 -c 3
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=64 time=0.604 ms
64 bytes from 10.1.0.20: icmp_seq=2 ttl=64 time=0.602 ms
64 bytes from 10.1.0.20: icmp_seq=3 ttl=64 time=0.732 ms

--- 10.1.0.20 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 0.602/0.646/0.732/0.060 ms

Notice that once a connection has been created and the device enp0s8 has been bound to it (all in the same command), the connection and device both come up, and that results in the device successfully pinging server2 on the other end of the link.
After a connection is created, you can modify its settings by using the command `nmcli con mod {connection_name} {setting} {value}`. When modifying a setting, the full dot format is required in the command. If the shorthand format is used, the new value in the command may be added to the existing value of the setting. For example, if the shorthand format is used to modify the IP address, the new IP address in the command is added to the device as a secondary IP address. On the other hand, if the full dot format is used, the IP address in the command replaces the IP address configured on the device. Example 3-52 shows how to modify the IP address of device enp0s8 to 10.1.0.100/24.

**Example 3-52  Deleting and Creating Connections**

```
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr
ipv4.addresses:                         10.1.0.10/24
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.10/24
[root@server1 ~]# nmcli con mod NetDev_1 ip4 10.1.0.100/24

! The new IP address is added as a secondary address due to the shorthand format
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr
ipv4.addresses:                         10.1.0.10/24, 10.1.0.100/24
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.10/24

[root@server1 ~]# nmcli con up NetDev_1
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnection/366)

! After resetting the con, the new IP address now is reflected to the device
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.10/24
IP4.ADDRESS[2]:                         10.1.0.100/24

[root@server1 ~]# nmcli con mod NetDev_1 ipv4.address 10.1.0.100/24
[root@server1 ~]# nmcli con up NetDev_1
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnection/367)
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr
ipv4.addresses:                         10.1.0.100/24
[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD
IP4.ADDRESS[1]:                         10.1.0.100/24
```

Note that each time a change is made to a connection using `nmcli`, the connection needs to be reactivated in order for the changes to be reflected to the device.

Adding routes using `nmcli` is different than adding routes using the `ip` utility in that when using `nmcli`, routes are added per interface and not globally. You add routes by using the syntax `nmcli con mod {intf} +ipv4.routes {destination} ipv4.gateway {next_hop}`. Therefore, to accomplish the same task that was done earlier by using the `ip` utility (to add a route on server2 to direct traffic destined for network 10.2.0.0/24 using the next hop 10.1.0.10 on server1), you use the following command: `nmcli con mod enp0s3 +ipv4.routes 10.2.0.0/24 ipv4.gateway 10.1.0.10`.

Unlike with the `ip` utility, changes made through `nmcli` are, by default, persistent and will survive a system reload.

It is important to understand the difference between the `ip` utility and NetworkManager. The `ip` utility is a program. When you use the `ip` command, you run this program, which makes a system call to the kernel, either to retrieve information or configure a component of the Linux networking system.

On the other hand, NetworkManager is a system daemon. It is software that runs (lurks) in the background, by default, and oversees the operation of the Linux network system. NetworkManager may be used to configure components of the network or to retrieve information about the network by using a variety of methods discussed earlier in this section—one of them being `nmcli`.

The nuances of how the `ip` utility interacts with NetworkManager are not discussed in detail here. All you need to know for now is that changes to the network that are made via the `ip` utility are detected and preserved by NetworkManager. There is no conflict between them. As mentioned at the very beginning of this section, different software on Linux can achieve the same result via different communication channels with the kernel. However, any software that needs access to the network will eventually have to go through the kernel.

### Network Scripts and Configuration Files

The third method for configuring network devices and interfaces is to modify network scripts and configuration files directly. Different files in Linux control different components of the networking ecosystem, and editing these files was the only way to configure networking on Linux before NetworkManager was developed. Configuration files and scripts can still be used instead of, or in addition to, NetworkManager.

On Linux distros in the Red Hat family, configuration files for network interfaces are located in the `/etc/sysconfig/network-scripts` directory, and each interface configuration file is named `ifcfg-<intf_name>`. The first script that is executed on system bootup is `/etc/init.d/network`. When the system boots up, this script reads through all interface configuration files whose names start with `ifcfg`. Example 3-53 shows the `ifcfg` file for the `enp0s8` interface.
Example 3-53  Interface Configuration File for Interface enp0s8

```
[root@server1 network-scripts]# cat ifcfg-enp0s8
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=dhcp
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=NetDev_1
UUID=a8ac9116-697a-4a0a-85a2-63428d6e75a3
DEVICE=enp0s8
ONBOOT=yes
[root@server1 network-scripts]#
```

The filename just needs to be prefixed with ifcfg. The network script simply scans the directory and reads any file whose name has this prefix. Therefore, you can safely assume that the configuration file is for the interface or connection. However, while the filename has to start with ifcfg, there is general consensus that the value in the DEVICE field (interface) should follow the ifcfg prefix.

The TYPE field in the file indicates the connection type, which is Ethernet in this case. The BOOTPROTO field is set to dhcp, which means the connection gets an IP address via DHCP. If a static IP address is required on the interface, then dhcp is replaced with none. The interface associated with this configuration is also shown in the DEVICE field (enp0s8 in this case), and the ONBOOT field indicates that this connection is to be brought up at system bootup. When a static IP address is required on the interface, the fields IPADDR, PREFIX, and GATEWAY and their respective values are added to the file.

When ONBOOT=yes is set, the /etc/init.d/network script checks whether this interface is managed by NetworkManager. If it is and the connection has already been activated, no further action is taken. If the connection has not been activated, the script requests NetworkManager to activate the connection. In case the connection is not managed by NetworkManager, the network script activates the connection by running another script, /usr/sbin/ifup. The ifup script checks the field TYPE in the ifcfg file, and based on that, it calls another type-specific script. For example, if the type of the connection is Ethernet, the ifup-eth script is called. Linux requires type-specific scripts because different connection types require different configuration parameters. For example, the concept of an SSID (wireless network name) does not exist for an Ethernet connection. Similarly, to bring down an interface for an unmanaged interface, the ifdown script is called. The vast majority of interface types are managed by NetworkManager by default, unless the line NM_CONTROLLED=no has been added to the ifcfg file.
While the recommended method for configuring interfaces is to use the `nmcli` utility, as discussed in the previous section, you can also configure interfaces by editing the corresponding `ifcfg` file.

Static routes configured on a system have configuration files named `route-intf_name` in the same directory as the interface configuration files. As you have probably guessed, the name has to be prefixed with `route`. However, the `<intf_name>` is just a naming convention, and the file may have any name as long as the prefix `route` is there. The routing entries in the file may have one of two formats:

- The `ip` command arguments format:

  `{destination}/mask` via `{next_hop} [dev interface]`

  With this format, specifying the interface using `[dev interface]` is optional.

- The network/netmask directives format:

  `ADDRESS[N]:{destination}`
  `NETMASK[N]:{netmask}`
  `GATEWAY[N]:{next_hop}`

  where `N` is the routing table entry starting with 0 and incrementing by 1 for each entry, without skipping any values. In other words, if the routing table has four entries, the entries are numbered from 0 to 3.

Going back to the network of three servers in Figure 3-2, where server1 is required to route between server2 on subnet 10.1.0.0/24 and server3 on subnet 10.2.0.0/24: the static routes previously configured in order to route between the servers are deleted, after which the ping from server2 to server3 fails, as shown in Example 3-54.

**Example 3-54  Ping Fails Due To Lack of Static Routes on server2 and server3**

```bash
# No routes in routing table of server2 to remote subnet 10.2.0.0/24
[root@server2 ~]# ip route
10.1.0.0/24 dev enp0s3 proto kernel scope link src 10.1.0.20 metric 100

# Ping to the directly connected interface on server1 is successful
[root@server2 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.828 ms

--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.828/0.828/0.828/0.000 ms
```

```bash
# Pings fail from server2 to server3
[root@server2 ~]# ping -c 1 10.2.0.10
ping: unknown host 10.2.0.10
```
The file route-enp0s3 is created under the directory `/etc/sysconfig/network-scripts/` on both servers. A routing entry is added to the routing configuration file on server2 by using the `ip` command arguments format, and a routing entry is added to the file on server3 by using the network/netmask directives format, as shown in Example 3-55.

**Example 3-55** Routing Configuration Files Added on Both server2 and server3

```bash
# server2

! No routing configuration files in the directory
[root@server2 ~]# cd /etc/sysconfig/network-scripts/
[root@server2 network-scripts]# ls -l | grep "route"
[root@server2 network-scripts]#

! Create the file route-enp0s3 and populate it with a route to the remote subnet 10.2.0.0/24 using the IP Command Arguments format
[root@server2 network-scripts]# touch route-enp0s3
[root@server2 network-scripts]# echo "10.2.0.0/24 via 10.1.0.10" >> route-enp0s3
[root@server2 network-scripts]# ls -l | grep "route"
-rw-r--r--. 1 root root    26 Aug 17 15:52 route-enp0s3
```
The ping test is now successful, and server2 can reach server3, as shown in Example 3-56.
Example 3-56  Ping from server2 to server3 and Vice Versa Is Successful Now

[root@server2 network-scripts]# ping -c 1 10.2.0.30
PING 10.2.0.30 (10.2.0.30) 56(84) bytes of data.
64 bytes from 10.2.0.30: icmp_seq=1 ttl=63 time=2.11 ms

--- 10.2.0.30 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.119/2.119/2.119/0.000 ms
[root@server2 network-scripts]#

[root@server3 network-scripts]# ping -c 1 10.1.0.20
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=63 time=1.58 ms

--- 10.1.0.20 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.585/1.585/1.585/0.000 ms
[root@server3 network-scripts]#

The network script is run as a service and, like any other service, can be controlled by using the command systemctl {start|stop|restart|status} network. To enable/disable the network service at startup, you use the command chkconfig network {on|off}. Keep in mind that after a configuration file is changed, the network service has to be restarted for the changes to take effect. It goes without saying that any configuration done via amending the network configuration files is persistent and will remain intact after a system reload.

Network Services: DNS

Domain Name System (DNS) is a hierarchical naming system used on the Internet and some private networks to assign domain names to resources on the network. Domain names tend to be easier to remember than IP addresses. Using domain names provides the additional capability to resolve a domain name to multiple IP addresses for purposes such as high availability or routing user traffic based on the geographically closest server.

DNS uses the concept of a resolver, commonly referred to as a DNS server, which is a server or a database that contains mappings between domain names and the information related to each of those domain names, such as the IP addresses. These mappings are called records. DNS is hierarchical and distributed. The majority of DNS servers maintain records for only some domain names and then initiate queries to other DNS servers for the rest of the domain names, for which it does not maintain records.
Performing a DNS query means sending a request to a DNS server to resolve the domain name and return the data associated with that domain name. To resolve a domain name on Linux to its corresponding information, including its IP address, you use the `dig` command, which stands for *domain information groper*. Example 3-57 shows `dig` being used to resolve google.com to its public IP address. The public IP address received from the DNS response is highlighted in the example.

**Example 3-57 Using the `dig` Command to Resolve google.com**

```
[root@server1 ~]# dig google.com
;; <<>> DiG 9.9.4-RedHat-9.9.4-51.el7_4.2 <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38879
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;google.com.

;; ANSWER SECTION:
google.com. 264 IN A 216.58.207.14

;; Query time: 31 msec
;; SERVER: 192.168.8.1#53(192.168.8.1)
;; WHEN: Fri Aug 17 17:16:06 +03 2018
;; MSG SIZE  rcvd: 55
```

In Example 3-57, the DNS server used for the name resolution is 192.168.8.1. The IP address of this DNS server is configured in the `/etc/resolv.conf` file, shown in Example 3-58. To configure other DNS servers, you list each server’s IP address on a new line in this file.

**Example 3-58 List of DNS Servers in the `/etc/resolv.conf` File**

```
[root@server1 ~]# cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 192.168.8.1
```

Manual DNS entries are configured in the /etc/hosts file. If an entry for a domain name is found in that file, the DNS servers are not consulted for resolution. There is one caveat, though: The dig command still requests the name resolution from the DNS server configured in /etc/resolv.conf. However, the ping command and also the web browsers on the system use the hosts file, and, therefore, use the manual entry there. Try to add a manual entry for google.com in the hosts file, pointing to an IP address that is not reachable and then try to use dig, use ping, and browse to google.com and notice how each of these behave differently.

Summary

This chapter takes Linux administration a step further and covers storage, security, and networking. It discusses the following topics:

- Partitioning, formatting, and managing physical storage
- Creating physical volumes, volume groups, and logical volumes using LVM
- User and group security management
- File security management, including permission bits and ACLs
- Linux system security, including the Linux firewall
- Managing Linux networking by using the ip utility
- Managing Linux networking by using the NetworkManager CLI (nmcli)
- Managing Linux networking via network scripts and configuration files
- Network services such as DNS

Chapter 4, “Linux Scripting,” builds on this chapter and covers Linux scripting, which is one big step towards automation.
Symbols

& (ampersands)
  &= operator, Python, 284
  AND operator, Python, 281, 285–286
*
  *= operator, Python, 281
  ** assignment operator, Python, 281
  **= operator, Python, 284
  *= operator, Python, 284
\ (backslashes)
  \<, regular expressions (regex), 185, 188–189
  \\, regular expressions (regex), 185, 188–189
  \>, regular expressions (regex), 185, 188–189
  regular expressions (regex), 186
^ (cotes)
  ^= operator, Python, 284
  regular expressions (regex), 185, 187
  XOR operator, 281
[ ] (curly braces)
  [N], regular expressions (regex), 186, 189–192
[N, M], regular expressions (regex), 186, 189–192
regular expressions (regex), 185
$ (dollar signs), regular expressions (regex), 185, 187
. (dots),
  .* notation, regular expressions (regex), 190
  .. notation, Linux directories, 37
  regular expressions (regex), 185, 189
= (equal signs)
  = assignment operator, Python, 284
  == (double equal sign)
    conditional statements, 229
    Jinja2 operator, 1019
    Python operator, 285
    conditional statements, 229
!= operator
  Jinja2, 1019
  Python, 285
/ (forward slashes)
  / assignment operator, Python, 281
  /: root directory, Linux, 36–37
  /= operator, Python, 284
< (left arrows)
- operator, Python, 285, 1019
<< operator, Python, 281
<= assignment operator
Jinja2, 1019
Python, 285
-
(minus signs)
-= operator, Python, 284
- assignment operator, Python, 281
() (capture groups), regular expressions (regex), 185–186
% (percentage symbols)
%= operator, Python, 284
Modulo Operator, 274, 281
| (pipes), 58, 65–67, 281
| (OR operator)
Python, 281, 285
regular expressions (regex), 186
|= operator, Python, 284
+
(plus signs)
+= operator, Python, 281
regular expressions (regex), 186, 189–190, 192
#!/ (hashbangs), Linux shell scripting, 205–208
? (question marks), regular expressions (regex), 186, 189, 191–192
> (right arrows)
> Jinja2 operator, 1019
> Python operator, 285
>> (right arrow, equal sign)
assignment operator, Jinja2, 1019
assignment operator, Python, 285
>>> operator, Python, 284
; (semicolons)
;&, case-in constructs, 233–234
;; (double semicolons), case-in constructs, 233–234
;&&, case-in constructs, 233–234
Linux notation, 61–62
[ ] (square brackets)
[=] operator, Python, 284
[first _literal - last _literal], regular expressions (regex), 185
[literals], regular expressions (regex), 185
Python
lists, 286–287
strings, 278
regular expressions (regex), 185
~ (NOT) operator, Python, 281, 285

Numbers

0-RTT, TLS, 502–503
1xx information status codes, 411
2xx successful status codes, 411–412
3xx redirection status codes, 412
4xx server error status codes, 413–414
5xx client error status codes, 414
200 OK responses, static routing
DELETE method, 405–406
POST method, 394–401
PUT method, 403–404

