Python Networking Solutions Guide

Leverage the Power of Python to Automate and Maintain your Network Environment

Tolga Koca

Python Networking Solutions Guide

Leverage the Power of Python to Automate and Maintain your Network Environment

Tolga Koca

Python Networking Solutions Guide

Leverage the Power of Python to Automate and Maintain your Network Environment

---

Tolga Koca

www.bpbonline.com

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN: 978-93-5551-361-8

www.bpbonline.com

Dedicated to

My Lovely Family:

Merve Aydin Koca

Ali Seydi Koca

Guluzar Koca (R.I.P.)

Duygu Koca

Yesim Eren

About the Author

Tolga Koca has 10+ years of experience as a Network, Cloud, and DevOps Engineer. He has worked with Internet Service Providers (ISPs) and Enterprise Companies and also created an online learning platform focused on Network Automation, Cloud, and DevOps training (networksautomation.com). He believes in sharing his knowledge with others, and he practices this by giving live webinars.

About the Reviewer

Pravesh Kumar Sharma, a long-time cybersecurity practitioner, Cyber Security Specialist in the Indian Air Force. Enrolled just after completing 10+2, he graduated in Electronics and Communication Engineering from the Institution of Engineers Kolkata. After having 10+ years in web programming and networking, he switched over to offensive security as a pentester.

He is the winner of many capture the flags hackathons, an honorable mention of the SANS hacking challenge. He is an avid learner and stood top 1% in Indian Institute of Technology Madras-driven NPTEL courses.

A Certified Information Systems Security Professional(CISSP) and Red Hat Certified Engineer(RHCE) person, he has an interest in solving problems at its grassroots levels. Python and Powershell are his favorite scripting languages. Besides this, he is MTech Software Systems from Birla Institute of Science and Technology(BITS) Pilani Rajasthan.

His name appears in TryHackMe top 1% worldwide. You can connect with him on **https://www.linkedin.com/in/pravesh-kumar-sharma-infosec/**

Acknowledgement

There are a few people I want to thank for their support during the writing of this book. First, I would like to thank my lovely wife, Merve, for always supporting me when I was stuck technically and mostly emotionally. And thanks to all my family members who supported me from the beginning to the end of the book.

I also want to thank Ozgur Kok for his outstanding mentorship and for always encouraging me.

Finally, I would like to thank Serina Haratoka for guiding me to unleash the power in myself and lightening my way during this period.

Also, special thanks to my cute cats, Mango and Kiwi, who always supported me from their boxes next to me while I was writing.

I want to thank BPB Publications, for their positive and encouraging support made this book to become true.

Preface

This book shows the Python programming language's importance and power to automate network devices such as routers, switches, firewalls, system devices like Linux servers, and cloud devices like the AWS platform. It shows to manage and configure thousands of devices with a single script, saving time and preventing faults.

This book covers network automation with Python specifically for Network, System, and DevOps engineers. It explains the Python basics from scratch with various features and modules. It covers the most helpful connection methods to login to multiple devices concurrently and manages them with scripts. It explains creating a customized network automation tool with many scripts.

This book is divided into 11 chapters. It covers network automation and Python basics, connecting devices with Python, managing devices by scripts, creating a network automation tool, etc. The detailed chapter information is listed below.

Chapter 1 covers the fundamentals of network automation and Python programming language basics. You will learn to install all the tools and packages for different operating systems. Then, you write a basic Python script and execute it.

Chapter 2 covers the essential Python functions for beginners and data types that will be used in later scripts. There are many data types in Python, focusing on the most used ones, including their methods for network automation. Then, it continues with the Python statements and conditions to create the main structure of the scripts.

Chapter 3 explains various built-in and 3rd party Python modules. File handling modules will be shown to create, modify and delete text, word, or excel files. One of the most important modules of the manipulation, which is the RE module, is explained in this chapter. It deeply presents the RE module with its functions, sets, and other features. The last part explains some advanced features of Object Oriented Programming (OOP) of Python to use in the more complex scripts.

Chapter 4 focuses on the Python connection modules and script examples. It explains the netmiko, paramiko, and telnetlib modules to connect to the network devices. There are various examples of collecting logs from devices and creating customized tools explaining each script code line-by-line. These examples are focused on connecting Cisco network devices, but these scripts can also be used with other vendor products.

Chapter 5 focuses on configuring network devices with some automation modules such as Jinja2, NAPALM, and nornir modules. These modules make configuring devices a more advanced and automated way. They create YAML files to create simple scripting files to make network automation easy.

Chapter 6 explains the file transfer protocols with the necessary 3rd party modules. You can login to devices with SSH, FTP, SFP, and SCP protocols and upload or download multiple files from network devices with scripts. It also focuses on plotting module the network data, such as device CPU values and the interface bandwidth utilization to the plot.

Chapter 7 focuses on upgrading and rebooting devices, collecting alarms, communicating with devices by the SNMP protocol, sending email notifications, and making reachability tests such as ping and traceroute for network and system devices.

Chapter 8 covers the system device management. It explains to create a Linux server environment step-by-step. It focuses on maintaining these servers by collecting server data such as CPU, memory, interface, and file information. It also explains configuring multiple servers by installing software packages, user management, rebooting servers, and managing the server processes.

Chapter 9 covers the security features and services for the network and system devices. It has various example scripts with explanations. Examples of scripts are to manage security services in Linux servers, manipulate network packets, check security logs, and capture network packets.

Chapter 10 explains to create a network automation tool. It's a command-line interface (CLI) based tool that combines several scripts to automate, maintain, and configure devices.

Chapter 11 focuses on network automation in Amazon's AWS Cloud Platform. It explains the Boto3, an AWS management module that manages

EC2 instances, S3 buckets, and IAM user management.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/33ehv8a

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Python-Networking-Solutions-Guide**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

1. Introduction to Network Automation
- Structure
- Objectives
- Introduction to network automation
  - *Benefits of network automation*
  - *Future of networking*
- Introduction to Python
  - *Python usage area*
- Python installation
  - *Python for Windows*
  - *Python for Linux*
  - *Python for MAC*
- Running Python codes
- Pycharm installation for Windows
- Install and import Python modules
- Conclusion
- Multiple choice questions
  - *Answer*
- Questions

2. Python Basics
- Structure
- Objectives
- Print and ınput functions
  - *Print()*
  - *Input ( )*
- Data types
  - *String and integer*
  - *String methods*
  - *List*
  - *List methods*
  - *Dictionary*

Dictionary methods
Statements and conditions
If condition
For statement
While statement
Break and continue statement
Range statement
For else statement and nested loops
Try…except statement
Conclusion
Multiple choice questions
Answers
Questions