A

absolute paths
Linux directories, 38–39
XPath expressions, 576–577
abstraction
API, 12–13
defined, 9–13
NaC, 12
OOP, 257
single sources of truth, 11–12
access tokens, OAuth protocol, 481–483
ACI (Application Centric Infrastructures), 13
ACL, Linux, 148–155
ad hoc command, Ansible, 994–997
Adj-SID, 824, 827, 839–842
AEAD (Authenticated Encryption with Associated Data), 495–496
AES-CCM protocol, 495
AES-GCM protocol, 495
alert protocol, TLS 1.3, 499
algorithms, 258–259
ampersands (&), &= operator, Python, 284
AND operator, Python, 281, 285
analyzing cost/benefit analysis, 1112
anchors, YAML, 624–625
AND operator (&), Python, 281, 285–286
AND/OR logic, combining multiple conditional statements, 1022–1024
Android, Linux distributions, 26
annotation, JSON schemas, 596
Ansible, 15–16, 989
ad hoc command, 994–997
basics, 989–990
call flows, for a single command, 990–991
conditional statements, 1016–1019
AND/OR logic, 1022–1024
checking for substrings in variables, 1021–1022
checking for variables, 1019–1021
combining multiple conditional statements, 1022–1024
Jinja2 templates, 1045–1049
with loops, 1024–1027
with loops and variables, 1027–1033
configuring, 991–995
connection plug-ins, 1003–1004
filters, 1013–1015
help, 995–996
installing, 990
inventories
default paths, 992
IP addresses, 993–994
simple inventory files, 992–993
inventory files, defining variables, 1009–1011
IOS XE
clearing counters, 1060–1061
configuring, 1061–1069
configuring with ios_* modules, 1069–1073
configuring with iosxr_* modules, 1083–1084
preparing for Ansible management, 1055–1057
preparing for NETCONF management, 1095–1096
updating files with additional hosts/variables, 1057–1058
verifying Ansible management, 1057
verifying operational data, 1058–1060
IOS XR
configuring, 1078–1084
preparing for management, 1073–1074
preparing for NETCONF management, 1096–1098
verifying Ansible management, 1074–1075
verifying operational data, 1074–1078
ios_command module
clearing counters, 1060–1061
verifying operational data, 1058–1060
iosxr_command
input parameters, 1001
playbooks, 997–999
verifying operational data, 1074–1078
Jinja2 templates, 1034–1040
conditional statements, 1045–1049
loops, 1040–1043
playbooks, 1040–1043
variables, 1042–1043
Linux, host files, 993
loops
conditional statements with loops, 1024–1027
Jinja2 templates, 1040–1043
modules
control functions, 1001–1002
debug modules, 999
file modules, 1003
iosxr_command, parameters, 1001
network modules, 1003
return values, 1001–1002
structure of, 1000
utility modules, 1003
NETCONF
configuring, 1103–1107
IOS XE management, 1095–1096
IOS XR management, 1096–1098
NX-OS management, 1098
verifying operational data, 1098–1103
NX-OS
collecting show output with nxos_command, 1086–1088
configuring, 1090–1095
interactive commands, 1088–1089
preparing for management, 1084–1085
preparing for NETCONF management, 1098
verifying management, 1085–1086
verifying operational data, 1086–1089
overview of, 989–990
playbooks, 990, 997–1000
conditional statements
with loops and variables, 1032–1033
defining variables, 1005–1006
Jinja2 templates, 1040–1043
Python, 991–992
variables, 999
Boolean variables, 1006
checking for substrings with conditional statements, 1021–1022
checking with conditional statements, 1019–1021
conditional statements with loops and variables, 1027–1033
defining from external files, 1007–1009
defining in inventory files, 1009–1011
defining in playbooks, 1005–1006
dictionary variables, 1007
importing from external files, 1007–1009
Jinja2 templates, 1042–1043
list variables, 1007
setting dynamically, 1011–1013
string variables, 1006
types of, 1006–1007
version command, 991–992
anywhere selection, XPath expressions, 576
API (Application Programming Interface)
abstraction, 12–13
automation, 12–13
AXL API, 944
CER API, 944
collaboration API, 942–944
AXL API, 944
CER API, 944
CUCM Serviceability API, 945
Finesse Desktop API, 946–947
PAWS API, 944
REST API, 945–946, 948–954
TSP API, 945
UDS API, 945
URL API, 945
xAPI, 946
XML API, 945
CUCM Serviceability API, 945
DNA Center API
device management, 934
Eastbound API, 933
event notifications, 935
Integration API, 935–936
Intent API, 934, 936–941
Northbound API, 933
Southbound API, 933
webhooks, 935
Westbound API, 933
eastbound API, 883
endpoints, 882
IOS XE
gNMI, insecure mode, 815
NETCONF, 918–922
programmability, 885–886
IOS XR
NETCONF, 916–918
programmability, 886–887
Linux, 24
Meraki API, 922, 923
Captive Portal API, 923
Dashboard API, 922–931
Location Scanning API, 923
MV Sense API, 923
Webhook Alerts API, 922
model-based industry-standard API
IOS XE programmability, 885
Open NX-OS programmability, 884
NETCONF
IOS XE, 918–922
IOS XR, 916–918
NX-OS, 905–916
northbound API, 883
NX-API CLI, use cases, 893–898
NX-API REST, use cases, 898–905
Open NX-OS
  Bash shells, 887–891
  Guest shells, 887, 891–892
  NETCONF, 905–916
  programmability, 884–885
  use cases, 887–892
PAWS API, 944
platforms, 882
Postman, 436–437
  installing, 438
  interface, 438–441
  usage, 441–446
resource server calls, OAuth protocol, 483
REST API, 322, 392–393, 945–946, 948–954
RESTful API, 883
RPC-based API, 883
rules of thumb, 1118
service layer API, IOS XR programmability, 886
southbound API, 883
transport protocols, 18–19
TSP API, 945
UDS API, 945
URL API, 945
vendor/API matrix, network programmability, 957–958
web/API development, 336–337
  back end development, 336
  Django, 337–345
  Flask, 345–352
  front end development, 336
  Postman, 337–345
  webhooks, 882
westbound API, 883
XML API, 945
transport protocols, 18–19
API resource, RESTCONF, 747–749
applications
developing
different environments, 311
  Docker, 317–331
  Git, 312–317
  organizing development environment, 311–312
  Python modules, 333–336
  replicating product environments, 312
  reusable code, 312
  version control, 311
  virtualenv tool, 331–333
Django
  creating applications, 341–345
demo applications, 343–345
dockerizing, 326–331
hosting
containerized application hosting, 1116
  IOS XE programmability, 886
  IOS XR programmability, 887
  iPerf, 1116
  native application hosting, 1115–1116
  Open NX-OS programmability, 885
  rules of thumb, 1115–1116
Linux communication, 24
Python
  machine learning, 382–384
  network automation, architectures, 353–354, 371–375
arrays

JSON arrays, 593

Linux scripting

- adding/removing elements, 224–226
- associative arrays, 222
- concatenating, 221–226
- declaring, 222–224
- defined, 222
- indexed arrays, 222–224

assignment operators, Python, 284

associative arrays, 222

asterisks (*)

Python

- *= operator, 284
- ** assignment operator, 281
- **= operator, 284
- assignment operator, 281

regular expressions (regex), 185, 189–190

asymmetric keys, 490

attributes, XML, 558, 568, 570

augmentation, YANG modules, 656–658

authentication

- host-based authentication, 517–518
- HTTP/1.1, 469–471

- base64 encoding, 472, 474
- basic authentication, 472–474
- OAuth protocol, 474–483
- UTF-8 encoding, 472–473
- workflows, 470

- key-based authentication, SSH, 523–525
- MAC, 493–494
- NETCONF, 694
- Nexus switches, 401–402, 463
password authentication, 517, 522–523, 525–526
peer authentication, 496–497
public key authentication, 516–517
SSH Authentication Protocol, 514–516
host-based authentication, 517–518
password authentication, 517, 522–523, 525–526
public key authentication, 516–517
authorization grants, OAuth protocol, 477–481
 automation
API, 12–13
benefits, 6
broken processes, 1110
cloud computing, 1118
complexity, 1111–1112
configuration management automation
IOS XE programmability, 886
Open NX-OS programmability, 885
cost/benefit analysis, 1112
defined, 5–6
model-driven telemetry, 1113–1114
Network Programmability and Automation toolbox, 14–15
Ansible. See also separate entry, 15–16
Linux, 16–17
protocols, 18–19
Python, 15
virtualization, 17
YANG, 17

networks
 architectures, 353–354
 Jinja2 templates, 363–375
 NAPALM libraries, 354–359, 371–375
one-time automations, 1111
orchestration versus, 6–7
reusing automations, 1111
rules of thumb, 1109–1112, 1118
single sources of truth, 11–12
software/network engineers, 19–20
awk programming language, 194–197
AXL API, 944

B

back end web/API development, 336
backslashes (\), regular expressions (regex), 186
<, regular expressions (regex), 185, 188–189
>, regular expressions (regex), 185, 188–189
base64 encoding, HTTP/1.1

authentication, 472, 474

Bash, 184
Arguments.bash script, 213–214
arithmetic operators, 220–222, 229
CLI programmable interface creation, 963–967
Expect programming language, 245–246
file comparison operators, 230–232
functions, 244
HTTP, 447–454
integer comparison operators, 229–230
IOS XR programmability, 886
Linux, 29
Linux interface configuration, 969–970
Open NX-OS, 884, 887–891
scripting, 206–207, 213–214
SSH, 539–540, 548–549
string comparison operators, 228–229
string operators, 227–228
Bearer Tokens, HTTP, 475–476
beginnings/endings of words, matching, regular expressions (regex), 188–189
benefit/cost analysis automation, 1112
BGP, SR-TE, 836–843
BGP-LS (BGP-Link State), 843
lab topologies, 845–846
link NLRI, 856–857
node NLRI, 854–855
NPF-XR, 845, 849, 851–854
peering, 843–844, 847–849
prefix NLRI, 858–859
routinig, 846–847
routing types (overview), 850–854
bidirectional RPC, 785
binary message framing, HTTP/2, 506–507
BIOS (Basic Input/Output Systems), 27
bitwise operators, Python, 281–283
blkid command, Linux, 148–149
block ciphers, 492–493
blocking, head-of-line, 504
Boolean data, JSON, 593

Boolean variables, Ansible, 1006
boot directory, Linux, 36
boot process, Linux, 26–28
broken processes, automation, 1110
buffered/unbuffered access, Linux /dev directory, 120
built-in data types, YANG modules, 647–648
Business Edition (Cisco), 942
bytecode, Python, 265–267
generators, 264
interpreters, 264
bzip2 archiving utility, Linux, 67, 69

call flows, PCEP, 861–864
<cancel-commit> operations, NETCONF, 722–724
candidate configuration, NETCONF, 722–724, 732
capabilities, NETCONF, 731
candidate configuration capability, 732
confirmed commit capability, 732
distinct startup capability, 733
rollback-on-error capability, 732–733
URL capability, 733–734
validate capability, 733
writable-running capability, 732
XPath capability, 735
Capabilities RPC, gNMI, 810–811
CAPEX (Capital Expenditures), 2
Captive Portal API (Meraki), 923
capture groups ( ( ) ), regular expressions (regex), 185–186
career paths, software/network engineers, 1118–1119
carets (^)

^= operator, Python, 284
regular expressions (regex), 185, 187
XOR operator, 281
case-in constructs, Linux scripting, 232–234
;&233–234
;;, 233–234
;;&233–234
cat command, 41–42, 61–62
cat/proc/cpuinfo command, Linux, 87–88
CBC (Cipher Block Chaining), 492–493
CCM (Counter with CBC mode), 495–496
CentOS
Docker
containers, 322–325
installing, 318–320
Git, setting up, 313–314
OpenSSH installations, 522
SCP, 549–550
SSH setup, 521–526
CER API, 944
changing
CLI, 19
file users/groups, Linux, 146
channels, SSH Connection Protocol, 518–521
character classes, matching uppercase/lowercase characters in regular expressions (regex), 187–188
child/parent relationships, XML, 555–556
chmod command, Linux, 144–146
chown command, Linux, 146
Chrome (Google), LocalRepo repositories, 114–117
ciphers, 492
block ciphers, 492–493
CBC, 492–493
CMAC, 494
CTR mode, 492–493
symmetric ciphers, 492
Cisco Business Edition, 942
Cisco collaboration portfolio, 942–944
Cisco devices
server status codes, 414
SSH setup, 545–549
Cisco Finesse, 943, 946–947
Cisco IOS XE
Ansible
clearing counters, 1060–1061
configuring IOS XE, 1061–1069
configuring IOS XE with ios_* modules, 1069–1073
configuring IOS XR with iosxr_* modules, 1083–1084
preparing IOS XE for Ansible management, 1055–1057
preparing IOS XE for NETCONF management, 1095–1096
updating files with additional hosts/variables, 1057–1058
verifying IOS XE for Ansible management, 1057
verifying operational data, 1058–1060
SSH setup, 526–531, 545–546
Cisco IOS XR
Ansible
configuring IOS XR, 1078–1084
preparing for management, 1073–1074
preparing IOS XR for NETCONF management, 1096–1098
verifying IOS XR for Ansible management, 1074–1075
verifying operational data, 1074–1078
PCEP configurations, 867–880
SSH setup, 532–536, 546–547
Cisco IP Phones, 944
Cisco Meeting Server, 943
Cisco Unified Contact Center, 943
Cisco Webex Board, 943
Cisco Webex Cloud Calling, 942
Cisco Webex Contact Center, 943
Cisco Webex Meetings, 943
REST API, 945–946, 948–954
TSP API, 945
URL API, 945
XML API, 945
Cisco Webex Room Series, 944
Cisco Webex Support, 943
Cisco Webex Teams, 942, 945–946, 948–954
classes, Python modules, 335–336
cleaning up networks, 1110
clear command, 30
CLI (Command-Line Interface)
API versus, 8
changing, 19
command shell, Linux, 28–30
loopback interfaces, CLI programmability, 962–963
network programmability, 958–959, 962–967
NX-API CLI
Open NX-OS programmability, 884
use cases, 893–898
as programmable CLI, 962–967
transport protocols, 19
unstructured data, 19
client registration, OAuth protocol, 476–477
client requests, HTTP, 388–392, 397–398
CONNECT method, 407
DELETE method, 405–406
GET method, 398
Bash shells, 447–454
Postman, 445–446
HEAD method, 398
header fields, 422–425
OPTIONS method, 407–408
POST method, 399–402
Postman, 443–445
Python and HTTP, 465
PUT method, 402–405
TRACE method, 408
client/server connections, HTTP, 394–395
client-streaming RPC, 785
cloning
git repositories, 316–317
YANG modules, 665
<close-session> operations, NETCONF, 721–722
cloud computing
Cisco Webex Cloud Calling, 942
rules of thumb, 1118
CMAC (Cipher-based MAC), 494
CMDB (Configuration Management Database), 12
code execution, Python, 263–269
bytecode, 265–267
generators, 264
interpreters, 264
code testing/verification, 269
compiling code, 265–266
executable Python files, 265
lexical analyzers, 263
numeric data, 269
simple Python program, 264
tokenizers, 263
collaboration
API, 942–944
  AXL API, 944
  CER API, 944
  CUCM Serviceability API, 945
  Finesse Desktop API, 946–947
  PAWS API, 944
  REST API, 945–946, 948–954
  TSP API, 945
  UDS API, 945
  URL API, 945
  xAPI, 946
  XML API, 945
Cisco collaboration portfolio, 942–944
endpoints, 943
  Cisco IP Phones, 944
  Cisco Webex Room Series, 944
platforms, API, 942
  AXL API, 944
  CER API, 944
  CUCM Serviceability API, 945
  Finesse Desktop API, 946–947
  PAWS API, 944
  REST API, 945–946, 948–954
  TSP API, 945
  UDS API, 945
  URL API, 945
  xAPI, 946
  XML API, 945
collections, YAML, 618–620
command shell, Linux, 28–30
comments
  Linux scripting, 207–208
  XML, 558
  YAML, 616
<commit> operations, NETCONF, 722–724
commit phase, Git workflows, 317
compact GBP, 1113–1114
comparison operators
  Jinja2, 1018–1019
  Python, 284–285
compiling
  Linux software installations, 94, 96–97
  Python code, 265–266
complex elements, XML validation, 570–573
complex numbers, Python, 276
compose versions, Docker, 320–322
compression, HTTP/1.1, 396–397
computer science concepts, 255
concatenating
  arrays, 221–226
  strings, Python, 277
conditional statements
  Ansible, 1016–1019
    checking for substrings in variables, 1021–1022
    checking for variables, 1019–1021
    combining multiple conditional statements, 1022–1024
    conditional statements with loops, 1024–1027
    conditional statements with loops and variables, 1027–1033
Jinja2 templates, 1045–1049
AND/OR logic, 1022–1024
Jinja2 templates, 1045–1049
Linux scripting, 226
  == (double equal sign), 229
  = (equal sign), 229
  case-in constructs, 232–234
  if-then constructs, 226–232
nested code blocks with conditional statements, Python control flow, 295–296
conferencing, 943
  Cisco Meeting Server, 943
  Cisco Webex Meetings, 943
    REST API, 945–946, 948–954
    TSP API, 945
    URL API, 945
    XML API, 945
  Cisco Webex Support, 943
configuration files, Linux networking, 174–179
configuration management automation
  IOS XE programmability, 886
  IOS XR programmability, 887
  Open NX-OS programmability, 885
configuring
  Ansible, 991–995
  candidate configuration capability, NETCONF, 732
  CMDB, 12
  IOS XE with Ansible
    general configuration, 1061–1069
    with ios_* modules, 1069–1073
    with iosxr_* modules, 1083–1084
  IOS XR with Ansible, general configuration, 1078–1083
  NETCONF, 1103–1107
    <cancel-commit> operations, 722–724
    candidate configuration operations, 722–724
    <close-session> operations, 721–722
    <commit> operations, 722–724
    configuration validation, 724–725
    <copy-config> operations, 719
    datastore configurations, 712–720
    datastore operations, 720–721
    <delete-config> operations, 719–720
    <discard-changes> operations, 722–724
    <edit-config> operations, 712–719
    <kill-session> operations, 721–722
    <lock> operations, 720–721
    session operations, 721–722
    <unlock> operations, 720–721
    <validate> operations, 724–725
  NX-OS
    with nx-os_* modules, 1093–1095
    with nx-os_config modules, 1086–1088
  YAML configuration files, building, 635–637
confirmed commit capability, NETCONF, 732
CONNECT method, 407
connection plug-ins, Ansible, 1003–1004
connections, NetworkManager
attributes, 169–170
creating, 171–174
deleting, 171–174
listing, 169
constructors, Python modules, 335–336
Contact Center, 942–943
container nodes, YANG data modeling, 652–653
containerized application hosting, 1116
containers
Docker, 1115
  Docker Swarm, 1115
hello-world containers, 322–325
  virtualization, 317–318
Kubernetes, 1115
  rules of thumb, 1114–1115
content layer
  NETCONF, 693, 725–730
  RESTCONF, 743
content parameter, RESTCONF, 771
control flow, Python
  elif statements, 297–298
  for loops, 301–302, 306
    nested for loops, 303–304
    range() function, 302–303
  if-else statements, 296–300
  nested code blocks with conditional statements, 295–296
  while loops, 304–306
cookies, HTTP state management, 483–487
<copy-config> operations, 719
copying
directories, Linux, 49–51
files, Linux, 46–48
public keys onto servers, 524–525
SCP
  CentOS, 549–550
  SFTP comparisons, 550
cost/benefit analysis, automation, 1112
costs
  CAPEX, 2
  human operations, 2
  networks, 2
  OPEX, 2
cp command, 46–48, 49–51
CRLF, HTTP messages, 415, 418
cryptography, 488, 495–496
  AEAD, 495–496
  AES-CCM protocol, 495
  AES-GCM protocol, 495
  CCM, 495
  ciphers, 492
    block ciphers, 492–493
    CBC, 492–493
    CMAC, 494
    CTR mode, 492–493
    symmetric ciphers, 492
digital signatures, 496–497
encryption keys, 488–489
  asymmetric keys, 490
  DH protocol, 490–492
  ephemeral keys, 490
generation/exchanges, 488–492
HKDF, 492
  key exchange algorithm, 490
  KM, 492
  PFS, 490
PSK, 489
symmetric keys, 489
GCM, 495
MAC, 493–494
peer authentication, 496–497
.csv files, 1052
CTR mode, ciphers, 492–493
CUCM (Cisco Unified Communications Manager), 942
AXL API, 944
CER API, 944
CUCM Serviceability API, 945
PAWS API, 944
UDS API, 945
curly braces ({ })
{N}, regular expressions (regex), 186, 189–192
{N, M}, regular expressions (regex), 186, 189–192
regular expressions (regex), 185

D
daemons, Linux, 24
dashes (-)
-= operator, Python, 284
assignment operator, Python, 281
Dashboard API (Meraki), 922–931
data modeling, YANG, 642
defined, 639–640
importance of, 640–642
modules, 642–644
 \- augmentation, 656–658
\- built-in data types, 647–648
\- cloning, 665
\- derived data types, 648–649
\- deviations, 658–662
home of, 664–666
IETF YANG modules, 670–671
native (vendor-specific) modules, 666–669
OpenConfig YANG modules, 671–673
structure of, 644–646
verifying downloaded modules, 665–666
nodes, 649
 \- container nodes, 647–648
 \- grouping, 654–656
 \- leaf nodes, 649–651
 \- leaf-list nodes, 651–652
 \- list nodes, 647–648
pyang, 673–679, 683–687
pyangbind, 679–682
YANG 1.1, 662–663
data plane verification
SR, 830–831
SR-TE, 842–843
data resource, RESTCONF, 753–756
data streams, YAML
 \- saving to files, 629
 \- sorting, 630–631
data structures, Python, 286
dictionaries, 290–291
 \- deleting, 292
 \- functions, 292
 \- if-else statements, 299–300
lists, 286–288
 \- functions, 288–289
 \- if-else statements, 298–299
 \- nested lists, 289–290
 \- slicing, 286–287
 \- square brackets ([]), 286–287
 \- value assignments, 286–288
data structures, Python

- sets, 294–295
- tuples
  - deleting, 293
  - functions, 292–293
  - joining, 293