3. Python Networking Modules
Structure
Objectives
File handling
Open function
OS module
Word files
Excel files
RE modules
RE module functions
Special sequences
Sets in the RE module
Advanced topics of Python
Functions
Functions with parameters
Functions with default parameters
Call variables from functions
Creating modules
Classes
Conclusion
Multiple choice questions
Answers
Questions

4. Collecting and Monitoring Logs
Structure
Objectives
Connection modules
SSH connection
Paramiko module For SSH
Connect 1 device with Paramiko
Running configuration commands with Paramiko
Connect to multiple devices with Paramiko
Netmiko module for SSH
Connect a single device with Netmiko
Connect to multiple devices with Netmiko
Telnet connection
Telnetlib module for telnet
Connect to multiple devices with telnetlib
Netmiko module for telnet
Collecting logs
Collecting version and device information
Collecting CPU levels
Finding duplicated IP address
Collecting logs with multithreading
Tools and calculators
IP address validator
Subnet calculator
Conclusion
Multiple choice questions
Answers
Questions

5. Deploy Configurations in Network Devices
Structure
Objectives
Configure network devices
Configuration of interfaces
Replacing configurations on files
Configure devices with Jinja2 template
Introduction to Jinja2 template

Introduction to YAML language
Rendering Jinja template with a YAML file
Configure devices with Jinja
If statement in Jinja
Configure devices with Napalm module
Collect logs from devices with NAPALM
Configure devices with NAPALM
Configure devices with Nornir module
Configure inventory in Nornir
Connection to devices with Nornir-Netmiko
Connection to devices with Nornir-NAPALM
Configure devices by Nornir and Jinja template
Conclusion
Multiple choice questions
Answers
Questions

6. File Transfer and Plotting
Structure
Objectives
File transfers
Backup configuration file with SSH
File transfer with FTP connection
File transfer with SFTP connection
File transfer with Netmiko SCP connection
Netmiko SCP connection with concurrent module
File transfer with Nornir SCP connection
Backup configuration file with SCP
Plotting data
Plotting CPU levels
Plotting interface bandwidth
Conclusion
Multiple choice questions
Answers
Questions

7. Maintain and Troubleshoot Network Issues

- Structure
- Objectives
- Upgrade network devices
- Alert alarms in devices
- Collect logs with SNMP
- Send logs via email
- Reachability test to network devices
    - *Ping test script*
    - *Traceroute test script*
- Conclusion
- Multiple choice questions
    - *Answers*
- Questions

8. Monitor and Manage Servers

- Structure
- Objectives
- Implement server environment
    - *Download VMware player and Ubuntu*
    - *Install Ubuntu on VMware*
    - *Activate SSH connection*
- Maintain Linux servers
    - *Collect logs via syslog*
    - *Login servers with secure password*
    - *Collect CPU and memory levels*
    - *Collect interface information*
    - *Collect type and permission of files*
- Server configurations
    - *Create users in servers*
    - *Install packages*
    - *Transfer files with Paramiko*
    - *Reboot servers concurrently*
    - *Stop running processes by script*
- Conclusion
- Multiple choice questions
    - *Answers*
- Questions

9. Network Security with Python
Structure
Objectives
Activate security services
Install and activate the “Firewalld” service on servers
Configure firewall settings on servers
Create access lists in network devices
Manipulate network packets with scapy
Check logs and configurations
Check CPU levels periodically with Crontab
Check router configuration for insecure passwords
Check port security configuration in routers
Collect packets from ports with Pyshark
Conclusion
Multiple choice questions
Answers
Questions

10. Deploying Automation Software
Structure
Objectives
Introduction to InquirerPy module
Automation tool design
Create main tool script
Create subtask scripts
Network device scripts
Server scripts
Other remaining scripts
Conclusion
Multiple choice questions
Answers
Questions

11. Automate Cloud Infrastructures with Python
Structure
Objectives
Cloud environment deployment

Introduction to AWS
Installation of Boto3 and AWS CLI
EC2 instance management
Manage EC2 instances with Python
Connection to EC2 instances
S3 bucket management
EBS volume management
Manage EBS volumes
Create snapshots of EBS volumes
Attach EBS volume to EC2 instance
IAM user management
Conclusion
Multiple choice questions
Answers
Questions

Index

CHAPTER 1

Introduction to Network Automation

This chapter will focus on the basics of network automation and understanding the current and future of networking in the industry. It will explain the benefits of using automation in network environments for companies and engineers. We will learn the basics of Python programming and the usage areas of the language. We will install the necessary packages and tools for the network automation.

Structure

In this chapter, we will cover the following topics:

- Introduction to network automation
  - Benefits of network automation
  - Future of networking
- Introduction to Python
  - Python usage area
- Python installation
  - Python for Windows
  - Python for Linux
  - Python for Mac
- Running Python codes
- Pycharm installation for Windows
- Install and import Python modules

Objectives

This chapter aims to introduce network automation and Python programming language. We will download and then install the Python package for each OS as Windows, Linux, and macOS. We will also look at how to install the Pycharm tool, which is an **Integrated Development Environment** (**IDE**). We will write our first Python code, and finally, install third-party modules and import them into our Python codes.

Introduction to network automation

Before explaining network automation, we should start with what automation is. In simple terms, automation is the use of technology to perform tasks automatically. It has been rising since the 1950s. Many companies were using automation in different fields.

We can say that network automation is performing tasks automatically by reducing human interaction with network devices. With network automation, there will be no more manual steps, like making **command line interface** (**CLI**) connections and running commands manually to manage network devices. Network automation scripts are pre-programmed for specific purposes like software upgrades, collecting device logs, file transfers, and comparing configurations.

Python, Perl, Bash or Go scripting languages that can be used for network automation. These scripting languages are all open-source and free. A network engineer must learn one of these languages to write scripts in network automation.

There are also open-source network automation tools like Ansible, Puppet, and Chef. These are network automation frameworks with libraries for specific demands or vendors, which make network automation simpler.

Network automation is also known as **network orchestration**. We can organize, manage, and troubleshoot our whole network structure or orchestrate with network automation scripts and tools.

So, why do we need network automation? In recent years, the internet has been growing exponentially, and it will continue. Business demands are always changing, and maintenance has become harder. You can think that if a mobile app does not open for 5 to 10 seconds, you may directly delete it. The delay even in milliseconds can make big problems in many businesses.

But we are human; we can always make mistakes, and manual maintenance is very slow. This is where automation enters our life. Network automation is still in the early stages, but tech pioneers like Amazon, Google, and Facebook are using it very efficiently.

Benefits of network automation

So, what will change after we use automation in our network environment? Is it necessary for each environment? First, we need to check the details of the current network environments. For almost 20 years, network configurations have not changed dramatically, and a network engineer's role has also stayed the same for a decade. There are **Network Management Systems** (**NMS**) tools that make monitoring and configuration more automated, but those are mostly vendor-specific tools. They are not flexible to the new requirements of customers. Additionally, acceptance of new technologies in networking has come only slowly because even small mistakes get us in big trouble.

With **Software Defined Networks** (**SDN**), all network infrastructures are evolving. And with SDN and machine learning, networks are becoming more flexible and easier to maintain. Network automation will be more important in the near future.

Network automation has many benefits for engineers and companies, and it improves engineering quality with Python language.

- **Reduce the number of human errors**: With network automation, we can reduce the frequency of human mistakes in operations. Operations are done within a limited time. One command mistake can affect many things in the network and can also cause service interruption. We can use scripting to eliminate it. There will be no more mistakes in command entrance in devices. The idea is not to reduce network engineers in IT teams but to reduce human mistakes; we still need network engineers to make the network automation work.
- **Improved efficiency**: We can ensure faster operations and troubleshooting. Collecting logs is often painful, and it takes a long time if there are multiple devices to collect. All tasks are done manually via SSH tools like SecureCRT or Putty. But with the prepared Python scripts, we can collect any logs from any device with

just a couple of clicks. That means no more copy/paste work. Machines are much better than us at repetitive things, so the workload of network engineers is reduced for repeatable actions.

- **Create your own automation tool**: We can create our automation tool for specific expectations. For example, we need to check the CPU levels of 500 devices. If we have an NMS tool, we can collect CPU data of all devices. Otherwise, it's almost impossible to enter devices by CLI one by one and collect CPU data. Even if we have an NMS tool, it has a limitation: we cannot run every log or feature in that tool. If our NMS tool has no feature to collect CPU levels, we cannot perform this task.

  But with Python, we can write scripts for any of the logs or statistics to collect from devices. We can collect any logs we require. We can write one time, modify for new purposes and use it numerous times.

- **Manage logs**: We can manage logs efficiently. Because the logs are dummy when we collect them, we need to specify them, like filtering, sorting, or checking specific values or lines. With Python scripting, we can preset all of them, and then just run scripts to convert logs to readable data for us.

  For example, we need to upgrade core routers, so we can collect logs before and after the upgrade as pre-checks and post-checks. But the logs are dummies. If we run the show version command in a cisco router, only one line is important for us: the one that has the software version information. Then, we must get the backup of the device configuration. That's a lot of work to do, and it's painful if we have many devices.

  But on the other hand, we can create a script that logs in to devices; collect logs; gets the necessary information like device version, CPU level, BGP neighborship summary, and interface status; and puts all the information in Excel and compares it for us. This way, we can determine what changed after the upgrade in just seconds. By scripting a couple of lines, we can get the important parts from the collected logs and make them easy to read for us.

Future of networking

As technology is growing and changing all over the world in all sectors, telecommunication, data center and cloud companies are also evolving. So, the role of network engineers is changing. In the near future, network engineers must know at least one scripting language and automation tool.

The change starts with SDN and virtualization. Network devices or systems will find the issue and take the responsibility to solve the issues by themselves with machine learning. New vendors and companies will be involved, and network structures will be smarter. BGP, MPLS or any other network knowledge will not be enough for a network engineer because software and scripting will enter daily life rapidly.

Introduction to Python

According to Python official (**www.python.org**), Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It's an open-source programming language that's free of cost; here are its characteristics:

- **Interpreted**: Python is an interpreted language. It means that the code that is written with this high-level programming language is converted to machine code and runs each task line-by-line. Python converts each line to the machine's readable code. It is also easy to compile; Python does not need any compiler like C++ or Java. So, we can develop code much faster than other languages.
- **Easy syntax**: Python uses indentations. It's easier to read instead of other programming languages.
- **Increasing community**: Python has a very good community. You can find anything on GitHub or StackOverflow.
- **Platform independent**: It is also platform-independent, so you can use it for different operating systems.

Python usage area

Python has a big usage area. In many sectors, we can see Python programming. We can develop desktop mobile apps, back-end servers, games, audio, video apps and more, but Python is mainly used for data science, automation, and web development.

The most popular usage of Python is for data science. In data science, there are many sub-areas like data analysis, data visualization, machine learning, and artificial intelligence. Data science is one of the hottest trends in recent years. And in the future, it will be much more important in the tech world, which means the popularity of Python will increase all the more.

And in automation, Python is the most popular language. Here, we are interested in Python. With scripts, we can do network automation easily. Ansible, a network automation tool, is also written in Python.

Figure 1.1 shows questions asked each month as a percentage, based on Stack Overflow, which is one of the biggest QA platforms for software developers. You can easily see how Python is becoming the most popular programming language in the future:

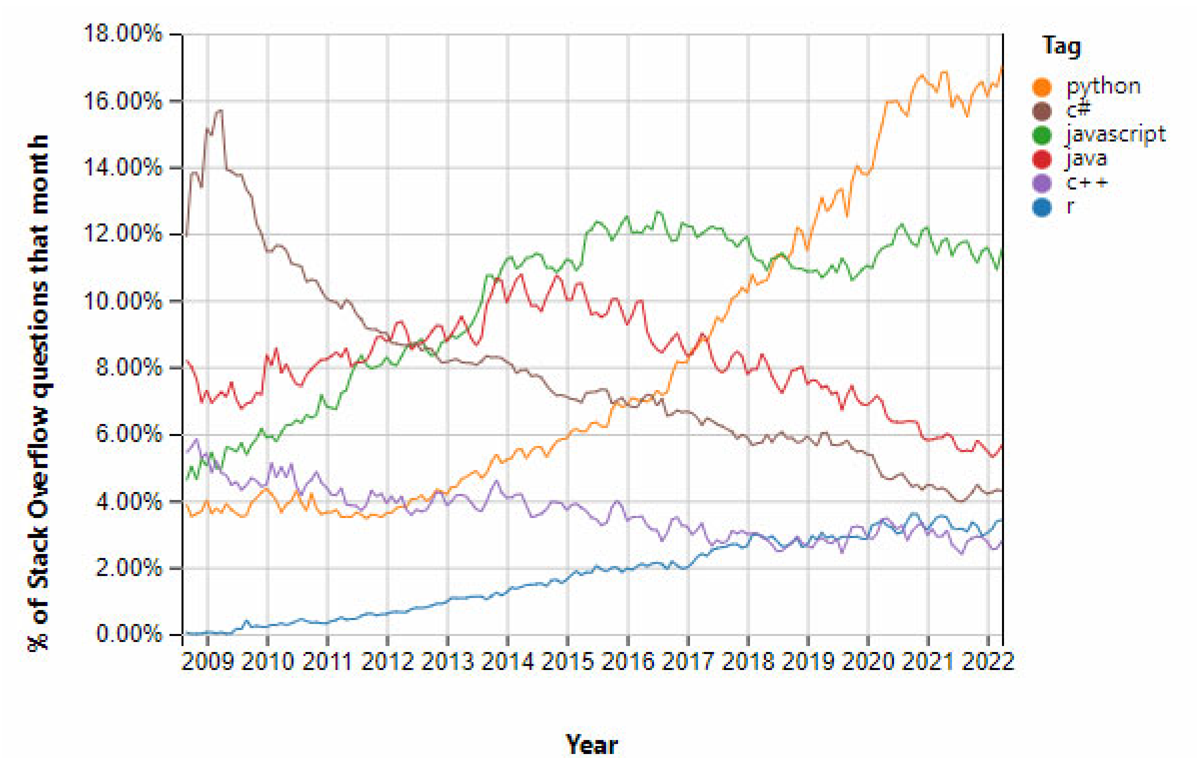


Figure 1.1: *Stack Overflow questions by month - insights.stackoverflow.com/trends*

Python installation

Note that Python 3.9+ cannot be used on Windows 7 or earlier. Python 2.7 is the end of the Python 2.x series and is succeeded by Python 3. End-of-support of the Python 2 version expired on 01-01-2020.

Python for Windows

To use the Python programming language, we need to install the Python tool first. We can download Python by clicking on the following link from Python's official website:

https://www.Python.org/downloads/

When we enter the website, there is a **`Download the latest version`** for section on the top, as shown in *Figure 1.2*. If you enter this site using a Windows machine, you see it as a Windows download. If you enter it using a Linux or MAC system, you can see the specific OS download button:

Figure 1.2: *Download Python for Windows*

We can download the latest stable version of Python by clicking on the **`Download Python`** button; the latest release is currently 3.10.4.

You can also download other OS installations of Python below the **`Download button`**. If you want to download older versions of Python, you can scroll down on the same website to see all active Python releases. There are two main versions active as Python version 2 and version 3. Both versions have many releases. We will focus on Python version 3 in this book.

After the download is finished, when we open the installer, we need to check the **`Add Python 3.10 to PATH`** box. By default, it's unchecked. When we install Python, it has not been added to **`Environment Variables`** in Windows Systems. So, the command prompt will not recognize the Python commands. By checking that box as shown in *Figure 1.3*, we can enter the Python command in the command prompt:

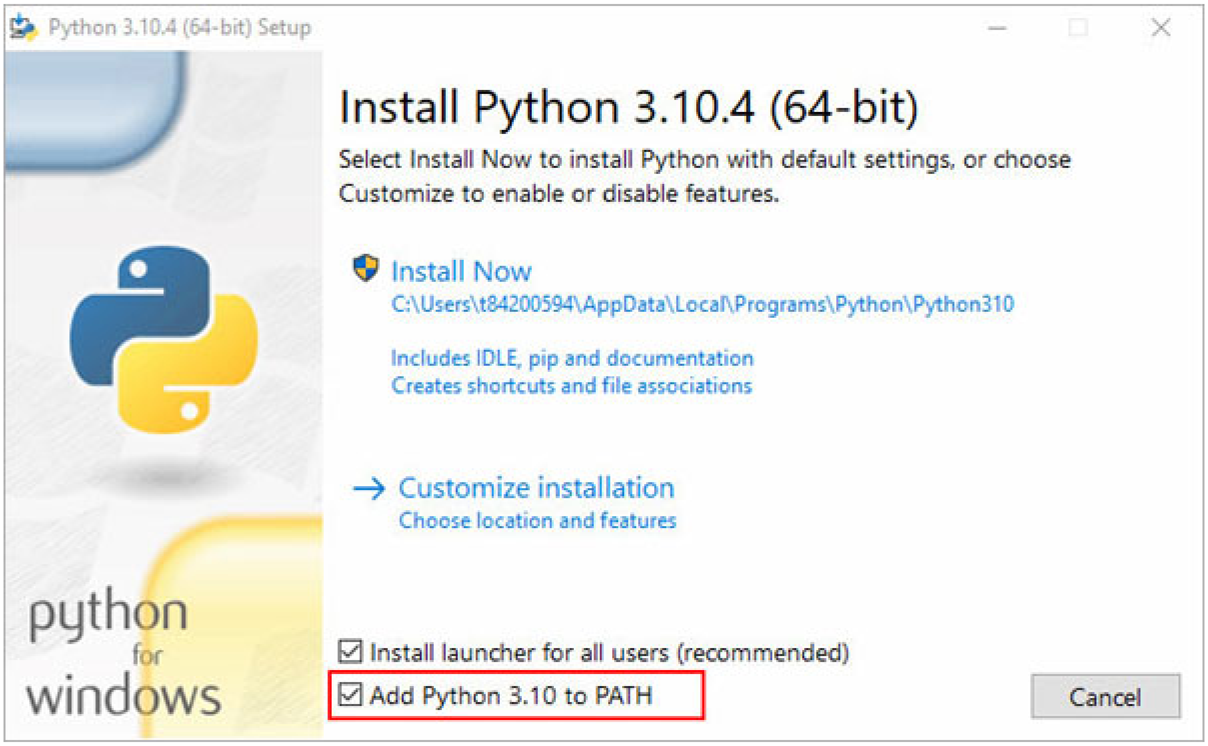


Figure 1.3: *Installation of Python*

After checking the box, we can install Python with the default settings by clicking on the **`Install Now`** button. If you need to install Python with custom settings, you can click on the **`Customize installation`** button.

When the installation finishes, we can close the installation screen and open the **command prompt** (**cmd**) by clicking on the Windows start button and writing **`command prompt`**. Alternatively, as a shortcut, we can click on the Windows button on the keyboard and the **`R`** key at the same time, and we will see the **`Run`** tool of windows will be opened. We can just write **`cmd`** and click on *Enter*.