**Data Types**
- JSON, 592–594
- numbers, Python, numbers, 273–276
- numbers data types, Python, numbers, 273–276
- Python, 270, 276–280
- XML, 567–568

**Data Validation, JSON Schemas**
- 609–614

**Databases, Rules of Thumb**
- 1117

**Datastores**
- **NETCONF**
  - configuring datastores, 712–720
  - datastore operations, 720–721
- RESTCONF, 749–750

**Date Command, Linux**
- 85–86

**Debian, Linux Distributions**
- 26

**Debugging**
- debug modules, Ansible, 999
- SSH, 528–531, 533–534

**Decision Tree Algorithms, Python**
- Machine Learning, 382–384

**Declaring**
- arrays, Linux scripting, 222–224
- variables, Linux scripting, 218–219
- XML declarations, 566–567

**Delegating PCEP, LSP Delegation**
- 864–867

**<delete-config> Operations**
- 719–720

**DELETE Method**
- 405–406, 770–771

**Deleting**
- connections, NetworkManager, 171–174
- dictionaries, Python, 292
- directories, Linux, 50–51
- files, Linux, 47, 48
- groups, Linux user/group management, 141–142
- tuples, Python, 293
- users, Linux user/group management, 141
- variables, Python, 272–273

**Dependencies, Software**
- 95

**Dependency Hell**
- 1114–1115

**Depsolve**
- 95

**Depth Parameter, RESTCONF**
- 771

**Derived Data Types, YANG Modules**
- 648–649

**/Dev Directory, Linux Storage**
- 36, 119–120
- contents of, 120
- device file types, 120–121
- fdisk command, 121–125
- file system creation, 125–126
- hard disk partitions, 121–125
- mkfs command, 125–126
- mounting file systems, 126–128
- unmounting file systems, 127

**Developing Applications**
- Different Environments, 311
- Docker, 317
  - CentOS containers, 322–325
  - clients, 322
  - commands list, 325–326
  - components of, 322
  - compose versions, 320–322
container virtualization, 317–318
docker images command, 325
docker pull command, 325
Dockerfile instructions, 326–328
dockerizing applications, 326–331
hello-world containers, 322–325
installing, 318–320
Python orchestration, 376–378
REST API, 322
servers, 322
verifying, 320–322
Git
commit phase, 317
flexibility, 312
initialization commands, 314
performance, 312
pull phase, 317
push phase, 317
repositories, 312–313, 314–317
security, 312
server setup, 313–314
workflows, 317
organizing development environment, 311–312
Python
machine learning, 382–384
modules, 333–336
network automation, architectures, 353–354, 371–375
network automation, Jinja2 templates, 363–375
network automation, NAPALM libraries, 354–359, 371–375
orchestration, 375–382
web/API development, Django, 337–345
web/API development, Flask, 345–352
replicating product environments, 312
reusable code, 312
version control, 311
virtualenv tool, 331
creating virtual environments, 332–333
installing, 331
web/API development, 336–337
back end development, 336
Django, 337–345
Flask, 345–352
front end development, 336
Postman, 337–345
deviations, YANG modules, 658–662
device drivers, Linux, 23
device management, DNA Center, 934
device mappers, 135
DH (Diffie-Hellman) protocol, 490–492
dictionaries, Python, 290–291, 583–585
deleting, 292
functions, 292
if-else statements, 299–300
dictionary variables, Ansible, 1007
dict.items() function, Python, 292
dict.keys() function, Python, 292, 307–308
### dict.values() function, Python
292

### dig command, Linux
180–181

### digital signatures
496–497

### directories

**Linux, 48–49**
- `/: root directory, 36`
- `absolute paths, 38–39`
- `attributes, 40–41`
- `boot directory, 36`
- `copying, 49–51`
- `creating, 49, 51`
- `deleting, 50–51`
- `dev directory, 36`
- `double dot notation (..), 37`
- `etc directory, 36`
- `home directory, 36`
- `media directory, 37`
- `mnt directory, 37`
- `moving, 49, 51`
- `navigating, 38–41`
- `opt directory, 37`
- `proc directory, 37`
- `relative paths, 38–39`
- `renaming, 50–51`
- `root directory, 37`
- `run directory, 37`
- `srv directory, 37`
- `sys directory, 37`
- `tmp directory, 37`
- `usr directory, 37`
- `usr/bin directory, 37`
- `usr/local directory, 37`
- `usr/sbin directory, 37`
- `var directory, 37`

**local/remote directory operations, SFTP, 542–543**

### direct-tcpip channels, SSH
- `Connection Protocol, 521`

### disabling password authentication
- `525–526`

### <discard-changes> operations
- `NETCONF, 722–724`

### distinct startup capability
- `NETCONF, 733`

### distributions, Linux
26

**Django, web/API development, 337**
- `application web servers, 338–339`
- `creating applications, 341–343`
- `demo applications, 343–345`
- `installing, 337–338`
- `migrations, 338–339`
- `models.py files, 343`
- `serializer.py files, 343`
- `settings.py files, 339–341`
- `starting new projects, 337–338`
- `urls.py files, 343`
- `views.py files, 343`

**dmesg command, Linux, 90**

**dmidecode command, Linux, 88–89**

### DNA Center
931–933
- `device management, 934`
- `Eastbound API, 933`
- `event notifications, 935`
- `Integration API, 935–936`
- `Intent API, 934, 936–941`
- `Northbound API, 933`
- `Southbound API, 933`
- `webhooks, 935`
- `Westbound API, 933`

### DNF (Dandified YUM)
95, 117

### DNS (Domain Name System)
179–181

**Docker, 317, 1115**
CentOS containers, 322–325
clients, 322
commands list, 325–326
components of, 322
compose versions, 320–322
containers, virtualization, 317–318
docker images command, 325
docker pull command, 325
docker Swarm, 1115
Dockerfile instructions, 326–328
dockerizing applications, 326–331
hello-world containers, 322–325
installing, 318–320
Python orchestration, 376–378
REST API, 322
servers, 322
verifying, 320–322
documentation, rules of thumb, 1111
dollar signs ($) metacharacter, regular expressions (regex), 185, 187
dots (.)
  .* notation, regular expressions (regex), 190
  .. notation, Linux directories, 37
  regular expressions (regex), 185, 189
double asterisks (**), assignment operator, Python, 281
double dot notation (..), Linux directories, 37
double equal sign (==)
  conditional statements, 229
Jinja2, 1019
Python, 285
double semicolons (;;), case-in constructs, 233–234
DRY principle, 311, 336
DTD (Document Type Definition), 563
eexample of, 563–564
joint XML/DTD files, 564–565
dynamic data type allocation, Python, 271–272
dynamically setting Ansible variables, 1011–1013

E
eastbound API, 883, 933
echo command, Linux, 208–210, 222–226
<edit-config> operations, 712–719
elif statements, Python control flow, 297–298
encapsulation, OOP, 257
encoding protocols, 18
encryption
  AEAD, 495–496
  keys, 488–489
  asymmetric keys, 490
  DH protocol, 490–492
  ephemeral keys, 490
  generation/exchanges, 488–492
  HKDF, 492
  key exchange algorithm, 490
  KM, 492
  PSK, 490
  symmetric keys, 489
end tags, XML, 555
endings/beginnings of words, matching with regular expressions (regex), 188–189
endpoints
  API, 882
  collaboration endpoints, 943
    Cisco IP Phones, 944
    Cisco Webex Room Series, 944
  Enoch, Linux distributions, 26
  entity bodies, HTTP, 391
  entity header fields, HTTP, 416, 427–430
  EPEL repositories, 111–112
  ephemeral keys, 490
  equal sign (=)
    assignment operator, Python, 284
    conditional statements, 229
    double equal sign (==)
      conditional statements, 229
      Jinja2 operator, 1019
      Python operator, 285
  error reporting, RESTCONF, 746
  etc directory, Linux, 36
  event notifications, DNA Center, 935
  exit() function, Python, 272
  Expect programming language, Linux
    scripting, 245–246
  expressions, XPath, 575–576
    absolute path, 576
    absolute path and multiple outputs, 577
    anywhere selection, 576
    path definitions, 578
    predicates, 577
  external JSON schemas, referencing, 602–609

F

FCAPS (Fault, Configuration, Accounting, Performance, Security), 3

fdisk command, Linux, 121–125
fetching/uploading files via SFTP, 543
fields parameter, RESTCONF, 771, 773–777
file comparison operators, Bash shells, 230–232
file modules, Ansible, 1003
files
  Linux
    attributes, 40–41
    copying, 46–48
    creating, 46
    deleting, 47, 48
    moving, 47, 48
    permissions, 40
    removing, 47, 48
    renaming, 47, 48
    permissions, changing with SFTP, 544–545
  security management, Linux
    ACL, 148–155
    changing file users/groups, 146
    creating files under different groups, 147
    default users/groups of new files, 147
    logging into different groups, 147–148
    permissions, 143–146, 150–152
  file systems, Linux, 35–37
    creating file systems, 125–126
    mounting file systems, 126–128
    unmounting file systems, 127
  filter parameter, RESTCONF, 771
  filters, Ansible, 1013–1015
  finding sources of truth, 1110–1111
  findmnt command, Linux, 148–149
  Finesse (Cisco), 943, 946–947
firewalld service, 155–158
Flask, web/API development
  accessing in-memory employee data, 347–349
  installing in virtual environments, 345–346
  retrieving ID-based data, 349–350
  simple applications, 346–347
float() function, Python, 275
floating point data, Python, 275
flow control, Python
  elif statements, 297–298
  for loops, 301–302, 306
  nested for loops, 303–304
  range() function, 302–303
  if-else statements, 296–300
  nested code blocks with conditional statements, 295–296
  while loops, 304–306
for-do loops, 235–237
for loops, Python control flow, 301–302, 306
  nested for loops, 303–304
  range() function, 302–303
format() function, Python, 280
formatting
  strings, Python, 280
  XML, 561–562
forward slash (/)
  /: root directory, Linux, 36
  /= operator, Python, 284
  assignment operator, Python, 281
forwarded-tcpip channels, SSH
  Connection Protocol, 520–521
forwarding
  interfaces, Linux networking, 167
  SR, 832
frames, HTTP/2, 504–505
  binary message framing, 506–507
  multiplexing, 505–506
from-import statements, Python
  modules, 334–335
front end web/API development, 336
funcname() function, Python, 307
functions
  Linux scripting, 240–244
  Python, 306–307
    calling in code blocks, 307–309
    defining in code blocks, 307–309
    dictionary functions, 292
    representations, 307
GBP, compact GBP, 1113–1114
GCM (Galois/Counter Mode), 495
general header fields, HTTP, 416, 418–422
<get> operations, NETCONF, 702–703
<get-config> operations, NETCONF, 702–703
GET method
  HTTP client requests, 398
    Bash shells, 447–454
    Postman, 445–446
  request messages, 389–391, 416–417
  RESTCONF, 760–763
Get RPC, gNMI, 801–807
getfacl command, Linux, 149–155
Git
  commit phase, 317
  flexibility, 312
initialization commands, 314
performance, 312
pull phase, 317
push phase, 317
repositories, 312–313
cloning, 316–317
fetching remote repository updates, 314–315
setting up, 315–316
updating local repositories, 314–315
security, 312
server setup, 313–314
workflows, 317
gNMI (gRPC Network Management Interface), 798–799
anatomy of, 799–801
Capabilities RPC, 810–811
Get RPC, 801–807
insecure mode, IOS XE, 815
managing network elements, 814–818
network programmability, 961–962
Python
metaclasses, 815–816
sample Get script, 816–818
Set RPC, 807–810
Subscribe RPC, 811–814
GNU Bash Manual, 184
GNU C library (glibc), 24
GNU info files, 35
Google Chrome, LocalRepo repositories, 114–117
grep command, regular expressions (regex), 184–193
groupadd command, Linux, 141
groupdel command, Linux, 141–142
grouping nodes, YANG data modeling, 654–656
group/user management, Linux, 136–138
creating
groups, 141
new users, 138–141
deleting
groups, 141–142
users, 141
getting user information, 136
modifying user details, 142
passwords
changing, 141
setting user passwords, 138–141
gRPC (Google Remote Procedure Calls)
gNMI, 798–799
anatomy of, 799–801
Capabilities RPC, 810–811
Get RPC, 801–807
insecure mode, IOS XE, 815
managing network elements, 814–818
Python, metaclasses, 815–816
Python, sample Get script, 816–818
Set RPC, 807–810
Subscribe RPC, 811–814
history of, 782–784
networks
managing network elements, 814–818
programmability, 961–962
principles of, 782–784
Protobuf
example of, 788–789
in Python, 790–798
server sample, 794–797
as a transport, 784–786
guest shells
  IOS XE programmability, 885
  Open NX-OS, 884, 887, 891–892
GUI and Linux CLI, 28–29
gzip archiving utility, Linux, 67–68

handshake protocol, TLS 1.3, 499–502
hard disk partitions, /dev directory, 121–125
hard links, Linux, 51–55
hashbangs (#!), Linux shell scripting, 205–208
head and tail command, 45–46
HEAD method
  HTTP client requests, 398
  RESTCONF, 763
headers, HTTP, 389–391
  client request header fields, 422–425
  entity header fields, 416, 427–430
  general header fields, 416, 418–422
  overview of, 416–418
  request header fields, 416
  response header fields, 416
  server response header fields, 425–427
head-of-line blocking, 504
hello messages, NETCONF, 696–698, 973–974
hello-world containers, Docker, 322–325
help
  Ansible, 995–996
  Linux, 31–35
  SFTP, 541–542
hierarchical inheritance, OOP, 257
history command, 30
HKDF (HMAC-based Extract-and-Expand Key Derivation Function), 492
HMAC (Hash function-based MAC), 494
home directory, Linux, 36
host-based authentication, 517–518
hosting applications
  containerized application hosting, 1116
  iPerf, 1116
  native application hosting, 1115–1116
  rules of thumb, 1115–1116
HTTP (HyperText Transfer Protocol), 414
  Bash shells, 447–454
  client requests, 388–392, 397–398
    CONNECT method, 407
    DELETE method, 405–406
    GET method, 398
    GET method, Bash shells, 447–454
    GET method, Postman, 445–446
    HEAD method, 398
    header fields, 422–425
    OPTIONS method, 407–408
    PUT method, 402–405
    TRACE method, 408
  client/server connections, 394–395
  entity bodies, 391
  entity header fields, 427–430
  GET, request messages, 389–391, 416–417
headers, 389–391
  client request header fields, 422–425
  entity header fields, 416, 427–430
  general header fields, 416, 418–422
  overview of, 416–418
  request header fields, 416
  response header fields, 416
  server response header fields, 425–427
HTTP/1.1, 388, 504
  authentication, 469–471
  authentication, base64 encoding, 472, 474
  authentication, basic authentication, 472–474
  authentication, OAuth protocol, 474–483
  authentication, UTF-8 encoding, 472–473
  authentication, workflows, 470
  compression, 396–397
  persistent connections, 395–396
  pipelining, 396
HTTP/2, 503–504, 507–508
  frames, 504–505
  frames, binary message framing, 506–507
  frames, multiplexing, 505–506
  messages, 504–505
  streams, 505
HTTP/3, 504
HTTPS, 503
hyperlinks, 388
hypertext, 388
messages
  CRLF, 416, 418
  format of, 415
  headers, client request header fields, 422–425
  headers, entity header fields, 416, 427–430
  headers, general header fields, 416, 418–422
  headers, overview of, 416–418
  headers, request header fields, 416
  headers, response header fields, 416
  headers, server response header fields, 425–427
HTTP/2, 504–505
  start lines, 415
  whitespaces, 416
methods, 389
  CONNECT method, 407
  DELETE method, 405–406
  GET method, 398
  GET method, Bash shells, 447–454
  GET method, Postman, 445–446
  HEAD method, 398
  OPTIONS method, 407–408
  PUT method, 402–405
  TRACE method, 408
overview, 387–392
POST method
  Flask, 350–352
  RESTCONF, 740, 745–746
Postman, 436–437
  installing, 438
  interface, 438–441
Python
  requests packages, 464–467
  socket modules, 455–457
  TCP over Python, 455–457
  urllib packages, 458–463
REST, framework, 392–394
RESTCONF, 740
  HTTP headers, 745–746
  request messages, 740
  response messages, 740
RFC 2616, 416
RFC 3986, 393–394
RFC 6265, 484
RFC 6749, 475–476
RFC 6750, 475–476
RFC 7230, 416–417
RFC 7231, server status codes, 409–410
RFC 7540, 388
server status codes, 408–409
  1xx information status codes, 411
  2xx successful status codes, 411–412
  3xx redirection status codes, 412
  4xx server error status codes, 413–414
  5xx client error status codes, 414
  Cisco devices, 414
  RFC 7231, 409–410
socket modules, 455–457
state management with cookies, 483–487
TCP over Python, 455–457
transactions, 389
  client requests, 397–408
  server status codes, 408–414
as transfer protocol, 387
URI, 389, 431, 432–436
urllib packages, 458–463
versions of, 388
HTTPS (HTTP over TLS), 503
human operations, costs, 2
hyperlinks, 388
hypertext, 388

IaC (Infrastructure as Code), .
  See also NaC, 12
IBN (Intent-Based Networking)
  defined, 13–14
  intent assurance, 14
  intent fulfillment, 14
id command, Linux, 136
identity operators, Python, 284–285
IETF drafts, SDN, 823
IETF YANG modules, 670–671
if-else statements, Python control flow, 296–300
if-then constructs, Linux scripting, 226–232
import statements, Python modules, 333–334
importing Ansible variables from external files, 1007–1009
indexed arrays, 222–224
info command, 35
inheritance, OOP, 256–257
initialization commands, Git, 314
inode numbers, 40
input() function, Python, 298
input, Linux scripting, 213
  Arguments, bash script, 213–214
  read command, 214–217
input/output redirection, Linux, 57–59
  >> notation, 61–62
  piping (|), 58, 65–67
  stderr, 59, 62–65
  stdin, 59, 61
  stdout, 59, 61, 62–65
  tee command, 66–67
insecure mode, gNMI in IOS XE, 815
insert parameter, RESTCONF, 771
installing
  Ansible, 990
  Django, 337–338
  Docker, 318–320
  Flask in virtual environments, 345–346
  OpenSSH, CentOS installations, 522
  Postman, 438
  protoc on Python, 790–791
  Python, 260–263
  software on Linux, 94–96
  virtualenv tool, 331
instance identifiers, 850
int() function, Python, 275
integer comparison operators, Bash shells, 229–230
integer operations, Python, 275
Integration API, DNA Center, 935–936
Intent API, DNA Center, 934, 936–941
intent assurance, 14
intent fulfillment, 14
interface state (Linux), toggling, 161
interfaces, network programmability
  CLI programmable interfaces, 962–967
  Linux, 967–973
  NETCONF/YANG, 973–978
  RESTCONF/YANG, 978–987
interpreters, 28
inventories, Ansible
  default paths, 992
  IP addresses, 993–994
  simple inventory files, 992–993
inventory files, Ansible, 1009–1011
IOS XE
  Ansible
    clearing counters, 1060–1061
    configuring IOS XE, 1061–1069
    configuring IOS XE with ios_* modules, 1069–1073
    configuring IOS XR with iosxr_* modules, 1083–1084
    preparing IOS XE for Ansible management, 1055–1057
    preparing IOS XE for NETCONF management, 1095–1096
    updating files with additional hosts/variables, 1057–1058
    verifying IOS XE for Ansible management, 1057
    verifying operational data, 1058–1060
  gNMI, insecure mode, 815
  Jinja2 templates, 367–371
  NETCONF, 918–922
  programmability, 885–886
  SSH setup, 526–531, 545–546
JSD (JSON Schema Definition) 1147