We can write **Python --version** to verify the installation of Python in cmd. If you get Python word with the version information as an output, as shown here, it means that you installed Python successfully:

```
C:\> Python --version
Python 3.10.4
```

We can also start a Python session for simple lines of code by writing **Python** in cmd. There are three bigger signs in the last line of output. It means that we are inside a new Python session. We can write our code line by line here and can easily see the output without using any compilers:

```
C:\> Python
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41)
[MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Python for Linux

For many Linux systems, Python is an already installed package by default in the OS. For Ubuntu OS, Python version 3 is installed with Ubuntu OS. When we enter **Python3 --version** in the terminal and press **Enter**, you can see the current Python package version:

```
$ Python3 --version
Python 3.8.10
```

If we need to install Python from the package installer, we need to write the following command in the terminal:

```
$ sudo apt-get install Python3
```

Or we can update the Python3 package that is already installed with the following command:

```
$ sudo apt-get update
```

We can enter a new Python session with the **Python3** command in the terminal. For basic usage of Python, we can use this CLI or terminal. But for complex structures, it's better to use IDE tools like Pycharm or any text editor:

```
$ Python3
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
```

```
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Only Python version 3 is covered in this book, but if you need to use Python version 2, you can download it via the terminal. It will download the latest stable version 2 release:

```
$ sudo apt-get install Python
```

We can check the version with the **`Python version`** command and enter Python version 2 with the **`Python`** command in the terminal if necessary:

```
$ Python --version
Python 2.7.18
$ Python
Python 2.7.18 (default, Mar 8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Python for MAC

We can download Python for MAC devices from Python's official website, by clicking on the following link:

https://www.Python.org/downloads/

When we enter the website, there is a **`Download the latest version for`** section on the top. If you enter this site using your MAC device, you can see it as a MacOS download:

Figure 1.4: *Download Python for MAC*

After we download the package file, we can start the installation by double-clicking on the installer icon. And we can install Python3 by continuing the process on the installation window.

After the installation is finished, we can enter the **"Python3 –version"** command in the terminal to verify the installation.

```
% Python3 –version
Python 3.10.4
```

We can create a new Python CLI session by entering the **Python3** command in the terminal:

```
% Python3
Python 3.8.5 (default, Jul 21 2020, 10:48:26)
[Clang 11.0.3 (clang-1103.0.32.62)] on arwin
Type "help", "copyright", "credits" or "license" for more
information.
```

Running Python codes

We have a couple of options to run Python codes:

- We can create a new Python session from cmd (for Windows) and terminal (for Linux or Mac). We can write the Python code line-by-line. In each line, we enter a piece of code. For basic usage of Python, we can use it this way. But if we try to write scripts, this way is not sustainable. Each code proceeds in the same line. So, if we write the following example, in the first line, `5` assigns to `a`, in the second line, `10` assigns to `b`, and in the last line, we write a plus `b`, which is a calculation of `2` variables. And the result was `15`. So, the program shows the output after this line:

```
C:\> Python
>>> a = 5
>>> b = 10
>>> a + b
15
```

  If we write `a` variable and enter it, it will show the value of the `a` variable:

```
>>> a
5
```

- We can create Python files with text editors, like in notepad. We write the piece of code and save it as a `.py` example. The file extensions in Microsoft OS are hidden by default. If you create a file as `example.py`, it will be created as `example.py.txt`, which has a file type of a text document. So, you can follow these steps:

  Click on `File Explorer` | `View` | `Options`. Click on the drop-down arrow and click **change folder and search options**. Then, click on the `View` tab and uncheck hide extensions for known file types. You can also do it by typing one line in the CLI:

```
reg add C:\>
HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Adv
anced /v HideFileExt /t REG_DWORD /d 0 /f
0 = false, 1 = true
```

  If the `HideFileExt` value is set as 0, it means the file extension is visible. If the `HideFileExt` value is set as 1, the file extension is not visible, which is the default value:

Example 1.1: Simple Python File

```
example.py
a = 5
b = 10
print ( a + b )
```

After that, we can write the `Python` command with the filename to see the output of the code. Remember that you must be in the same directory with the Python file that you run or write the full path of the file.

```
C:\> Python Users/YOUR_USERNAME/Desktop/example.py
15
C:\Users\ YOUR_USERNAME \Desktop> Python example.py
15
```

There is no Python session here. Python directly starts and finishes the full code. This way is more convenient according to the first solution. This is because if we have many lines with multiple Python codes connected with different files, we can use them this way. But there are some missing parts here. We cannot debug or troubleshoot the coding issues with text editors.

- We have the best solution to eliminate the issues in the first three ways. There are many IDE tools for all programming languages. We can create projects and directories inside those projects, monitor the running code line-by-line, and debug or troubleshoot the issues inside our code with IDEs. Writing code is much easier with these tools. One of the most popular ones is the Pycharm IDE tool, which will be covered in this book.

Pycharm installation for Windows

Pycharm is an open-source tool owned by JetBrains Company. It's a code editor. We can download the Pycharm tool from the following link to the official website:

https://www.jetbrains.com/pycharm/download

We can use Pycharm on Windows, Linux and macOS devices. We can download it for specific OS. There are two download options, professional and community versions of Pycharm. Professional as paid version is used for scientific and web development of Python, including HTML, JavaScript, and SQL support.

The community version is totally free. This version is fairly enough for network automation:

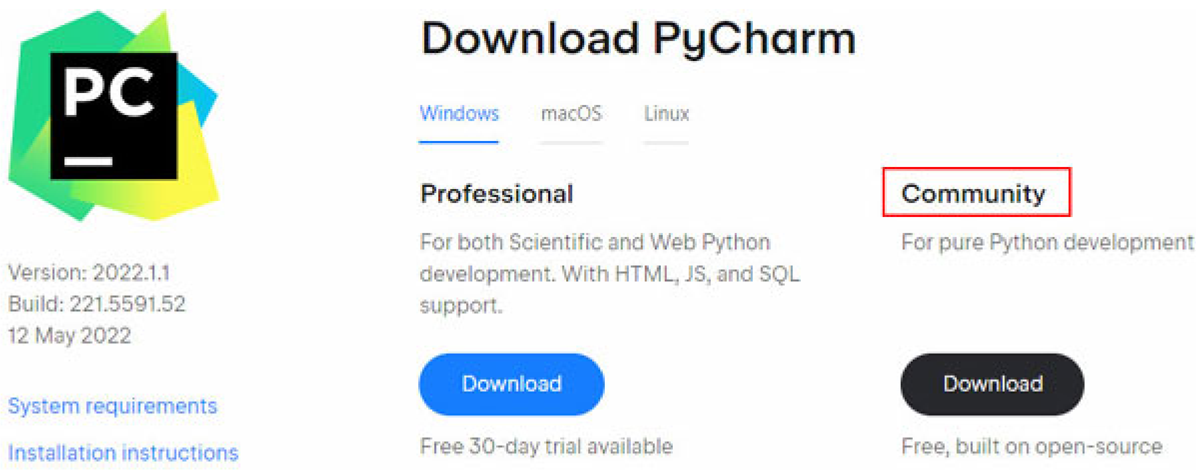


Figure 1.5: *Download Pycharm for Windows*

After we download the Pycharm community version as shown in *Figure 1.5*, we can start the installer just like installing a regular tool in Windows. After the installation is complete, we can open Pycharm. When we open Pycharm, it asks us to create a new project or open an old project. After we create a new project, on the **`project files`** tab, we can see all the files and directories inside our project.

We can create a new Python file on the project files tab by right-clicking on **`New/Python File`**. We create an **`example.py`** file and write three lines of code, as shown in *Figure 1.6*:

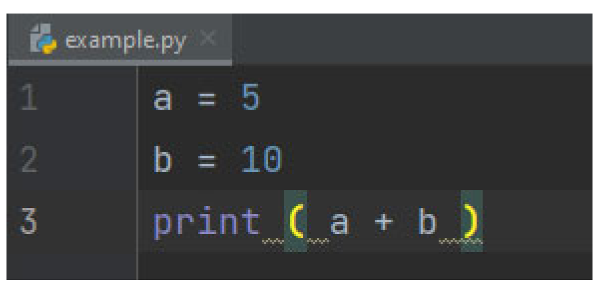


Figure 1.6: *Pycharm Code Example*

If we want to run this Python code, we have two options. We can press *Shift + 10* on the keyboard, or we can right-click on the code area and click on **`Run example.py`**.

When we run the code, a section will be opened: the **`Run`** section. We can see the output of the code here. Our simple code is the sum of the **`a`** and **`b`** variables. So, it's **`5`** plus **`10`**, and the result must be **`15`**. So, the output of our code is **`15`**. We can see the output in *Figure 1.7*:

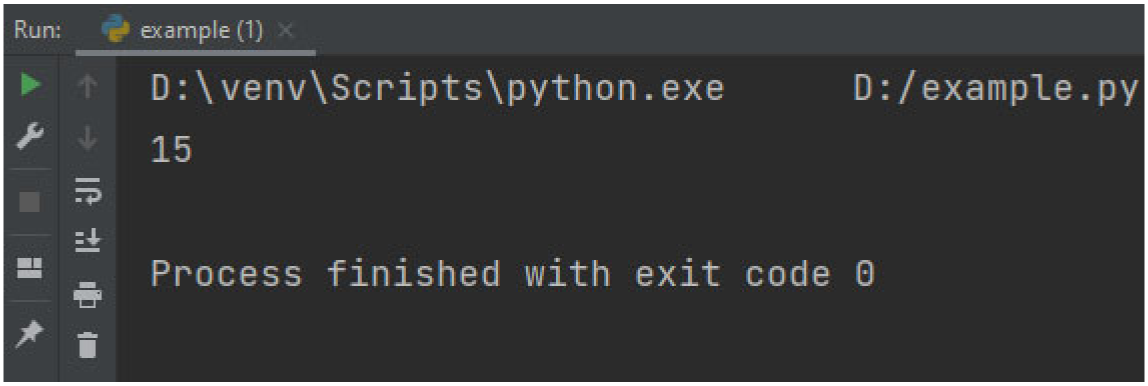


Figure 1.7: *Pycharm Code Output Example*

Install and import Python modules

In Python, we use libraries or modules to write our codes. There are standard built-in libraries and third-party libraries in Python.

Standard libraries are installed when we install Python to our PC. We don't need to install these libraries again. For example, the **`re`** module is one of the standard libraries in Python. It's used to check a set of strings that matches. This module will be covered in the next chapter. We don't need to install this module on our PC. We only need to import this module when we need to use it.

Third-party libraries or non-built-in libraries are additional libraries that we need to install on our PC if we need to use them. For example, the **`paramiko`** module is a third-party library. It's used to make an SSH connection to network and system devices. It's not installed during Python tool installation. If we need to use this module, we must install it, and then we can import our code to use. This module will be also covered in the next chapter.

To install a third-party library, we have two options in the Pycharm tool. We can click on the terminal tab in the following section of tools, as shown in *Figure 1.8*:

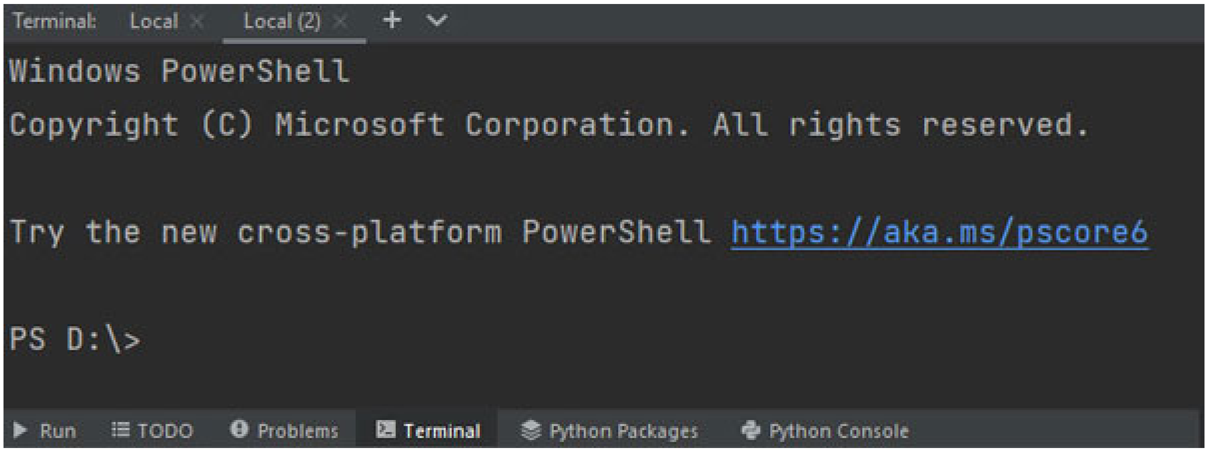


Figure 1.8: *Terminal in Pycharm*

In the terminal, we need to install modules with the **`pip`** command. We can write the command as **`pip install`** by writing the module name. In the following example, we try to install the **`paramiko`** module. If the module has not been installed yet, it will be downloaded and installed automatically. If the module is already installed, but a newer version is released, the **`pip`** command will download and install the latest version of that module:

```
C:\Sample_Project > pip install paramiko
```

If you write the mentioned command in the Pycharm terminal and get an error, you need to download the pip installation package to your PC. If your Python version is higher than v3.4 (released in 2014), you already have the pip package on your PC. If not, you must manually install the package. We can check whether the **`pip`** package is installed on our PC with the **`pip -V`** command. We can also check the package version.