IOS XR
   Ansible
      configuring IOS XR, 1078–1084
      preparing for management, 1073–1074
      preparing IOS XR for NETCONF management, 1096–1098
      verifying IOS XR for Ansible management, 1074–1075
      verifying operational data, 1074–1078
   Jinja2 templates, 367–371
      NETCONF, 916–918
      PCEP configurations, 867–880
      programmability, 886–887
      SSH setup, 532–536, 546–547
   ios_command module, Ansible
      clearing counters, 1060–1061
      verifying operational data, 1058–1060
   iosxr_command, Ansible
      input parameters, 1001
      playbooks, 997–999
      verifying operational data, 1074–1078
   IP addresses
      adding to/removing from Linux interfaces, 161–162
      Ansible inventories, 993–994
   ip command, Linux, 158–167
   IP Phones (Cisco), 944
   IP VPN, JSON schemas, 597–598, 601–602, 607–609
   iPerf, application hosting, 1116
   IS-IS IGP, SR configuration, 825–827
   is not operator, Python, 285
   is operator, Python, 285
   ITIL 4 management practices, 3–5
   J

Jinja2
   != operator, 1019
   < operator, 1019
   <= operator, 1019
   > operator, 1019
   >= operator, 1019
   comparison operators, 1018–1019
   Python functions, 1049
      join() function, 1050–1051
      map() function, 1054–1055
      split() function, 1051–1054
   templates, 363–364, 1034–1040
      conditional statements, 1045–1049
      IOS XE, 367–371
      IOS XR, 367–371
      loops, 1040–1043
      NAPALM libraries, 371–375
      Netmiko libraries, 364–367
      Nornir libraries, 367–369, 371–375
      NX-OS, 367–371
      nx-os_config modules, 1091–1093
      playbooks, 1043–1045
      variables, 1042–1043
      YAML and, 635–637
   jobs, Linux, 74
      displaying status, 80
      stopping, 80–81
   join() function, 1050–1051
   joining tuples, Python, 293
   joint XML/DTD files, 564–565
   journald command, Linux, 93–94
   JSD (JSON Schema Definition), 595
JSON (JavaScript Object Notation)
arrays, 593
Boolean data, 593
characteristics, 591
data example, 591–592
data format, 592–594
data types, 592–594
defined, 591
JSD, 595
JTOX drivers and pyang, 683–687
null values, 593
numbers, 593
objects
key/pair values, 592
out of range objects and data validation, 613–614
referencing external JSON schemas, 606–607
repetitive objects, 598–602
simple JSON object example, 592
typos and data validation, 611–612
unexpected keys and data validation, 612–613
schemas
annotation, 596
basic schema without content, 596–597
data validation, 609–614
definitions in, 598–601
IP VPN, 597–598, 601–602, 607–609
JSD, 595
keywords, 596
properties of, 597–598
purpose of, 595–596
referencing external schemas, 602–609

repetitive objects, 598–602
structure of, 595–598
validation keywords, 596–597
strings, 593
YAML versus, 615–616
JTOX drivers and pyang, 683–687

K

kernels, Linux
kernel space, 23–24
LKM, 25
microkernel kernels, 25
monolithic kernels, 25
SCI, 24
key/pair values, JSON objects, 592
keys
authentication, SSH, 523–525
encryption, 488–489
asymmetric keys, 490
DH protocol, 490–492
ephemeral keys, 490
generation/exchanges, 488–492
HKDF, 492
key exchange algorithm, 490
KM, 492
PFS, 490
PSK, 489
symmetric keys, 489
RSA keys, Linux network programmability, 971–973
keywords, JSON schemas, 596
kill command, Linux system maintenance, 78–80, 81
<kill-session> operations, NETCONF, 721–722
KM (Keying Material), 492
Kubernetes, 378–382, 1115
lab topologies
  BGP-LS, 845–846
  PCEP, 867
  SR, 825
lastlog command, Linux, 91
leading spaces, XML documents, 555
leaf nodes, YANG data modeling, 649–651
leaf-list nodes, YANG data modeling, 651–652
left arrow (<)
  Jinja2 operator, 1019
  Python operator, 285
left arrow, equal sign (<=)
  Jinja2 operator, 1019
  Python operator, 285
left shift operator (<<), Python, 281
less command, 43–44
lexical analyzers, Python code execution, 263
libraries
  NAPALM libraries, 354–359
  Netmiko libraries, Jinja2 templates, 364–367
  Nornir libraries, 354–359, 371–375
libsolv, 95
link NLRI, BGP-LS, 856–857
links, Linux
  hard links, 51–55
  soft links, 55–57
  symlinks, 56–57
Linux, 16–17, 21, 25
  >> notation, 61–62
  ACL, 148–155
  API, 24
applications, communication, 24
architecture, 23–25
archiving utilities
  bzip2, 67, 69
  gzip, 67–68
  tar, 67, 70–73
  xz, 67, 69–70, 72–73
Bash shells, 29
  IOS XR programmability, 886
  Open NX-OS, 886, 887–891
BIOS, 27
blkid command, 148–149
boot process, 26–28
bzip2 archiving utility, 67, 69
cat command, 41–42, 61–62
cat/proc/cpuinfo command, 87–88
chmod command, 144–146
clear command, 30
command shell, 28–30
comments in scripts, 207–208
compiling software, 94, 96–97
cp command, 46–48, 49–51
daemons, 24
date command, 85–86
depsolve, 95
/dev directory, storage, 119–120
  contents of, 120
  device file types, 120–121
  fdisk command, 121–125
  file system creation, 125–126
  hard disk partitions, 121–125
  mkfs command, 125–126
  mounting file systems, 126–128
  unmounting file systems, 127
development, 22
device drivers, 23
device mappers, 135
dig command, 180–181
directories, 48–49
   /: root directory, 36
   absolute paths, 38–39
   attributes, 40–41
   boot directory, 36
copying, 49–51
creating, 49, 51
deleting, 50–51
dev directory, 36
double dot notation (..), 37
etc directory, 36
home directory, 36
media directory, 37
mnt directory, 37
moving, 49, 51
navigating, 38–41
opt directory, 37
proc directory, 37
relative paths, 38–39
renaming, 50–51
root directory, 37
run directory, 37
srv directory, 37
sys directory, 37
tmp directory, 37
usr directory, 37
usr/bin directory, 37
usr/local directory, 37
usr/sbin directory, 37
var directory, 37
distributions, 26
dmesg command, 90
dmidecode command, 88–89
DNF, 95, 117
double dot notation (..), Linux
directories, 37
echo command, 208–210, 222–226
EPEL repositories, 111–112
fdisk command, 121–125
file security management
   ACL, 148–155
   changing file users/groups, 146
   creating files under different
groups, 147
   default users/groups of new
   files, 147
   logging into different groups,
   147–148
   permissions, 143–148, 150–
   152
file systems, 35–37
   creating, 125–126
   mounting, 126–128
   unmounting, 127
files
   attributes, 40–41
   creating, 46
   permissions, 40
   viewing, 41–46
findmnt command, 148–149
getfacl command, 149–155
GNU C library (glibc), 24
GNU info files, 35
grep command, regular expressions
   (regex), 184–193
groupadd command, 141
groupdel command, 141–142
Guest shells, Open NX-OS, 884, 886,
   887, 891–892
gzip archiving utility, 67–68
hard links, 51–55
head and tail command, 45–46
help, 31–35
history command, 30
history of, 21–22
host files and Ansible, 993
id command, 136
info command, 35
inode numbers, 40
interpreters, 28
ip command, 158–167
jobs, 74
displaying status, 80
stopping, 80–81
journald command, 93–94
kernels
kernel space, 23–24
LKM, 25
microkernel kernels, 25
monolithic kernels, 25
SCI, 24
kill command, 78–80, 81
lastlog command, 91
less command, 43–44
libsolv, 95
links
hard links, 51–55
soft links, 55–57
symlinks, 56–57
ls command, 31–34, 39, 143–144
 lspci command, 90
 lvdisplay command, 132–133
LVM, 128–135
man command, 34
MBR, 27
mkdir command, 49, 51, 134–135
mkfs command, 125–126, 133–134
more command, 42–43, 44
mv command, 47, 48, 50–51
networking, 158–159
adding/removing IP addresses from interfaces, 161–162
configuration files, 174–179
DNS, 179–181
ip command, 158–167
NetworkManager, 168–174
programmability, 960, 967–973
routing tables, 163–168
scripting, 174–179
sysctl command, 167–168
toggling interface state, 161
newgrp command, 138, 146–148
package management software, 95
DNF, 95, 117
RPM, 95, 97–101
YUM, 95, 101–117
patches, 22
pgrep command, 78
pinfo command, 35
piping (|), 58, 65–67
POST, 27
printf command, 210–213
processes, 73–74, 80–81
ps command, 74–77
pvdisplay command, 130–131, 133
pwd command, 29, 138–140
read command, 214–217
redirecting input/output, 57–59
>> notation, 61–62
piping (|), 58, 65–67
stderr, 59, 62–65
stdin, 59
stdin, sort command, 59–61
stdout, 59, 62–65
stdout, sort command, 61
tee command, 66–67
resource utilization, 83–85
rm command, 47, 48
rmdir command, 50–51
RPM, 95, 97–101
RSA keys, network programmability, 971–973
ryslogd command, 91–93
scale command, 221–222
scripting, 183
Arguments.bash script, 213–214
arithmetic operations, 220–222
arrays, adding/removing elements, 224–226
arrays, associative arrays, 222
arrays, concatenating, 221–226
arrays, declaring, 222–224
arrays, defined, 222
arrays, indexed arrays, 222–224
awk programming language, 194–197
Bash scripting, 184
Bash shells, 206–207, 213–214
Bash shells, Arguments.bash script, 213–214
case-in constructs, 232–234
comments, 207–208
conditional statements, 226
conditional statements, case-in constructs, 232–234
conditional statements, double equal sign (==), 229
conditional statements, equal sign (=), 229
conditional statements, if-then constructs, 226–232
Expect programming language, 245–246
functions, 240–244
GNU Bash Manual, 184
if-then constructs, 226–232
input, 213
input, Arguments.bash script, 213–214
input, read command, 214–217
loops, 234–235
loops, for-do loops, 235–237
loops, until-do loops, 239–240
loops, while-do loops, 237–239
networks, 174–179
output, 208
output, echo command, 208–210
output, POSIX, 210
output, printf command, 210–213
regular expressions (regex), 184–193
scale command, 221–222
sed command, 197–205
shells, 183
shells, structure of, 205–208
variables, 217–218
variables, declaring, 218–219
variables, one-dimensional variables, 218
variables, value assignments, 218–219
security, 135
ACL, 148–155
file security management, 143–148
system security, 155–158
user/group management, 136–142
sed command, 197–205
services, 74, 81–83
setfacl command, 151–155
sg command, 146–147
shells, 183, 205–208
shred command, 48
soft links, 55–57
software
  dependencies, 95
  installation/maintenance, 94–96
software repositories, 95–96
  EPEL repositories, 111–112
  listing, 110–111
  LocalRepo repositories, 112–117
sort command, stdin, 59–61
source code, software installation, 94
SSH, network programmability, 971–973
stat command, 62–63
stderr, 59, 62–65
stdin, 59, 61
stdout, 59, 61, 62–65
storage, 119
  /dev directory, 119–128
  LVM, 128–135
  physical storage, 119–128
su command, 29–30
sudo command, 136
sudo yum remove httpd command, 109
sudo yum update command, 110
symlinks, 56–57
sysctl command, 167–168
syslog messages, 93–94
system calls, 24
system daemons, 24
system information
cat/proc/cpuinfo command, 87–88
date command, 85–86
dmesg command, 90
dmidecode command, 88–89
dlpci command, 90
timedatectl command, 86
update command, 86–87
system logs, 91
  journald command, 93–94
  lastlog command, 91
  rotation, 91
  rsyslogd command, 91–93
  tail command, 91
system maintenance, 73
  jobs, 74, 80–81
  kill command, 78–80, 81
  pgrep command, 78
  processes, 73–74, 80–81
  ps command, 74–77
  services, 74, 81–83
  systemctl command, 81–83
  threads, 73
system security, 155–158
systemctl command, 81–83
tar archiving utility, 67, 70–73
tee command, 66–67
Terminal, 29
threads, 73
time stamps, 41
timedatectl command, 86
today, 22
top command, 83–85
touch command, 46, 48
update command, 86–87
usage, 22
user space (userland), 23–24
useradd command, 138–140
userdel command, 141
user/group management, 136–138
  changing passwords, 141
creating groups, 141
creating new users, 138–141
deleting groups, 141–142
deleting users, 141
getting user information, 136
modifying user details, 142
setting user passwords, 138–141
usermod command, 142
vgdisplay command, 132, 133–134
volumes
  logical volumes, 132–133
  physical volumes, 130–131
  volume groups, 132, 133–134
  mounting, 134–135
xz archiving utility, 67, 69–70, 72–73
YUM, 95, 101
  commands list, 102–103
  sudo yum remove httpd command, 109
  sudo yum update command, 110
  yum info command, 104–105
  yum install command, 106–109
  yum list command, 103
  yum repolist all command, 110–111
  yum search command, 103–104
  yum-config-manager command, 111–112
  list nodes, YANG data modeling, 653–654
  list variables, Ansible, 1007
  list.append() function, Python, 288–289
  list.pop() function, Python, 288–289
  list.reverse() function, Python, 289
  lists, Python, 286–287
    accessing data/operations, 287–288
    functions, 288–289
    if-else statements, 298–299
    nested lists, 289–290
    slicing, 286–287
    square brackets ([ ]), 286–287
    value assignments, 286–288
  list.sort() function, Python, 289
LKM, Linux, 25
local Git repositories, updating, 314–315
local/remote directory operations, SFTP, 542–543
LocalRepo repositories, 112–117
Location Scanning API (Meraki), 923
<lock> operations, NETCONF, 720–721
logical AND/OR, combining multiple conditional statements, 1022–1024
logical operators
  Python, 284–285, 297–298
  XPath values, 577
logins, SFTP, 541
loopback interfaces
  CLI programmability, 962–963
  NETCONF programmability, 975–978
loops
  Ansible
    conditional statements with loops, 1024–1027
    Jinja2 templates, 1040–1043
for loops, Python control flow, 301–302, 306
  nested for loops, 303–304
  range() function, 302–303
Jinja2 templates, 1040–1043
Linux scripting, 234–235
  for-do loops, 235–237
  until-do loops, 239–240
  while-do loops, 237–239
nx-os_config modules, 1090–1091
while loops, 304–306
lowercase/uppercase characters (regular expressions), matching, 187–188
ls command, 31–34, 39, 143–144
LSP, PCEP
  PCC, 874–876, 879–880
  PCE, 876–880
lspci command, Linux, 90
lvdisplay command, Linux, 132–133
LVM (Logical Volume Manager), 128–135

M
MAC (Message Authentication Code), 493–494
machine learning, Python, 378–382
man command, 34
managing
  CMDB, 12
  configuration management automation
    IOS XE programmability, 886
    Open NX-OS programmability, 885
devices, DNA Center, 934
file security management, Linux
  ACL, 148–155
changing file users/groups, 146
creating files under different groups, 147
default users/groups of new files, 147
logging into different groups, 147–148
permissions, 143–148, 150–152
IOS XE with Ansible
  clearing counters, 1060–1061
  NETCONF, 1095–1096
  preparing for management, 1055–1057
  preparing for NETCONF management, 1095–1096
  updating files with additional hosts/variables, 1057–1058
  verifying management, 1057
  verifying operational data, 1058–1060
IOS XR with Ansible
  NETCONF, 1096–1098
  preparing for management, 1073–1074
  preparing for NETCONF management, 1096–1098
  verifying operational data, 1074–1078
ITIL 4 management practices, 3–5
NETCONF, verifying operational data, 1098–1103
networks
  defined, 3–5
  device management with
    NETCONF/YANG, 975–978
FCAPS, 3
NX-OS with Ansible
  collecting show output with
    nxos_command, 1086–1088
configuring with nx-os_*
modules, 1093–1095
configuring with nx-os_config
modules, 1086–1088
interactive commands,
1088–1089
NETCONF, 1098
preparing for management,
1084–1085
preparing for NETCONF
management, 1098
verifying management,
1085–1086
verifying operational data,
1086–1089
package management software, 95
  DNF, 95, 117
  RPM, 95, 97–101
  YUM, 95, 101–117
state management, HTTP, 483–487
user/group management, Linux,
  136–138
  changing passwords, 141
  creating groups, 141
  creating new users, 138–141
  deleting groups, 141–142
  deleting users, 141
  getting user information, 136
  modifying user details, 142
  setting user passwords,
  138–141
manual software compilation/
installation, Linux, 96–97
map() function, 1054–1055
mappings, YAML, 618–620
matching
  anything/everything with .* notation,
  regular expressions (regex), 190
  beginnings/endings of words,
  regular expressions (regex),
  188–189
  uppercase/lowercase characters,
  regular expressions (regex),
  187–188
MBR (Master Boot Records), 27
media directory, Linux, 37
Meeting Server (Cisco), 943
membership operators, Python,
  285–286
Meraki API, 922, 923
  Captive Portal API, 923
  Dashboard API, 922–931
  Location Scanning API, 923
  MV Sense API, 923
  Webhook Alerts API, 922
merge keys, YAML, 624–625
messages, HTTP
  CRLF, 415, 418
  format of, 415
  headers
    client request header fields,
    422–425
    entity header fields, 416,
    427–430
    general header fields, 416,
    418–422
    overview of, 416–418
    request header fields, 416
    response header fields, 416
    server response header fields,
    425–427
HTTP/2, 504–505
MAC, 493–494
start lines, 415
whitespaces, 416
messages layer
NETCONF, 693, 695–696
  hello messages, 696–698
  rpc messages, 698–699
  rpc-reply messages, 699–701
RESTCONF, 742–743, 759
  constructing messages, 745
  content parameter, 771
  DELETE method, 770–771
  depth parameter, 771
  editing data, 759–763
  error reporting, 746
  fields parameter, 771, 773–777
  filter parameter, 771
  GET method, 760–763
  HEAD method, 763
  HTTP headers, 745–746
  insert parameter, 771
  OPTIONS method, 759–760
  PATCH method, 767–770
  point parameter, 771
  POST method, 763–765
  PUT method, 765–767
  query parameters, 771–777
  request messages, 743–744
  response messages, 744–745
  retrieving data, 759–763
  start-time parameter, 771
  stop-time parameter, 771
  with-defaults parameter, 771
methods, HTTP, 389
  CONNECT method, 407
  DELETE method, 405–406
  GET method, 398
    Bash shells, 447–454
    Postman, 445–446
  HEAD method, 398
  OPTIONS method, 407–408
  POST method, 399–402
    Postman, 443–445
    Python and HTTP, 465
  PUT method, 402–405
  TRACE method, 408
microkernel kernels, Linux, 25
microservice architectures, 782
migrations, Django, 338–339
minus sign (-)
  -= operator, Python, 284
  assignment operator, Python, 281
mkdir command, 49, 51, 134–135
mkfs command, Linux, 125–126, 133–134
mnt directory, Linux, 37
model-based industry-standard API
  IOS XE programmability, 885
  IOS XR programmability, 886
  Open NX-OS programmability, 884
model-driven telemetry
  rules of thumb, 1113–1114
  SNMP, 1113
  syslog, 1113
modeling data, YANG, 639–640, 642
  importance of, 640–642
  modules, 642–644
    augmentation, 656–658
    built-in data types, 647–648
    cloning, 665
    derived data types, 648–649
    deviations, 658–662
    home of, 664–666
    IETF YANG modules, 670–671
    native (vendor-specific) modules, 666–669
<table>
<thead>
<tr>
<th>Key Term</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenConfig YANG modules</td>
<td>671–673</td>
</tr>
<tr>
<td>structure of</td>
<td>644–646</td>
</tr>
<tr>
<td>verifying downloaded modules</td>
<td>665–666</td>
</tr>
<tr>
<td>nodes</td>
<td>649</td>
</tr>
<tr>
<td>container nodes</td>
<td>647–648</td>
</tr>
<tr>
<td>grouping</td>
<td>654–656</td>
</tr>
<tr>
<td>leaf nodes</td>
<td>649–651</td>
</tr>
<tr>
<td>leaf-list nodes</td>
<td>651–652</td>
</tr>
<tr>
<td>list nodes</td>
<td>647–648</td>
</tr>
<tr>
<td>pyang</td>
<td>673–679, 683–687</td>
</tr>
<tr>
<td>pyangbind</td>
<td>679–682</td>
</tr>
<tr>
<td>YANG 1.1</td>
<td>662–663</td>
</tr>
<tr>
<td>models.py files</td>
<td>343</td>
</tr>
<tr>
<td>modifying user details, Linux user/group management</td>
<td>142</td>
</tr>
<tr>
<td>Ansible</td>
<td></td>
</tr>
<tr>
<td>control functions</td>
<td>1001–1002</td>
</tr>
<tr>
<td>debug modules</td>
<td>999</td>
</tr>
<tr>
<td>file modules</td>
<td>1003</td>
</tr>
<tr>
<td>ios_command module</td>
<td>1058–1061</td>
</tr>
<tr>
<td>iosxr_command, parameters</td>
<td>1001</td>
</tr>
<tr>
<td>network modules</td>
<td>1003</td>
</tr>
<tr>
<td>return values</td>
<td>1001–1002</td>
</tr>
<tr>
<td>structure of</td>
<td>1000</td>
</tr>
<tr>
<td>utility modules</td>
<td>1003</td>
</tr>
<tr>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>application development</td>
<td>333–336</td>
</tr>
<tr>
<td>classes</td>
<td>335–336</td>
</tr>
<tr>
<td>constructors</td>
<td>335–336</td>
</tr>
<tr>
<td>from-import statements</td>
<td>334–335</td>
</tr>
<tr>
<td>import statements</td>
<td>333–334</td>
</tr>
<tr>
<td>YANG data modeling</td>
<td>642–644</td>
</tr>
<tr>
<td>augmentation</td>
<td>656–658</td>
</tr>
<tr>
<td>built-in data types</td>
<td>647–648</td>
</tr>
<tr>
<td>cloning</td>
<td>665</td>
</tr>
<tr>
<td>derived data types</td>
<td>648–649</td>
</tr>
<tr>
<td>deviations</td>
<td>658–662</td>
</tr>
<tr>
<td>home of</td>
<td>664–666</td>
</tr>
<tr>
<td>IETF YANG modules</td>
<td>670–671</td>
</tr>
<tr>
<td>native (vendor-specific) modules</td>
<td>666–669</td>
</tr>
<tr>
<td>OpenConfig YANG modules</td>
<td>671–673</td>
</tr>
<tr>
<td>structure of</td>
<td>644–646</td>
</tr>
<tr>
<td>verifying downloaded modules</td>
<td>665–666</td>
</tr>
<tr>
<td>Modulo Operator (%)</td>
<td>274, 281</td>
</tr>
<tr>
<td>monitoring networks, streaming telemetry</td>
<td>1113–1114</td>
</tr>
<tr>
<td>monolithic kernels, Linux</td>
<td>25</td>
</tr>
<tr>
<td>more command</td>
<td>42–43, 44</td>
</tr>
<tr>
<td>mounting</td>
<td></td>
</tr>
<tr>
<td>file systems</td>
<td>126–128</td>
</tr>
<tr>
<td>volumes, LVM</td>
<td>134–135</td>
</tr>
<tr>
<td>moving</td>
<td></td>
</tr>
<tr>
<td>directories, Linux</td>
<td>49, 51</td>
</tr>
<tr>
<td>files, Linux</td>
<td>47, 48</td>
</tr>
<tr>
<td>MPLS FIB, SR</td>
<td>828–830</td>
</tr>
<tr>
<td>MPLS LSP, PCEP</td>
<td>864–867</td>
</tr>
<tr>
<td>multi-level inheritance, OOP</td>
<td>257</td>
</tr>
<tr>
<td>multiple inheritance, OOP</td>
<td>257</td>
</tr>
<tr>
<td>multiplexing frames, HTTP/2</td>
<td>505–506</td>
</tr>
<tr>
<td>mv command</td>
<td>47, 48, 50–51</td>
</tr>
<tr>
<td>MV Sense API (Meraki)</td>
<td>923</td>
</tr>
</tbody>
</table>
NaC (Network as Code), 12
namespaces, XML, 559–561
NAPALM libraries, network
automation, 354–359, 371–375
native (vendor-specific) YANG
modules, 666–669
native application hosting, 1115–1116
ncclient, NETCONF, 735–739
negative/positive values, arithmetic
operators, Python, 282
nested code blocks with conditional
statements, Python control flow,
295–296
nested for loops, 303–304
nested lists, Python, 289–290
nesting format, XML, 561–562
nesting relationships, XML, 555–556
NETCONF, 689
Ansible and
configuring, 1103–1107
IOS XE management, 1095–1096
IOS XR management,
1096–1098
NX-OS management, 1098
verifying operational data,
1098–1103
architecture, 692–693
authentication, 694
<cancel-commit> operations,
722–724
candidate configuration operations,
722–724
capabilities, 731
candidate configuration
capability, 732
confirmed commit capability,
732
distinct startup capability, 733
rollback-on-error capability,
732–733
URL capability, 733–734
validate capability, 733
writable-running capability, 732
XPath capability, 735
<close-session> operations, 721–722
<commit> operations, 722–724
configuration validation, 724–725
connections, 694
content layer, 693, 725–730
YANG, 725–729
<copy-config> operations, 719
data delivery, 694
data integrity/confidentiality, 694
datastores
configuring, 712–720
operations, 720–721
<delete-config> operations, 719–720
<discard-changes> operations,
722–724
<edit-config> operations, 712–719
<get> operations, 702–703
<get-config> operations, 702–703
hello messages, 696–698, 973–974
high-level operations, 691–692
IOS XE, 918–922
IOS XR, 916–918
<kill-session> operations, 721–722
<lock> operations, 720–721
messages layer, 693, 695–696
hello messages, 696–698
rpc messages, 698–699
rpc-reply messages, 699–701
ncclient, 735–739
netconf_get module
configuring, 1103–1107
verifying operational data, 1098–1103
NETCONF/YANG, network programmability, 973–978
network programmability, 960, 973–978
NX-OS, 905–916
operations layer, 693, 701–702
<cancel-commit> operations, 722–724
candidate configuration operations, 722–724
<close-session> operations, 721–722
<commit> operations, 722–724
configuration validation, 724–725
<copy-config> operations, 719
datastore configurations, 712–720
datastore operations, 720–721
<delete-config> operations, 719–720
<discard-changes> operations, 722–724
<edit-config> operations, 712–719
<get> operations, 702–703
<get-config> operations, 702–703
<kill-session> operations, 721–722
<lock> operations, 720–721
session operations, 721–722
subtree filters, 703–710
<unlock> operations, 720–721
<validate> operations, 724–725
XPath filters, 710–712
over SSH, 694–695
overview, 689–692
Python, ncclient, 735–739
reliability, 694
RESTCONF and, 740–742
rpc messages, 691, 698–699
rpc-reply messages, 691, 699–701
session operations, 721–722
transport layer, 692–693
NETCONF over SSH, 694–695
transport protocol requirements, 693–694
<unlock> operations, 720–721
<validate> operations, 724–725
Working Group, 689–690
XML attributes, 558
YANG, 725–729
NETCONF/YANG
configuring, 1103–1107
netconf_get module, 1103–1107
verifying operational data, 1098–1103
Netmiko libraries, Jinja2 templates, 364–367
network modules, Ansible, 1003
NetworkManager, 168–169
connections
attributes, 169–170
creating, 171–174
deleting, 171–174
listing, 169
interfaces, listing, 171
networks
abstraction, defined, 9–13
Ansible. See also separate entry, 15–16
automation
  architectures, 353–354
  Jinja2 templates, 363–375
  NAPALM libraries, 354–359, 371–375
CAPEX, 2
cleaning up, 1110
engineers
  automation, 19–20
  career paths, questions, 1118–1119
FCAPS, 3
IBN, 13–14
IP VPN, JSON schemas, 597–598, 601–602, 607–609
Linux, 16–17, 158–159
  adding/removing IP addresses from interfaces, 161–162
  configuration files, 174–179
  DNS, 179–181
  ip command, 158–167
  NetworkManager, 168–171
  routing tables, 163–168
  scripting, 174–179
  sysctl command, 167–168
  toggling interface state, 161
managing
  defined, 3–5
  devices with NETCONF/YANG, 973–978
  devices with RESTCONF/YANG, 978–987
FCAPS, 3
monitoring, streaming telemetry, 1113–1114
Network Programmability and Automation toolbox, 14–15
  Ansible. See also separate entry, 15–16
  Linux, 16–17
  protocols, 18–19
  Python, 15
  virtualization, 17
  YANG, 17
OPEX, 2
overlay networks, 9, 821
programmability, 253–254
  CLI, 958–959, 962–967
  gNMI, 961–962
  gRPC, 961–962
  Linux shells, 960, 967–973
  NETCONF/YANG, 973–978
  network programmability, 960
  RESTCONF, 961
  RESTCONF/YANG, 978–987
  SNMP, 959–960
  vendor/API matrix, 957–958
protocols, 18
  encoding, 18
  transport protocols, 18–19
Python, 15
SDN, 13, 819
  BGP-LS, 843
  BGP-LS, lab topologies, 845–846
  BGP-LS, link NLRI, 856–857
  BGP-LS, node NLRI, 854–855
  BGP-LS, NPF-XR, 845, 849, 851–854
  BGP-LS, peering, 843–844, 847–849
  BGP-LS, prefix NLRI, 858–859
  BGP-LS, routing, 846–847
BGP-LS, routing types
(overview), 850–854
controllers, 821–823
network requirements,
819–821
overlay networks, 821
PCEP, call flows, 861–864
PCEP, IOS XR configurations,
867–880
PCEP, LSP delegation, 864–867
PCEP, PCC, 860–861, 867–870,
874–880
PCEP, PCE, 860–861, 869–874,
876–880
PCEP, RFC, 860–861
PCEP, state synchronization, 863
SR, Adj-SID, 824, 827, 839–842
SR, data plane verification,
830–831
SR, forwarding, 832
SR, IETF drafts, 823
SR, lan topologies, 825
SR, MPLS FIB for NPF-XR,
828–830
SR, Node-SID, 824
SR, NPF-XR, 835
SR, Prefix-SID, 824, 827,
834–836
SR, segments, 824, 825–827
SR, SR algorithm, 827
SR, SRGB, 824, 827, 831
SR, SRLB, 824
SR, SR-MPLS, 824
SR, SR-TE, 832–843
underlay networks, 821–822
underlay networks, 9, 821–822
virtualization, 17
VLAN, awareness, 8–9
VPN, IP VPN, 597–598, 601–602,
607–609
YANG, 17
Neutron ML2 (OpenStack), Open
NX-OS programmability, 885
newgrp command, Linux, 138,
146–148
Nexus switches
authentication, 401–402, 463
static routing
DELETE method, 405–406
POST method, 399–401
PUT method, 402–405
NLRI, BGP-LS
link NLRI, 856–857
node NLRI, 854–855
prefix NLRI, 858–859
nodes
XPath, 574
YAML, 617
YANG data modeling, 649
container nodes, 647–648
grouping, 654–656
leaf nodes, 649–651
leaf-list nodes, 651–652
list nodes, 647–648
Node-SID, 824
Nornir libraries, network automation,
northbound API, 883, 933
not in operator, Python, 286
NOT operator (~), Python, 281, 285
notifications, DNA Center, 935
NPF-XR
BGP-LS, 845, 849, 851–854
SR, 835
SR-TE, 836–843
NSX, 13
null values, JSON, 593
number data types, Python, 273–276
numbers, JSON, 593

NX-API CLI
Open NX-OS programmability, 884
use cases, 893–898

NX-API REST
Open NX-OS programmability, 884
use cases, 898–905

NX-OS
Ansible
collecting show output with
nxos_command, 1086–1088
configuring with nx-os_ modules, 1093–1095
configuring with nx-os_config
modules, 1090–1093
interactive commands, 1088–1089
preparing for management, 1084–1085
preparing NX-OS for
NETCONF management, 1098
verifying management, 1085–1086
verifying operational data, 1086–1089
Bash shells, 887–891
Guest shells, 887, 891–892
Jinja2 templates, 367–371
NETCONF, 905–916

nxos_command
collecting show output, 1086–1088
configuring with nx-os_ modules, 1093–1095
configuring with nx-os_config
modules, 1086–1088
interactive commands, 1088–1089

nx-os_config modules
configuring NX-OS, 1086–1088
Jinja2 templates, 1091–1093
loops, 1090–1091
variables, 1090–1091
programmability, 884–885
SSH setup, 537–540, 547–548
use cases, 887–892

OAuth protocol, HTTP/1.1
authentication, 474–475
access tokens, 481–483
API resource server calls, 483
authorization grants, 477–481
Bearer Tokens, 475–476
client registration, 476–477
workflows, 476

Objects
JSON objects
defined, 593
key/pair values, 592
out of range objects and data
validation, 613–614
referencing external JSON
schemas, 606–607
repetitive objects, 598–602
simple JSON object example, 592
typos and data validation, 611–612
unexpected keys and data
validation, 612–613
Python objects, serializing with YAML, 628–629
octal notation, file permissions, 145–146
one-dimensional variables, 218
one-time automations, 1111
OOP (Object-Oriented Programming), 256
  abstraction, 257
  encapsulation, 257
  inheritance, 256–257
  polymorphism, 257
OpenConfig YANG modules, 671–673
OpenFlow, IOS XE programmability, 886
Open NX-OS
  Bash shells, 887–891
  NETCONF, 905–916
  programmability, 884–885
  use cases, 887–892
OpenSSH, 510, 522
OpenStack Neutron ML2, Open NX-OS programmability, 885
operations layer,
  NETCONF, 693, 701–702
    <cancel-commit> operations, 722–724
    candidate configuration operations, 722–724
    <close-session> operations, 721–722
    <commit> operations, 722–724
    configuration validation, 724–725
    <copy-config> operations, 719
    datastores, 712–721
    <delete-config> operations, 719–720
    <discard-changes> operations, 722–724
    <edit-config> operations, 712–719
    <get> operations, 702–703
    <get-config> operations, 702–703
    <kill-session> operations, 721–722
    <lock> operations, 720–721
    session operations, 721–722
    subtree filters, 703–710
    <unlock> operations, 720–721
    <validate> operations, 724–725
    XPath filters, 710–712
RESTCONF, 743
  operations resource, RESTCONF, 756–758
OR operator (|)
  Python, 281, 285
  regular expressions (regex), 186
operators
  Python, 281
    arithmetic operators, 281–283
    assignment operators, 284
    bitwise operators, 281–283
    comparison operators, 284–285
    identity operators, 284–285
    logical operators, 284–285, 297–298
    membership operators, 285–286
XPath operator values, 577
OPEX (Operating Expenses), 2
opt directory, Linux, 37
OPTIONS method, 407–408, 759–760
PCEP (Path Computation Element Protocol) 1165

orchestration, 375–376
  automation versus, 6–7
defined, 6–7
  Docker, 376–378
  Kubernetes, 378–382
tools (overview), 7
output/input redirection, Linux, 57–59
  >> notation, 61–62
  piping (|), 58, 65–67
  stderr, 59, 62–65
  stdin, 59, 61
  stdout, 59, 61, 62–65
  tee command, 66–67
output, Linux scripting, 208
  echo command, 208–210
  printf command, 210–213
overlay networks, 9, 821

P

package management software
  DNF, 95, 117
  RPM, 95, 97–101
  YUM, 95, 101
    commands list, 102–103
    sudo yum remove httpd command, 109
    sudo yum update command, 110
    yum info command, 104–105
    yum install command, 106–109
    yum list command, 103
    yum repolist all command, 110–111
    yum search command, 103–104
    yum-config-manager command, 111–112
pair/key values, JSON objects, 592
parent/child relationships, XML, 555–556
parentheses ( ), regular expressions (regex), 185
partitions, /dev directory, 121–125
passwords
  authentication, 517, 522–523, 525–526
  user/group management, Linux
    changing passwords, 141
    setting user passwords, 138–141
PATCH method, RESTCONF, 767–770
patches, 22
pattern validation, XML, 569–570
PAWS API, 944
PCC (Path Computation Clients), 860–861
  LSP configuration, 874–880
  SR-TE configuration, 867–870
PCE (Path Computation Elements)
  LSP configuration, 876–880
  PCEP, 860–861
  SR-PCE, 874
  SR-TE configuration, 869–874
PCEP (Path Computation Element Protocol)
  call flows, 861–864
  IOS XR configurations, 867–880
  lab topologies, 867
  LSP delegation, 864–867
PCE, 860–861
PCC, 860–861
LSP configuration, 874–880
SR-TE configuration, 867–870
PCEP, 860–861
LSP configuration, 876–880
SR-PCE, 874
SR-TE configuration, 869–874
peer authentication, 496–497
peering, 867
RFC, 860–861
state synchronization, 863
performance, Git, 312
periods (.),
. notation, regular expressions (regex), 190
.. notation, Linux directories, 37
regular expressions (regex), 185, 189
permissions
file permissions, changing with SFTP, 544–545
file security management, Linux, 143–144, 150–152
octal notation, 145–146
symbolic notation, 144–145
persistent connections, HTTP/1.1, 395–396
person in list() function, Python, 307–308
PFS (Perfect Forward Secrecy), 490
pipes (), 58, 65–67, 281
| (OR operator)
Python, 281, 285
regular expressions (regex), 186
|= operator, Python, 284
pipelining, HTTP/1.1, 396
platforms, API, 882
playbooks, Ansible, 990, 997–1000
conditional statements with loops and variables, 1032–1033
Jinja2 templates, 1040–1043
variables, defining, 1005–1006
+ (plus signs)
+ assignment operator, Python, 281
+= operator, Python, 284
regular expressions (regex), 186, 189, 190, 192
point parameter, RESTCONF, 771
polymorphism, OOP, 257
positive/negative values, arithmetic operators, Python, 282
POSIX (Portable Operating System Interface), Linux scripting, 210
POST (Power-On Self-Tests), 27
POST method
Flask, 350–352
HTTP client requests, 399–402
Postman, 443–445
Python and HTTP, 465
RESTCONF, 763–765
Postman, 436–437
HTTP
GET request messages, 447–454
POST method, 443–445
installing, 438
interface, 438–441
usage, 441–446
web/API development, 345
pound signs, #! (hashbangs), 205–208
predefined entries, XML, 557
predicates, XML expressions, 577
prefix NLRI, BGP-LS, 858–859
prefixes, XML namespaces, 560–561
Prefix-SID, 824, 827, 834–836
print_persons() function, Python, 307–309
printf command, Linux, 210–213
proc directory, Linux, 37
processes, Linux, 73–74, 80–81
product environments, replicating, 312
programmable interfaces, 881–882
API
  classifications, 882–883
eastbound API, 883
endpoints, 882
IOS XE, gNMI insecure mode, 815
IOS XE, NETCONF, 918–922
IOS XE, programmability, 885–886
IOS XR, NETCONF, 916–918
IOS XR, programmability, 886–887
NETCONF on IOS XE, 918–922
NETCONF on IOS XR, 916–918
NETCONF on NX-OS, 905–916
northbound API, 883
NX-API CLI, use cases, 893–898
NX-API REST, use cases, 898–905
Open NX-OS, Bash shells, 887–891
Open NX-OS, Guest shells, 887, 891–892
Open NX-OS, NETCONF, 905–916
Open NX-OS, programmability, 884–885
Open NX-OS, use cases, 887–892
platforms, 882
RESTful API, 883
RPC-based API, 883
southbound API, 883
webhooks, 882
westbound API, 883
collaboration platforms, 942
  Cisco collaboration portfolio, 942–944
collaboration API, 944–954
DNA Center, 931–933
device management, 934
Eastbound API, 933
event notifications, 935
Integration API, 935–936
Intent API, 934, 936–941
Northbound API, 933
Southbound API, 933
webbooks, 935
Westbound API, 933
Meraki API, 922, 923
  Captive Portal API, 923
Dashboard API, 922–931
Location Scanning API, 923
MV Sense API, 923
Webhook Alerts API, 922
programming

API, 7–8
algorithms, 258–259
computer science concepts, 255
defined, 7–8, 249–250
IOS XE, 885–886
IOS XR, 886–887
Network Programmability and Automation toolbox, 14–15
  Ansible. See also separate entry, 15–16
  Linux, 16–17
  protocols, 18–19
  Python, 15
  virtualization, 17
  YANG, 17
networks, 253–254
  CLI, 958–959, 962–967
gNMI, 961–962
gRPC, 961–962
  Linux shells, 960, 967–973
  NETCONF/YANG, 973–978
  network programmability, 960
  RESTCONF, 961
  RESTCONF/YANG, 978–987
  SNMP, 959–960
  vendor/API matrix, 957–958
OOP, 256
  abstraction, 257
  encapsulation, 257
  inheritance, 256–257
  polymorphism, 257
Open NX-OS, 884–885
pseudocode, 251–253
Python
  bytecode, 265–267
  bytecode interpreters, 264
  bytecode generators, 264
  code execution, 263–269
  code testing/verification, 269
  compiling code, 265–266
  complex numbers, 276
  data structures, 286–295
  data types, 270
  data types, numbers, 273–276
  data types, strings, 276–280
dictionaries, 290–292
dictionaries, 290–292
  executable Python files, 265
  float() function, 275
  floating point data, 275
  format() function, 280
  fundamentals, 260
  installing, 260–263
  int() function, 275
  integer operations, 275
  lexical analyzers, 263
  lists, 286–290
  Modulo Operator (%), 274
  numeric data, 269
  operators, 281–286
  sets, 294–295
  simple Python program, 264
  slicing string indexes, 278–279
  square brackets ([ ]), 278
  str.lower() function, 279–280
  str.replace() function, 279–280
  str.strip() function, 279–280
  str.upper() function, 279–280
  sum() function, 249–250
  tokenizers, 263
tuples, 292–293
  variables, 270–273
scripting languages vs scripting, 250–253
service providers, SDN, 819
   BGP-LS, 843
   BGP-LS, lab topologies, 845–846
   BGP-LS, link NLRI, 856–857
   BGP-LS, node NLRI, 854–855
   BGP-LS, NPF-XR, 845, 849, 851–854
   BGP-LS, peering, 843–844, 847–849
   BGP-LS, prefix NLRI, 858–859
   BGP-LS, routing, 846–847
   BGP-LS, routing types (overview), 850–854
   controllers, 821–823
   network requirements, 819–821
   overlay networks, 821
   PCEP, call flows, 861–864
   PCEP, IOS XR configurations, 867–880
   PCEP, lab topologies, 867
   PCEP, LSP delegation, 864–867
   PCEP, PCC, 860–861, 867–870, 874–880
   PCEP, PCE, 860–861, 869–874, 876–880
   PCEP, peering, 867
   PCEP, RFC, 860–861
   PCEP, state synchronization, 863
   SR, Adj-SID, 824, 827, 839–842
   SR, data plane verification, 830–831
   SR, forwarding, 832
   SR, IETF drafts, 823
   SR, lan topologies, 825
   SR, MPLS FIB for NPF-XR, 828–830
   SR, Node-SID, 824
   SR, NPF-XR, 835
   SR, Prefix-SID, 824, 827, 834–836
   SR, Prefix-SID, 824, 827, 834–836
   SR, segments, 824, 825–827
   SR, SR algorithm, 827
   SR, SRGB, 824, 827, 831
   SR, SRLB, 824
   SR, SR-MPLS, 824
   SR, SR-TE, 832–843
   underlay networks, 821–822
web application development, 250–251
Protobuf, gRPC, 786–790
   example of, 788–789
   in Python, 790–798
protocols, 18
   encoding, 18
   gRPC
      gNMI, 798–799
      gNMI, anatomy of, 799–801
      gNMI, Capabilities RPC, 810–811
      gNMI, Get RPC, 801–807
      gNMI, insecure mode in IOS XE, 815
      gNMI, managing network elements, 814–818
      gNMI, Python metaclasses, 815–816
      gNMI, Python sample Get script, 816–818
      gNMI, Set RPC, 807–810
      gNMI, Subscribe RPC, 811–814
   history of, 782–784
   managing network elements, 814–818
principles of, 782–784
Protobuf, example of, 788–789
Protobuf, Python, 790–798
server sample, 794–797
as a transport, 784–786
Protobuf, gRPC, 786–790
example of, 788–789
Python, 790–798
Python installations, 790–791
SCP
CentOS, 549–550
SFTP comparisons, 550
SFTP, 540–541
fetching/uploading files, 543
bhelp, 541–542
local/remote directory operations, 542–543
logins, 541
SCP comparisons, 550
SSH Authentication Protocol, 514–516
host-based authentication, 517–518
password authentication, 517
public key authentication, 516–517
SSH Connection Protocol, 518–521
SSH-TRANS, 513–514
transport protocols, 18–19, 781–782
ps command, Linux system maintenance, 74–77
pseudocode, 251–253
PSK (Pre-Shared Keys), 489
public keys
authentication, 516–517
copying onto servers, 524–525
pull phase, Git workflows, 317
push phase, Git workflows, 317
PUT method, 402–405, 765–767
pvdisplay command, Linux, 130–131, 133
pwd command, 29, 138–140
pyang, YANG data modeling, 673–679, 683–687
pyangbind, YANG data modeling, 679–682
Python, 15
Ansible, 991–992
code execution, 263–269
bytecode, 265–267
bytecode generators, 264
de code testing/verification, 269
compiling code, 265–266
executable Python files, 265
lexical analyzers, 263
numeric data, 269
simple Python program, 264
tokenizers, 263
code testing/verification, 269
comparison operators, 284–285
compiling code, 265–266
complex numbers, 276
tcontrol flow
elif statements, 297–298
if-else statements, 296–300
for loops, 301–304, 306
nested code blocks with conditional statements, 295–296
while loops, 304–306
data structures, 286
dictionaries, 290–292
lists, 286–290
<table>
<thead>
<tr>
<th>Python</th>
<th>1171</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sets</strong>, 294–295</td>
<td><strong>map() function</strong>, 1054–1055</td>
</tr>
<tr>
<td><strong>tuples</strong>, 292–293</td>
<td><strong>person_in_list() function</strong>, 307–308</td>
</tr>
<tr>
<td><strong>data types</strong>, 270</td>
<td><strong>print_persons() function</strong>, 307–309</td>
</tr>
<tr>
<td><strong>numbers</strong>, 273–276</td>
<td><strong>range() function</strong>, 302–303</td>
</tr>
<tr>
<td><strong>strings</strong>, 276–280</td>
<td><strong>representations</strong>, 307</td>
</tr>
<tr>
<td><strong>deleting</strong>, 292</td>
<td><strong>set.difference() function</strong>, 294–295</td>
</tr>
<tr>
<td><strong>functions</strong>, 292</td>
<td><strong>set.pop() function</strong>, 294</td>
</tr>
<tr>
<td><strong>if-else statements</strong>, 299–300</td>
<td><strong>set.remove() function</strong>, 294</td>
</tr>
<tr>
<td><strong>dict.items() function</strong>, 292</td>
<td><strong>set.union() function</strong>, 294–295</td>
</tr>
<tr>
<td><strong>dict.keys() function</strong>, 292, 307–308</td>
<td><strong>set.update() function</strong>, 294</td>
</tr>
<tr>
<td><strong>dict.values() function</strong>, 292</td>
<td><strong>split() function</strong>, 1051–1054</td>
</tr>
<tr>
<td><strong>DRY principle</strong>, 311, 336</td>
<td><strong>str.lower() function</strong>, 279–280</td>
</tr>
<tr>
<td><strong>exit() function</strong>, 272</td>
<td><strong>str.replace() function</strong>, 279–280</td>
</tr>
<tr>
<td><strong>float() function</strong>, 275</td>
<td><strong>str.reverse() function</strong>, Python, 292–293</td>
</tr>
<tr>
<td><strong>floating point data</strong>, 275</td>
<td><strong>str.strip() function</strong>, 279–280</td>
</tr>
<tr>
<td><strong>format() function</strong>, 280</td>
<td><strong>str.upper() function</strong>, 279–280, 292–293</td>
</tr>
<tr>
<td><strong>funcname() function</strong>, 307</td>
<td><strong>sum() function</strong>, 249–250</td>
</tr>
<tr>
<td><strong>functions</strong>, 306–307</td>
<td><strong>validate_person() function</strong>, 307–308, 309</td>
</tr>
<tr>
<td><strong>calling in code blocks</strong>, 307–309</td>
<td><strong>fundamentals</strong>, 260</td>
</tr>
<tr>
<td><strong>defining in code blocks</strong>, 307–309</td>
<td><strong>gNMI</strong></td>
</tr>
<tr>
<td><strong>dict.items() function</strong>, 292</td>
<td><strong>metaclasses</strong>, 815–816</td>
</tr>
<tr>
<td><strong>dict.values() function</strong>, 292</td>
<td><strong>gRPC and Protobuf</strong>, 790–798</td>
</tr>
<tr>
<td><strong>exit() function</strong>, 272</td>
<td><strong>HTTP</strong></td>
</tr>
<tr>
<td><strong>float() function</strong>, 275</td>
<td><strong>requests packages</strong>, 464–467</td>
</tr>
<tr>
<td><strong>format() function</strong>, 280</td>
<td><strong>socket modules</strong>, 455–457</td>
</tr>
<tr>
<td><strong>funcname() function</strong>, 307</td>
<td><strong>TCP over Python</strong>, 455–457</td>
</tr>
<tr>
<td><strong>input() function</strong>, 298</td>
<td><strong>urllib packages</strong>, 458–463</td>
</tr>
<tr>
<td><strong>int() function</strong>, 275, 1049–1055</td>
<td><strong>installing</strong>, 260–263</td>
</tr>
<tr>
<td><strong>join() function</strong>, 1050–1051</td>
<td><strong>JSON schemas, data validation</strong>, 609–610</td>
</tr>
<tr>
<td><strong>list.append() function</strong>, 288–289</td>
<td><strong>machine learning</strong>, 382–384</td>
</tr>
<tr>
<td><strong>list.pop() function</strong>, 288–289</td>
<td><strong>list.reverse() function</strong>, 289</td>
</tr>
<tr>
<td><strong>list.sort() function</strong>, 289</td>
<td><strong>list.update() function</strong>, 294</td>
</tr>
</tbody>
</table>
modules

application development, 333–336
classes, 335–336
constructors, 335–336
from-import statements, 334–335
import statements, 333–334

NETCONF, ncclient, 735–739

network automation

architectures, 353–354, 371–375
Jinja2 templates, 363–375
NAPALM libraries, 354–359, 371–375

network device management

NETCONF/YANG, 976–978
RESTCONF/YANG, 984–987

objects, serializing with YAML, 628–629

operators, 281

& operator, 281
&= operator, 284
**= operator, 284
*= operator, 284
^ operator, 281
^= operator, 284
|= operator, 284
== operator, 284
>= operator, 285
l operator, 281, 285
l= operator, 284
+= operator, 284
> operator, 285
>= operator, 285
>> (signed right shift) operator, 281
>>= operator, 284
[] square brackets, 278
[=] operator, 284
– operator, 281, 285
and operator, 281, 285
arithmetic operators, 281–283
assignment operators, 284
bitwise operators, 281–283
identity operators, 284–285
in operator, 286
integer operations, 275
is not operator, 285
is operator, 285
logical operators, 284–285
logical operators, if-else statements, 297–298
membership operators, 285–286
not in operator, 286
NOT operator, 281, 285
OR operator, 281, 285
orchestration, 375–376
Docker, 376–378
Kubernetes, 378–382
protoc installations, 790–791
PyYAML, 626–628
requests packages, 464–467
RESTCONF, 777–779
sets, 294–295
slicing, lists, 286–287
slicing string indexes, 278–279
regular expressions (regex)

Q

query parameters, RESTCONF, 771–777
question mark (?), regular expressions (regex), 186, 189, 191–192

R

range() function, Python, 302–303
read command, Linux, 214–217
record protocol, TLS 1.3, 499, 503
Red Hat, Linux distributions, 26
redirecting input/output, Linux, 57–59
| (piping), 58, 65–67
>> notation, 61–62
stderr, 59, 62–65
stdin, 59, 61
stdout, 59, 61, 62–65
tee command, 66–67
regexp, XML content validation, 569
regular expressions (regex)
* metacharacter, 185, 189–190
\ metacharacter, 186
\< metacharacter, 185, 188–189
\> metacharacter, 185, 188–189
^ metacharacter, 185, 187
,. 185
[N, M] metacharacter, 186, 189–192
[N] metacharacter, 186, 189–192
$ metacharacter, 185, 187
 . metacharacter, 185, 189
.* notation, 190
( ) (capture groups), 186
( ) (parentheses), 185
| OR operator, 186
+ metacharacter, 186, 189, 190, 192
? metacharacter, 186, 189, 191–192
[ ] (square brackets), 185
[first literal - last_literal] metacharacter, 185
[literals] metacharacter, 185
grep command, 184–193
matching uppercase/lowercase characters, 187–188
printing lines without patterns, 192–193
repetition metacharacters, 185–186, 190–191
testing, 186
relative paths, Linux directories, 38–39
remote subnets, Linux routing tables, 166–167
remote/local directory operations, SFTP, 542–543
removing Linux
directories, 50–51
files, 47, 48
routing table entries, 168
renaming Linux
directories, 50–51
files, 47, 48
repetition metacharacters, regular expressions (regex), 185–186, 190–191
repetitive objects, JSON schemas, 598–602
replicating product environments, 312
repositories
Docker, 318
Git, 312–313
  cloning, 316–317
  fetching remote repository updates, 314–315
  setting up, 315–316
  updating local repositories, 314–315
software, 95–96
  EPEL repositories, 111–112
  listing, 110–111
  LocalRepo repositories, 112–117
request header fields, HTTP, 416
request messages, RESTCONF, 740, 743–744
request targets, CONNECT method (HTTP), 407
request packages, Python and HTTP, 464–467
resources
Linux resource utilization, 83–85
RESTCONF, 746–747
  API resource, 747–749
  data resource, 753–758
  datastore resource, 749–750
  schema resource, 750–753
  YANG library version resource, 758
response header fields, HTTP, 416
response messages, RESTCONF, 740, 744–745
REST framework, 392–394
REST API, 322, 392–393, 945–946, 948–954
RESTCONF, 739
  architecture, 742–743
  content layer, 743
  editing data, 763–771
  error reporting, 746
HTTP, 740
  headers, 745–746
  response messages, 740
messages layer, 742–743, 759
  constructing messages, 745
  content parameter, 771
  with-defaults parameter, 771
  DELETE method, 770–771
  depth parameter, 771
  editing data, 763–771
  error reporting, 746
  fields parameter, 771, 773–777
  filter parameter, 771
  GET method, 760–763
  HEAD method, 763
  HTTP headers, 745–746
  insert parameter, 771
  OPTIONS method, 759–760
  PATCH method, 767–770
  point parameter, 771
  POST method, 763–765
  PUT method, 765–767
  query parameters, 771–777
  request messages, 743–744
  response messages, 744–745
  retrieving data, 759–763
  start-time parameter, 771
  stop-time parameter, 771
  NETCONF and, 740–742
  network programmability, 961, 978–987
  operations layer, 743
  overview, 739–742
  Python, 777–779
  request messages, 740, 743–744
  resources, 746–747
    API resource, 747–749
    data resource, 753–758
    datastore resource, 749–750
  schema resource, 750–753
  YANG library version resource, 758
  response messages, 744–745
  RESTCONF/YANG, network programmability, 978–987
  retrieving data, 759–763
  transport layer, 742, 743
  RESTful API, 883
  resuming stopped processes, 80–81
  reusing
    automations, 1111
    code, application development, 312
  RFC 2616, HTTP headers, 416
  RFC 3986, HTTP, 393–394
  RFC 6265, HTTP state management, 484
  RFC 6749, HTTP, OAuth protocol, 475–476
  RFC 6750, HTTP, OAuth protocol with Bearer Tokens, 475–476
  RFC 7230, HTTP headers, 416–417
  RFC 7231, HTTP/1.1 server status codes, 409–410
  RFC 7457, TLS, 488
  RFC 7540, HTTP, 388
  RFC 8446, TLS, 487
  RFC, PCEP, 860–861
  right arrows (>)
    > Jinja2 operator, 1019
    > Python operator, 285
    => (right arrow, equal sign)
      assignment operator, Jinja2, 1019
      assignment operator, Python, 285
    >> signed right shift operator,
      Python, 281
    >>> operator, Python, 284
rm command, 47, 48
rmkdir command, 50–51
rollback-on-error capability, NETCONF, 732–733
root directory (/), Linux, 36–37
rotating Linux system logs, 91
routing (static)
  200 OK responses
    DELETE method, 405–406
    POST method, 394–401
    PUT method, 403–404
Nexus switches
    DELETE method, 405–406
    POST method, 399–401
    PUT method, 403–404
routing tables, Linux, 163–166
  forwarding interfaces, 167
  remote subnets, 166–167
  removing entries, 168
  viewing, 163
RPC (Remote Procedure Calls)
  bidirectional RPC, 785
  Capabilities RPC, gNMI, 810–811
  client-streaming RPC, 785
  Get RPC, gNMI, 801–807
  gRPC
    gNMI, 798–799
    gNMI, anatomy of, 799–801
    gNMI, Capabilities RPC, 810–811
    gNMI, Get RPC, 801–807
    gNMI, insecure mode in IOS XE, 815
    gNMI, managing network elements, 814–818
    gNMI, Python sample Get script, 816–818
    gNMI, Set RPC, 807–810
    gNMI, Subscribe RPC, 811–814
history of, 782–784
managing network elements, 814–818
principles of, 782–784
Protobuf, example of, 788–789
Protobuf, Python, 790–798
server sample, 794–797
as a transport, 784–786
server-streaming RPC, 785
Set RPC, gNMI, 807–810
Subscribe RPC, gNMI, 811–814
unary RPC, 785
RPC-based API, 883
rpc messages, 691, 698–699
rpc-reply messages, 691, 699–701
RPM (RPM Package Manager), 95, 97–101
RSA keys
  generating/verifying, SSH NX-OS setups, 538–539
  Linux network programmability, 971–973
rules of thumb
  API, 1118
  application hosting, 1115–1116
  automation, 1109–1110
    complexity, 1111–1112
    cost/benefit analysis, 1112
    reusing automations, 1111
  cleaning up networks, 1110
  containers, 1114–1115
  databases, 1117
  documentation, 1111
model-driven telemetry, 1113–1114
search engines, 1117
software development methodologies, 1116–1117
run directory, Linux, 37
ryslogd command, Linux, 91–93

scripting, Linux, 183
Arguments.bash script, 213–214
arithmetic operations, 220–222
arrays
adding/removing elements, 224–226
associative arrays, 222
catenating, 221–226
declaring, 222–224
defined, 222
indexed arrays, 222–224
awk programming language, 194–197
Bash scripting, 184
Bash shells, 206–207, 213–214
case-in constructs, 232–234
comments, 207–208
conditional statements, 226
== (double equal sign), 229
= (equal sign), 229
case-in constructs, 232–234
if-then constructs, 226–232
Expect programming language, 245–246
functions, 240–244
GNU Bash Manual, 184
if-then constructs, 226–232
input, 213
Arguments.bash script, 213–214
read command, 214–217
loops, 234–235
for-do loops, 235–237
until-do loops, 239–240
while-do loops, 237–239
networks, 174–179
output, 208
echo command, 208–210
printf command, 210–213
POSIX, 210

saving YAML data streams, 629
scalars, YAML, 620–621
scale command, Linux, 221–222
schema resource, RESTCONF, 750–753
schemas, JSON, 595
annotation, 596
basic schema without content, 596–597
data validation, 609–614
definitions in, 598–601
external schemas, referencing, 602–609
IP VPN, 597–598, 601–602, 607–609
keywords, 596
properties of, 597–598
purpose of, 595–596
repetitive objects, 598–602
structure of, 595–598
validation keywords, 596–597
SCI, Linux kernels, 24
scope, Python variables, 272
SCP (Secure Copy Protocol)
CentOS, 549–550
SFTP comparisons, 550
scp (Secure Copy Protocol)
CentOS, 549–550
SFTP comparisons, 550
scientific programming language, 194–197
arrays
adding/removing elements, 224–226
associative arrays, 222
concatenating, 221–226
declaring, 222–224
defined, 222
indexed arrays, 222–224
awk programming language, 194–197
Bash scripting, 184
Bash shells, 206–207, 213–214
case-in constructs, 232–234
comments, 207–208
conditional statements, 226
== (double equal sign), 229
= (equal sign), 229
case-in constructs, 232–234
if-then constructs, 226–232
Expect programming language, 245–246
functions, 240–244
GNU Bash Manual, 184
if-then constructs, 226–232
input, 213
Arguments.bash script, 213–214
read command, 214–217
loops, 234–235
for-do loops, 235–237
until-do loops, 239–240
while-do loops, 237–239
networks, 174–179
output, 208
echo command, 208–210
printf command, 210–213
POSIX, 210

arrays
adding/removing elements, 224–226
associative arrays, 222
concatenating, 221–226
declaring, 222–224
defined, 222
indexed arrays, 222–224
awk programming language, 194–197
Bash scripting, 184
Bash shells, 206–207, 213–214
case-in constructs, 232–234
comments, 207–208
conditional statements, 226
== (double equal sign), 229
= (equal sign), 229
case-in constructs, 232–234
if-then constructs, 226–232
Expect programming language, 245–246
functions, 240–244
GNU Bash Manual, 184
if-then constructs, 226–232
input, 213
Arguments.bash script, 213–214
read command, 214–217
loops, 234–235
for-do loops, 235–237
until-do loops, 239–240
while-do loops, 237–239
networks, 174–179
output, 208
echo command, 208–210
printf command, 210–213
POSIX, 210

arrays
adding/removing elements, 224–226
associative arrays, 222
concatenating, 221–226
declaring, 222–224
defined, 222
indexed arrays, 222–224
awk programming language, 194–197
Bash scripting, 184
Bash shells, 206–207, 213–214
case-in constructs, 232–234
comments, 207–208
conditional statements, 226
== (double equal sign), 229
= (equal sign), 229
case-in constructs, 232–234
if-then constructs, 226–232
Expect programming language, 245–246
functions, 240–244
GNU Bash Manual, 184
if-then constructs, 226–232
input, 213
Arguments.bash script, 213–214
read command, 214–217
loops, 234–235
for-do loops, 235–237
until-do loops, 239–240
while-do loops, 237–239
networks, 174–179
output, 208
echo command, 208–210
printf command, 210–213
POSIX, 210

arrays
adding/removing elements, 224–226
associative arrays, 222
concatenating, 221–226
declaring, 222–224
defined, 222
indexed arrays, 222–224
awk programming language, 194–197
Bash scripting, 184
Bash shells, 206–207, 213–214
case-in constructs, 232–234
comments, 207–208
conditional statements, 226
== (double equal sign), 229
= (equal sign), 229
case-in constructs, 232–234
if-then constructs, 226–232
Expect programming language, 245–246
functions, 240–244
GNU Bash Manual, 184
if-then constructs, 226–232
input, 213
Arguments.bash script, 213–214
read command, 214–217
loops, 234–235
for-do loops, 235–237
until-do loops, 239–240
while-do loops, 237–239
networks, 174–179
output, 208
echo command, 208–210
printf command, 210–213
POSIX, 210
programming languages vs scripting, 250–253
regular expressions (regex), 184–193
  * metacharacter, 185, 189–190
  \ metacharacter, 186
  < metacharacter, 185, 188–189
  > metacharacter, 185, 188–189
  ^ metacharacter, 185, 187
  { }, 185
  {N, M} metacharacter, 186, 189–192
  {N} metacharacter, 186, 189–192
  $ metacharacter, 185, 187
  . metacharacter, 185, 189
  .* notation, 190
  ( ) (capture groups), 186
  ( ) (parentheses), 185
  | OR operator, 186
  + metacharacter, 186, 189, 190, 192
  ? metacharacter, 186, 189, 191–192
  [ ] (square brackets), 185
  [first_literal - last_literal] metacharacter, 185
  [literals] metacharacter, 185
grep command, 184–193
matching uppercase/lowercase characters, 187–188
printing lines without patterns, 192–193
repetition metacharacters, 185–186, 190–191
testing, 186
scale command, 221–222
sed command, 197–205
shells, 183, 205–208
variables, 217–218
declaring, 218–219
one-dimensional variables, 218
value assignments, 218–219
web application development, 250–251
SDLC (Software Development Life Cycle), 1116
SDN (Software-Defined Networking), 13, 819
BGP-LS, 843
lab topologies, 845–846
link NLRI, 856–857
node NLRI, 854–855
NPF-XR, 845, 849, 851–854
peering, 843–844, 847–849
prefix NLRI, 858–859
routing, 846–847
routing types (overview), 850–854
controllers, 821–823
network requirements, 819–821
overlay networks, 821
PCEP
call flows, 861–864
IOS XR configurations, 867–880
lab topologies, 867
LSP delegation, 864–867
PCC, 860–861
PCC, LSP configuration, 874–880
PCC, SR-TE configuration, 867–870
PCE, 860–861
PCE, LSP configuration, 876–880
PCE, SR-PCE, 874
PCE, SR-TE configuration, 869–874
peering, 867
RFC, 860–861
state synchronization, 863

SR
Adj-SID, 824, 827, 839–842
data plane verification, 830–831
forwarding, 832
IETF drafts, 823
lab topologies, 825
MPLS FIB for NPF-XR, 828–830
Node-SID, 824
NPF-XR, 835
Prefix-SID, 824, 827, 834–836
segments, 824, 825–827
SR algorithm, 827
SRGB, 824, 827, 831
SRLB, 824
SR-MPLS, 824
SR-TE, 832–843
underlay networks, 821–822

search engines, rules of thumb, 1117

security
AEAD, 495–496
AES-CCM protocol, 495
AES-GCM protocol, 495
CCM, 495
ciphers, 492
  block ciphers, 492–493
  CBC, 492–493
  CMAC, 494
  CTR mode, 492–493
  symmetric ciphers, 492
digital signatures, 496–497
encryption keys, 488–489
  asymmetric keys, 490
  DH protocol, 490–492
  ephemeral keys, 490
generation/exchanges, 488–492
  HKDF, 492
  key exchange algorithm, 490
  KM, 492
  PFS, 490
  PSK, 489
  symmetric keys, 489
  GCM, 495
  Git, 312
Linux, 135
  ACL, 148–155
  file security management, 143–148
  system security, 155–158
  user/group management, 136–142
MAC, 493–494
peer authentication, 496–497
TLS, 487–488
  0-RTT, 502–503
  HTTPS, 503
  TLS 1.3, 498–499
  TLS 1.3, alert protocol, 499
  TLS 1.3, handshake protocol, 499–502
  TLS 1.3, record protocol, 499, 503
  workflows, 499–500
sed command, Linux, 197–205
Segment Routing. See SR
segments, 824
sequences, YAML, 618–620
serializer.py files, Django, 343
server response header fields, HTTP, 425–427

servers
application web servers, running with Django, 338–339
Docker servers, 322
Git servers, setting up, 313–314
HTTP client/server connections, 394–395
public keys, copying onto servers, 524–525
SFTP logins, 541
status codes, HTTP, 408–409
  1xx information status codes, 411
  2xx successful status codes, 411–412
  3xx redirection status codes, 412
  4xx server error status codes, 413–414
  5xx client error status codes, 414
Cisco devices, 414
RFC 7231, 409–410

server-streaming RPC, 785

service layer API, IOS XR programmability, 886

service provider programmability, SDN, 819
BGP-LS, 843
  lab topologies, 845–846
  link NLRI, 856–857
  node NLRI, 854–855
NPF-XR, 845, 849, 851–854
peering, 843–844, 847–849
prefix NLRI, 858–859
routing, 846–847
routing types (overview), 850–854
controllers, 821–823
network requirements, 819–821
overlay networks, 821

PCEP
call flows, 861–864
IOS XR configurations, 867–880
lab topologies, 867
LSP delegation, 864–867
PCC, 860–861
PCC, LSP configuration, 874–880
PCC, SR-TE configuration, 867–870
PCE, 860–861
PCE, LSP configuration, 876–880
PCE, SR-PCE, 874
peering, 867
RFC, 860–861
state synchronization, 863

SR
Adj-SID, 824, 827, 839–842
data plane verification, 830–831
forwarding, 832
IETF drafts, 823
lab topologies, 825
MPLS FIB for NPF-XR, 828–830
Node-SID, 824
NPF-XR, 835
Prefix-SID, 824, 827, 834–836
segments, 824, 825–827
SR algorithm, 827
SRGB, 824, 827, 831
<table>
<thead>
<tr>
<th><strong>Software</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRLB, 824</td>
</tr>
<tr>
<td>SR-MPLS, 824</td>
</tr>
<tr>
<td>SR-TE, 832–843</td>
</tr>
<tr>
<td>underlay networks, 821–822</td>
</tr>
<tr>
<td>services, Linux, 74, 81–83</td>
</tr>
<tr>
<td>session channels, SSH Connection Protocol, 519–520</td>
</tr>
<tr>
<td>session operations, NETCONF, 721–722</td>
</tr>
<tr>
<td>Set RPC, gNMI, 807–810</td>
</tr>
<tr>
<td>set.add() function, Python, 294</td>
</tr>
<tr>
<td>set.difference() function, Python, 294–295</td>
</tr>
<tr>
<td>setfacl command, Linux, 151–155</td>
</tr>
<tr>
<td>set.pop() function, Python, 294</td>
</tr>
<tr>
<td>set.remove() function, Python, 294</td>
</tr>
<tr>
<td>sets, Python, 294–295</td>
</tr>
<tr>
<td>settings.py files, Django, 339–341</td>
</tr>
<tr>
<td>set.union() function, Python, 294–295</td>
</tr>
<tr>
<td>set.update() function, Python, 294</td>
</tr>
<tr>
<td>SFTP (Secure File Transfer Protocol)</td>
</tr>
<tr>
<td>fetching/uploading files, 543</td>
</tr>
<tr>
<td>help, 541–542</td>
</tr>
<tr>
<td>local/remote directory operations, 542–543</td>
</tr>
<tr>
<td>logins, 541</td>
</tr>
<tr>
<td>SCP comparisons, 550</td>
</tr>
<tr>
<td>SSH (Secure Shell), 540–541</td>
</tr>
<tr>
<td>sg command, Linux, 146–147</td>
</tr>
<tr>
<td>shells</td>
</tr>
<tr>
<td>Bash shells</td>
</tr>
<tr>
<td><em>Arguments.bash script</em>, 213–214</td>
</tr>
<tr>
<td>arithmetic operations, 220–222</td>
</tr>
<tr>
<td>arithmetic operators, 229</td>
</tr>
<tr>
<td>CLI programmable interface creation, 963–967</td>
</tr>
<tr>
<td>Expect programming language, 245–246</td>
</tr>
<tr>
<td>file comparison operators, 230–232</td>
</tr>
<tr>
<td>functions, 244</td>
</tr>
<tr>
<td>integer comparison operators, 229–230</td>
</tr>
<tr>
<td>Linux interface configuration, 969–970</td>
</tr>
<tr>
<td>scripting, 206–207, 213–214</td>
</tr>
<tr>
<td>string comparison operators, 228–229</td>
</tr>
<tr>
<td>string operators, 227–228</td>
</tr>
<tr>
<td>Linux shells, 183</td>
</tr>
<tr>
<td>network programmability, 960, 967–973</td>
</tr>
<tr>
<td>scripts, 205–208</td>
</tr>
<tr>
<td>SSH, Linux network programmability, 971–973</td>
</tr>
<tr>
<td>shred command, 48</td>
</tr>
<tr>
<td>signatures, digital, 496–497</td>
</tr>
<tr>
<td>signed right shift (&gt;&gt;) operator, Python, 281</td>
</tr>
<tr>
<td>single inheritance, OOP, 257</td>
</tr>
<tr>
<td>single sources of truth. See SSoT</td>
</tr>
<tr>
<td>Slackware, Linux distributions, 26</td>
</tr>
<tr>
<td>slicing, Python</td>
</tr>
<tr>
<td>lists, 286–288</td>
</tr>
<tr>
<td>string indexes, 278–279</td>
</tr>
<tr>
<td>SNMP (Simple Network Management Protocol)</td>
</tr>
<tr>
<td>model-driven telemetry, 1113</td>
</tr>
<tr>
<td>network programmability, 959–960</td>
</tr>
<tr>
<td>socket modules, Python and HTTP, 455–457</td>
</tr>
<tr>
<td>soft links, Linux, 55–57</td>
</tr>
<tr>
<td>software</td>
</tr>
<tr>
<td>compiling, manually, Linux, 96–97</td>
</tr>
<tr>
<td>dependencies, 95</td>
</tr>
<tr>
<td>dependency hell, 1114–1115</td>
</tr>
</tbody>
</table>
software development methodologies, 1116–1117
engineers
automation, 19–20
career paths, questions, 1118–1119
installing on Linux, 94–96
package management software, 95
DNF, 95, 117
RPM, 95, 97–101
YUM, 95, 101–117
repositories, 95–96
EPEL repositories, 111–112
listing, 110–111
LocalRepo repositories, 112–117
rules of thumb, 1116–1117
SDLC, 1116
SDN, 13
sort command, 59–61
stdin, 59–61
stdout, 61
sorting YAML data streams, 630–631
source code, Linux software installations, 94
sources of truth, 11–12
automation, reusing automations, 1111
finding, 1110–1111
southbound API, 883, 933
space characters in value fields, XML, 562
split() function, 1051–1054
square brackets ([ ])
[=] operator, Python, 284
[first_literal - last_literal], regular expressions (regex), 185
[_literals], regular expressions (regex), 185
Python
lists, 286–287
strings, 278
regular expressions (regex), 185
SR (Segment Routing)
Adj-SID, 824, 827, 839–842
data plane verification, 830–831
forwarding, 832
IETF drafts, 823
lab topologies, 825
MPLS FIB for NPF-XR, 828–830
Node-SID, 824
NPF-XR, 835
Prefix-SID, 824, 827, 834–836
segments, 824, 825–827
SR algorithm, 827
SRGB, 824, 827, 831
SRLB, 824
SR-MPLS, 824
SR-TE
Adj-SID, 839–842
BGP, 836–839
data plane verification, 842–843
NPF-XR, 836–843
PCEP PCC configuration, 839–842
PCEP PCE configuration, 869–874
policy components, 833–834
Prefix-SID, 834–836
process of, 832–833
SRGB (Segment Routing Global Blocks), 824, 827, 831
SRLB (Segment Routing Local Blocks), 824
SR-PCE, 874
srv directory, Linux, 37
SSH (Secure Shell)
  Authentication Protocol, 514–516
    host-based authentication, 517–518
    password authentication, 517, 522–523, 525–526
    public key authentication, 516–517
  Bash shells, 539–540, 548–549
  Connection Protocol, 518–521
deployment, 528–531, 533–534
  key-based authentication, 523–525
  Linux network programmability, 971–973
  NETCONF over SSH, 694–695
  OpenSSH, 510, 522
  overview, 509–510
  protocol setup flow, 510
  public keys, copying onto servers, 524–525
remote access requirements, 510
SCP
  CentOS, 549–550
  SFTP comparisons, 550
setting up, 521
  CentOS, 521–526
  Cisco devices, 545–549
  IOS XE, 526–531, 545–546
  IOS XR, 532–536, 546–547
  NX-OS, 537–540, 547–548
SFTP, 540–541
  fetching/uploading files, 543
tunneling, 520
  verifying sessions, 533–534
SSoT (Single Sources of Truth), 11–12, 1110–1111
start lines, HTTP messages, 415
start tags, XML, 555
start-time parameter, RESTCONF, 771
stat command, 62–63
state management, HTTP, 483–487
state synchronization, PCEP, 863
statements (conditional)
  Ansible, 1016–1019
    checking for substrings in variables, 1021–1022
    checking for variables, 1019–1021
    combining multiple conditional statements, 1022–1024
    conditional statements with loops, 1024–1027
    conditional statements with loops and variables, 1027–1033
    Jinja2 templates, 1045–1049
      AND/OR logic, 1022–1024
    Jinja2 templates, 1045–1049
Linux scripting, 226
  == (double equal sign), 229
  = (equal sign), 229
  case-in constructs, 232–234
  if-then constructs, 226–232
nested code blocks with conditional statements, Python control flow, 295–296

static routing
200 OK responses
   DELETE method, 405–406
   POST method, 394–401
   PUT method, 403–404

Nexus switches
   DELETE method, 405–406
   POST method, 399–401
   PUT method, 402–405

stderr, Linux, 59, 62–65
stdin, Linux, 59–61
stdout, Linux, 59, 61, 62–65
stopping jobs, 80–81
stop-time parameter, RESTCONF, 771

storage, Linux, 119
/dev directory, 119–120
   contents of, 120
   device file types, 120–121
   fdisk command, 121–125
   file system creation, 125–126
   hard disk partitions, 121–125
   mkfs command, 125–126
   mounting file systems, 126–128
   unmounting file systems, 127
LVM, 128–135
streaming telemetry, 1113–1114
streams, HTTP/2, 505

string comparison operators, Bash shells, 228–229

string data types, Python, 276–277
   concatenating, 277
   formatting, 280
   indexes, 277–279

string operators, Bash shells, 227–228

string variables, Ansible, 1006
strings, JSON, 593
str.lower() function, Python, 279–280
str.replace() function, Python, 279–280
str.reverse() function, Python, 292–293
str.strip() function, Python, 279–280
str.upper() function, Python, 279–280

stylesheets, XSLT, 578–579

su command, 29–30

subnets (remote), Linux routing tables, 166–167
Subscribe RPC, gNMI, 811–814
subtree filters, NETCONF, 703–710
sudo command, Linux, 136

sudo yum remove httpd command, Linux, 109

sudo yum update command, Linux, 110

sum() function, Python, 249–250

switches (Nexus)
   authentication, 401–402, 463
   static routing
      DELETE method, 405–406
      POST method, 399–401
      PUT method, 402–405
symbolic notation, file permissions, 144–145

symlinks, Linux, 56–57

symmetric ciphers, 492
symmetric keys, 489

sys directory, Linux, 37
sysctl command, Linux, 167–168

syslog, model-driven telemetry, 1113
syslog messages, Linux, 93–94

system calls, Linux, 24
system daemons, Linux, 24
system information, Linux
cat/proc/cpuinfo command, 87–88
date command, 85–86
dmesg command, 90
dmidecode command, 88–89
lsppci command, 90
timedatectl command, 86
update command, 86–87
system logs, Linux, 91
journald command, 93–94
lastlog command, 91
rotation, 91
ry slogd command, 91–93
tail command, 91
system maintenance, Linux, 73
jobs, 74
displaying status, 80
stopping, 80–81
kill command, 78–80, 81
grep command, 78
processes, 73–74, 80–81
ps command, 74–77
services, 74, 81–83
systemctl command, 81–83
threads, 73
system security, Linux, 155–158
systemctl command, Linux system maintenance, 81–83
defined, 555
end tags, 555
predefined entries, 557
prefixes, 560–561
start tags, 555
using multiple times, 558–559
values, 555
YAML, 617, 621–624
tail command, Linux system logs, 91
tar archiving utility, Linux, 67, 70–73
TCP over Python, 455–457
TE (Traffic Engineering), SR-TE
Adj-SID, 839–842
BGP, 836–839
data plane verification, 842–843
NPF-XR, 836–843
PCEP PCC configuration, 839–842
PCEP PCE configuration, 869–874
policy components, 833–834
Prefix-SID, 834–836
process of, 832–833
te command, 66–67
telemetry
IOS XE programmability, 886
IOS XR programmability, 887
model-driven telemetry, 1113–1114
Open NX-OS programmability, 885
streaming telemetry, 1113–1114
templates, Jinja2, 363–364
Ansible, 1034–1040
conditional statements, 1045–1049
loops, 1040–1043
playbooks, 1040–1043
variables, 1042–1043
IOS XE, 367–371
IOS XR, 367–371

T

tags
XML
child/parent relationships, 555–556
creating, 557
templates, Jinja2

NAPALM libraries, 371–375
Netmiko libraries, 364–367
Nornir libraries, 367–369, 371–375
NX-OS, 367–371
nx-os_config modules, 1091–1093

Terminal, 29
testing
  Python code, 269
  regular expressions (regex), 186
  repetition metacharacters, 190–191
threads, Linux, 73
thumb, rules of
  API, 1118
  application hosting, 1115–1116
  automation, 1109–1110
    complexity, 1111–1112
    cost/benefit analysis, 1112
    reusing automations, 1111
  cleaning up networks, 1110
  containers, 1114–1115
  databases, 1117
documentation, 1111
  model-driven telemetry, 1113–1114
  search engines, 1117
  software development methodologies, 1116–1117
time stamps, Linux files, 41
timedatectl command, Linux, 86
TLS (Transport Layer Security), 487–488
  0-RTT, 502–503
HTTPS, 503
  TLS 1.3, 498–499
    alert protocol, 499
    handshake protocol, 499–502
    record protocol, 499, 503
  workflows, 499–500
tmp directory, Linux, 37
toggling interface state, Linux, 161
tokenizers, Python code execution, 263
top command, Linux resource utilization, 83–85
touch command, 46, 48
TRACE method, 408
Traffic Engineering. See TE
transactions, HTTP, 389
  client requests, 397–398
    CONNECT method, 407
    DELETE method, 405–406
    GET method, 398
    GET method, Bash shells, 447–454
    HEAD method, 398
    OPTIONS method, 407–408
    PUT method, 402–405
    TRACE method, 408
  server status codes, 408–409
    1xx information status codes, 411
    2xx successful status codes, 411–412
    3xx redirection status codes, 412
    4xx server error status codes, 413–414
    5xx client error status codes, 414
    Cisco devices, 414
    RFC 7231, 409–410
transport layer
  NETCONF, 692–693
    over SSH, 694–695
    transport protocol requirements, 693–694
  RESTCONF, 742, 743
transport protocols, 18–19
gRPC
  gNMI, 798–799
  gNMI, anatomy of, 799–801
  gNMI, Capabilities RPC, 810–811
  gNMI, Get RPC, 801–807
  gNMI, insecure mode in IOS XE, 815
  gNMI, managing network elements, 814–818
  gNMI, Python metaclasses, 815–816
  gNMI, Python sample Get script, 816–818
  gNMI, Set RPC, 807–810
  gNMI, Subscribe RPC, 811–814
  history of, 782–784
  managing network elements, 814–818
  principles of, 782–784
  Protobuf, example of, 788–789
  Protobuf, Python, 790–798
  server sample, 794–797
  as a transport, 784–786
  requirements for efficient transport, 781–782
truth, sources of, 11–12
  automations, 1111
  finding, 1110–1111
TSP API (Transport Service Provider API), 945
tunneling, SSH, 520
tuples, Python
  deleting, 293
  functions, 292–293
  joining, 293
two asterisks (**), assignment operator, Python, 281
two dot notation (..), Linux directories, 37
two equal signs (==)
  conditional statements, 229
  Jinja2, 1019
  Python, 285
two semicolons (;), case-in constructs, 233–234

U

UDS API, 945
unary RPC, 785
unbuffered/buffered access, Linux / dev directory, 120
underlay networks, 9, 821–822
Unified Communications, 942
  Cisco Business Edition, 942
  Cisco Webex Cloud Calling, 942
  Cisco Webex Contact Center, 943
  Cisco Webex teams, 942
  Contact Center, 942–943
  CUCM, 942
    AXL API, 944
    CER API, 944
    CUCM Serviceability API, 945
    PAWS API, 944
    UDS API, 945
  Unified Contact Center (Cisco), 943
  Unified Contact Center (Cisco), 943
<unlock> operations, NETCONF, 720–721
unmounting Linux file systems, 127
unstructured data, CLI, 19
until-do loops, 239–240
update command, Linux, 86–87
upgrading
  IOS XE with Ansible hosts/variables, 1057–1058
  local Git repositories, 314–315
  remote Git repositories, fetching updates, 314–315
uploading files via SFTP, 543
uppercase/lowercase characters, matching, regular expressions (regex), 187–188
URI (Universal Resource Identifiers), 389, 431, 432–436
URL (Uniform Resource Locators), 431, 432
  API, 945
  NETCONF, 733–734
urllib packages, Python and HTTP, 458–463
urls.py files, Django, 343
URN (Uniform Resource Names), 431–432
user space (userland), Linux, 23–24
useradd command, Linux, 138–140
userdel command, Linux, 141
user/group management, Linux, 136–138
  changing passwords, 141
  creating groups, 141
  creating new users, 138–141
  deleting groups, 141–142
  deleting users, 141
  getting user information, 136
  modifying user details, 142
  setting user passwords, 138–141
usermod command, Linux, 142
usr directory, Linux, 37
usr/bin directory, Linux, 37
usr/local directory, Linux, 37
usr/sbin directory, Linux, 37
UTF-8 encoding, HTTP/1.1 authentication, 472–473
utility modules, Ansible, 1003

validate capability, NETCONF, 733
<validate> operations, 724–725
validate_person() function, Python, 307–308, 309
validating
data, JSON schemas, 609–614
  JSON schemas, 596–597
  NETCONF configurations, 724–725
  XML, 562–563
    DTD, 563–565, 574
    XSD, 565–574
value assignments
  lists, Python, 286–288
  variables, 218–219
value fields (XML), space characters in, 562
values, XML, 555
var directory, Linux, 37
variables
  Ansible, 999
    Boolean variables, 1006
      checking for substrings with conditional statements, 1021–1022
      checking with conditional statements, 1019–1021
      conditional statements with loops and variables, 1027–1033
      defining from external files, 1007–1009
defining in inventory files, 1009–1011
defining in playbooks, 1005–1006
dictionary variables, 1007
importing from external files, 1007–1009
Jinja2 templates, 1042–1043
list variables, 1007
setting dynamically, 1011–1013
string variables, 1006
types of, 1006–1007
Jinja2 templates, 1042–1043
Linux scripting, 217–218
declaring, 218–219
one-dimensional variables, 218
value assignments, 218–219
nx-os_config modules, 1090–1091
Python, 270–271
deleting, 272–273
dynamic data type allocation, 271–272
scope, 272
vendor/API matrix, network programmability, 957–958
vendor-specific (native) YANG modules, 666–669
verifying
data planes
SR, 830–831
SR-TE, 842–843
Docker, 320–322
IOS XE with Ansible
ios_command module, 1058–1060
verifying management, 1057
verifying operational data, 1058–1060
IOS XR with Ansible
verifying management, 1074–1075
verifying operational data, 1074–1078
NETCONF
configuring, 1103–1107
operational data, 1098–1103
NX-OS with Ansible
collecting show output with nxos_command, 1086–1088
configuring with nx-os_* modules, 1093–1095
configuring with nx-os_config modules, 1086–1088
interactive commands, 1088–1089
verifying management, 1085–1086
verifying operational data, 1086–1089
Python code, 269
YANG modules, downloaded, 665–666
version command, Ansible, 991–992
version control, application development, 311
vgdisplay command, Linux, 132, 133–134
viewing
Linux files, 41–46
routing tables, Linux, 163
views.py files, Django, 343
virtual environments
creating with virtualenv tool, 332–333
Flask installations, 345–346
virtualenv tool, 331
installing, 331
virtual environments, creating, 332–333

virtualization
defined, 8–9
Docker containers, 317–318
Network Programmability and Automation toolbox, 17
single sources of truth, 11–12
VLAN, awareness, 8–9
VLAN (Virtual Local-Area Networks), awareness, 8–9

volumes, Linux
logical volumes, 132–133
mounting, 134–135
physical volumes, 130–131
volume groups, 132, 133–134

VPN (Virtual Private Networks), IP
VPN and JSON schemas, 597–598, 601–602, 607–609

W

web application development, 250–251
web/API development, 336–337
back end development, 336
Django, 337

application web servers, 338–339
creating applications, 341–345
demo applications, 343–345
installing, 337–338
migrations, 338–339
models.py files, 343
serializer.py files, 343
settings.py files, 339–341
starting new projects, 337–338

urls.py files, 343
views.py files, 343

Flask
accessing in-memory employee data, 347–349
installing in virtual environments, 345–346
retrieving ID-based data, 349–350
simple applications, 346–347

front end development, 336
Postman, 345
Webex Board (Cisco), 943
Webex Cloud Calling (Cisco), 942
Webex Contact Center (Cisco), 943
Webex devices, xAPI, 946
Webex Meetings (Cisco), 943
REST API, 945–946, 948–954
TSP API, 945
URL API, 945
XML API, 945
Webex Room Series (Cisco), 944
Webex Support (Cisco), 943
Webex Teams (Cisco), 942, 945–946, 948–954
Webhook Alerts API (Meraki), 922
webhooks, 882, 933, 935
well-formed XML documents, 562
westbound API, 883, 933
while loops, control flow, 304–306
while-do loops, 237–239
whitespaces, HTTP messages, 416
Wind River, Linux distributions, 26
with-defaults parameter, RESTCONF, 771
writable-running capability, NETCONF, 732
XML (Extensible Markup Language) 1191

x11 channels, SSH Connection Protocol, 520
xAPI (Experience API), 946
XML (Extensible Markup Language) API, 945
attributes, 558, 568
AXL API, 944
basic XML document example, 554
child/parent relationships, 555–556
comments, 558
data types, 567–568
declarations, 566–567
DTD, 563
example of, 563–564
group XML/DTD files, 564–565
formatting rules, 561–562
history of, 553–554
leading spaces in XML documents, 555
namespaces, 559–561
nesting
format, 561–562
relationships, 555–556
overview, 553–554
predefined entries, 557
Python and XML processing, 580
dictionaries, 583–585
element mergers, 585–587
element name/attribute extraction, 581–582
properties/methods, 580–581
rerunning processing for updated documents, 587–588
script creation, 580–581
value extraction, 582–583
space characters in value fields, 562
tags
child/parent relationships, 555–556
creating, 557
defined, 555
end tags, 555
predefined entries, 557
prefixes, 560–561
start tags, XML, 555
using multiple times, 558–559
values, 555
usage, 553–554
validation, 562–563
DTD, 563–565, 574
XSD, 565–574
values, 555
well-formed XML documents, 562
XPath, 574, 575
expressions, 575–576
expressions, absolute path, 576
expressions, absolute path and multiple outputs, 577
expressions, anywhere selection, 576
expressions, path definitions, 578
expressions, predicates, 577
logical operator values, 577
nodes, 574–575
XSD, 565
attribute validation, 570–571
attributes, 568
complex elements, 570–573
content validation, predefined values, 568–569
content validation, regexp, 569
data types, 567–568
declarations, 566–567
DTD comparisons, 574
element validation, 567
example of, 565–566
pattern validation, 569–570
XSLT, 578, 579
elements of, 578
stylesheets, 578–579
YAML versus, 615–616
XOR operator (^), Python, 281
XPath, 574
expressions, 575–576
absolute path, 576
absolute path and multiple outputs, 577
anywhere selection, 576
path definitions, 578
predicates, 577
logical operator values, 577
NETCONF
capabilities, 735
filters, 710–712
nodes, 574–575
syntax elements, 575
XSD (XML Schema Definition), 565
attribute validation, 570–571
attributes, 568
complex elements, 570–573
content validation
predefined values, 568–569
regexp, 569
data types, 567–568
declarations, 566–567
DTD comparisons, 574
element validation, 567
example of, 565–566
pattern validation, 569–570
XSLT (XML Stylesheet Language Transformation), 578, 579
elements of, 578
stylesheets, 578–579
xz archiving utility, Linux, 67, 69–70, 72–73

Y - Z

YAML (YAML Ain't Markup Language), 615
anchors, 624–625
collections, 618–620
comments, 616
configuration files, building, 635–637
data readability, 616
data representation, 615
data streams
saving to files, 629
sorting, 630–631
data types, 616
example of, 625–626
files
creating, 616
extensions, 616
rules for creating, 616
Jinja templates, 635–637
JSON versus, 615–616
mappings, 618–620
merge keys, 624–625
multiple documents, loading, 632–633
nodes, 617
performance, 616
purpose of, 616
Python
PyYAML, 626–628
saving data streams to files, 629
serializing Python objects, 628–629
sorting data streams, 630–631
yaml.dump() method, 628–631
yaml.load() method, 631–632
yaml.load_all() method, 632–633
yaml.scan() method, 633–635
scalars, 620–621
sequences, 618–620
speed, 616
starting/closing documents, 616–617
structure of, 617–618
tags, 617, 621–624
usability, 616
XML versus, 615–616
yaml.dump() method, 628–631
yaml.load() method, 631–632
yaml.load_all() method, 632–633
yaml.scan() method, 633–635
YANG (Yet Another Next Generation), 17
content layer, NETCONF, 725–729
data modeling, 639–640, 642
importance of, 640–642
modules, 642–644
modules, augmentation, 656–658
modules, built-in data types, 647–648
modules, cloning, 665
modules, derived data types, 648–649
modules, deviations, 658–662
modules, home of, 664–666
modules, IETF YANG modules, 670–671
modules, native (vendor-specific) modules, 666–669
modules, OpenConfig YANG modules, 671–673
modules, structure of, 644–646
modules, verifying downloaded modules, 665–666
nodes, 649
nodes, container nodes, 647–648
nodes, grouping, 654–656
nodes, leaf nodes, 649–651
nodes, leaf-list nodes, 651–652
nodes, list nodes, 647–648
pyang, 673–679
pyang.JTOX drivers, 683–687
pyangbind, 679–682
YANG 1.1, 662–663
JSD, 595
NETCONF/YANG
configuring, 1103–1107
netconf_get module, 1103–1107
network programmability, 973–978
verifying operational data, 1098–1103
RESTCONF
data resource, 753–756
schema resource, 750–753
YANG library version resource, 758
RESTCONF/YANG, network programmability, 978–987
YANG 1.1, 662–663
YUM (Yellowdog Updater Modified), 95, 101
commands list, 102–103
sudo yum remove httpd command, 109
sudo yum update command, 110
yum info command, 104–105

yum install command, 106–109
yum list command, 103
yum repolist all command, 110–111
yum search command, 103–104
yum-config-manager command, 111–112