```
C:\Users\USERNAME> pip -V
pip 22.2.2 from
C:\Users\USERNAME\AppData\Local\Programs\Python\Python310\lib\s
ite-packages\pip (Python 3.10)
```

You can run the following commands to install or update the **`pip`** package to the latest version on Windows and Linux:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
Python get-pip.py
```

Another way to install a third-party module in Pycharm is to install it via Pycharm GUI. To do that, go to **`File/Settings`** on tabs. Inside the **`Project/Project Interpreter`** section in the opening window, you can see all the modules that are installed under the current project, as shown in *Figure 1.9*.

We can also see the difference between the currently installed version and the latest version. So, if any of the modules have an update, we can see it in the same window. For example, we can see that the paramiko module is already installed in our project as a 2.7.2 version, which is an older version, as shown in *Figure 1.9*:

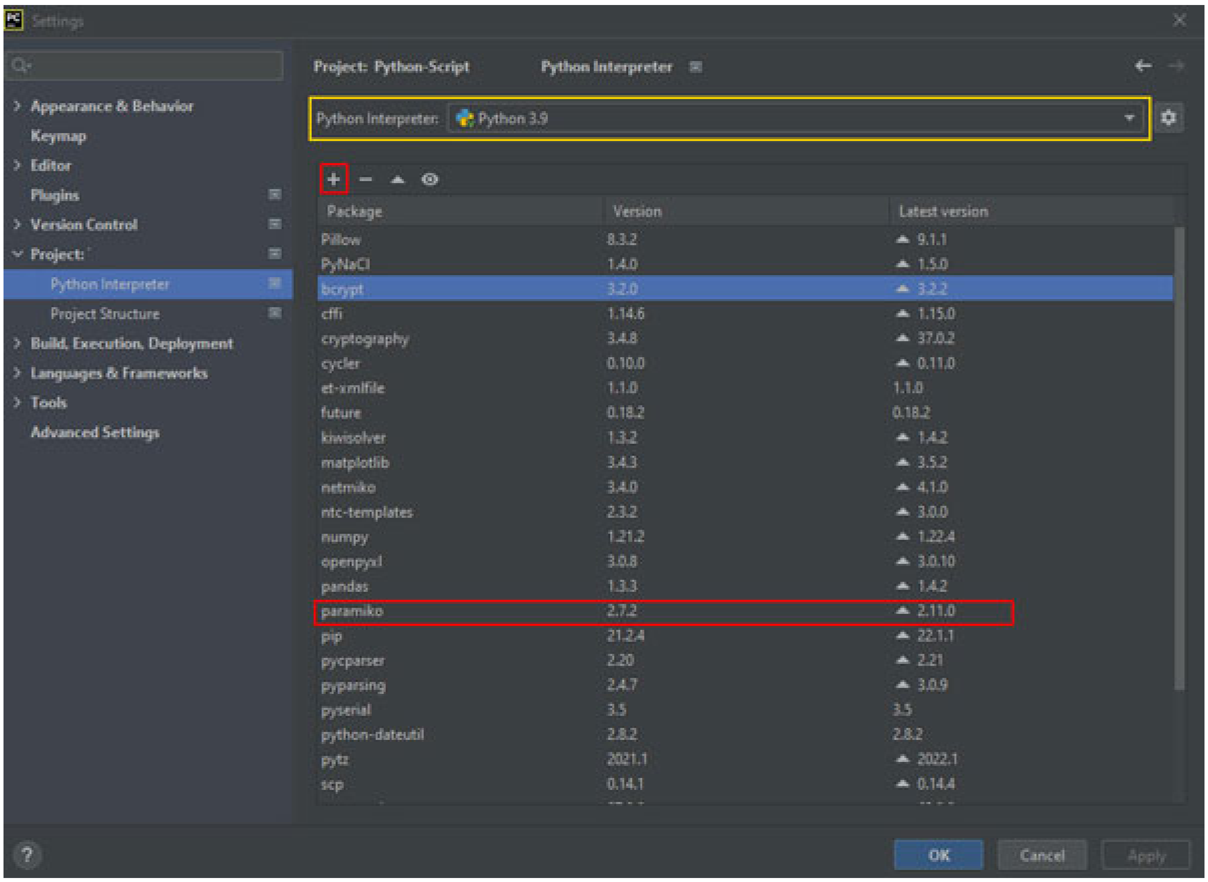


Figure 1.9: *Pycharm Module List*

In the Python interpreter section, labeled in yellow in *Figure 1.9*, we can change the Python interpreter. It means that we can use any interpreter in our current project. For example, we use many third-party modules in a project. And after that, we will create a new Python project. We can choose an interpreter for the old project when we create the new one, or we can change the interpreter in *Figure 1.9*.

We can install third-party modules by clicking on the plus character in the same window as the one shown in *Figure 1.9*. In the opening window, we can search for any of the modules to install. After we choose a specific module, we can click on the **Install Package** button, as shown in *Figure 1.10*:

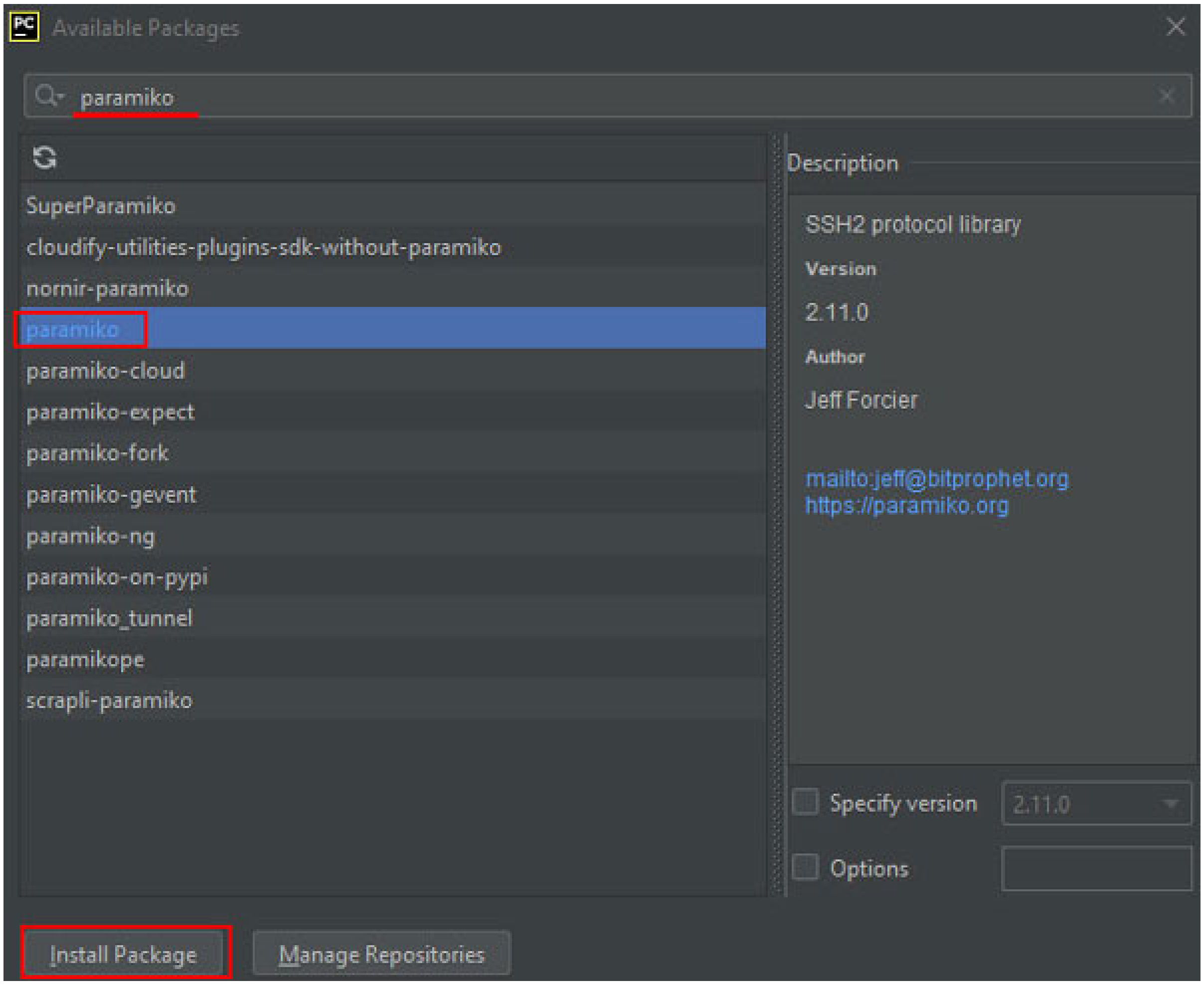


***Figure 1.10:** Installation of 3rd Party Modules*

The installation of modules is complete. Even if it is a standard or third-party library, we need to import each script that we use as a function from that library. For example, if we need to log in to a device with SSH protocol, we need to import the **paramiko** module at the beginning of our script. And if we need to use the **RE** module, we need to import it:

```
import paramiko
import re
```

After we import the modules, we can call any functions. Modules and functions will be covered in the next chapter in detail.

Conclusion

In this chapter, we learned what network automation is and how companies evolve according to network automation. We understood the benefits of automation, like reducing human mistakes and decreasing the workload for engineers. We introduced the basics of the Python language and looked at how to install the Python package and Pycharm tools to start network automation in our own environment. The version difference is an important topic in Python, and it is strongly recommended to start or continue with Python version 3. At the end, we learned the difference between built-in and third-party modules. We downloaded and installed these third-party modules and imported them into our code.

In the next chapter, we will start with the basics of Python programming language, like print and input functions, and data types and their methods, and also statements and conditions.

Multiple choice questions

1. What are the advantages of using network automation?
   a. Fewer human errors
   b. Faster operations and troubleshooting
   c. Reduced workload
   d. All of the above
2. Which of the following is not a feature of the Python language?
   a. Interpreted
   b. Open-source
   c. Object-oriented language
   d. Only works in Linux systems
3. What is the extension of a Python file?
   a. `.Python`
   b. `.pyt`
   c. `.py`
   d. `.pyth`

4. How to write a string in print function with Python version 3?

    a. `print Welcome to Network Automation`
    b. `print "Welcome to Network Automation"`
    c. `print ("Welcome to Network Automation")`
    d. `print "(Welcome to Network Automation)"`

5. Which command is used to download and install third-party modules?

    a. `pip install netmiko`
    b. `pip download netmiko`
    c. `pip add netmiko`
    d. `pip configure netmiko`

Answer

1. d
2. d
3. c
4. c
5. a

Questions

1. What is the benefit of using network automation for a company and for an engineer?
2. How is Python different from other high-level programming languages?
3. What is the process of importing third-party modules in Python?

CHAPTER 2
Python Basics

This chapter will focus on the basics of the Python programming language. This chapter will explain how to write simple Python scripts. We will write our first script examples and learn basic functions, data types, statements, and conditions in this chapter. We will build network automation scripts with the data types and statements that we will learn about in this chapter.

Structure

In this chapter, we will cover the following topics:

- Print and input functions
- Data types
    - String and integer
    - String methods
    - List
    - List methods
    - Set, tuple, and range
    - Dictionary
    - Dictionary methods
- Statements and conditions
    - If condition
    - For statement
    - While statement
    - Break and continue statement
    - Range statement
    - For else statement and nested loops

- Try – except statement

Objectives

This chapter aims to introduce basic Python functions that we use in many Python scripts. The chapter starts with print and input functions. It continues with the most important data types in network automation: string, integer, list, and dictionary and the methods that are used to manipulate the data. We compare the difference between other data types. Finally, we focus on conditions and statements like **`if`**, **`for`**, **`while`**, **`break`**, **`continue`**, **`range`** function, **`for…else`**, nested loops and **`try…except`** statements. We will write and explain several examples of these statements in detail.

Print and ınput functions

Before starting the Python basics, we can check two major and basic functions in Python: the **`print`** function and the **`input`** function. We will always use the **`print`** function to see the result of our codes. It's kind of an output of the result. Codes are written by functions in Python like in other programming languages.

Functions are shortcuts to codes. Function is a block of code. When we write a function to call, it runs the source code of a specific function.

For example, we have a print function, which is used to give the output when we run a code. It's just one word to call, but in the background, Python runs the full code of the **`print`** function. The developer only writes the function name, so the source code can be complex, but the usage is quite simple for functions. We just call to use them. Our code will be much simpler.

Another advantage of functions is that we can use them in repeatable things. So, there is no need to write the full code of function many times. It's enough to just call the function with its name.

Print()

The **`print`** function is one of the simplest and most useful functions in Python. It's used to display or show the value that we want as output. Its usage is also simple. After writing `print` as a word, we need to write an object inside a set of parentheses; this object can be a variable or a value.

For example, we can write code for calculation of variables like `a` and `b`, where a equals `10` and `b` equals `20`. And `c` equals `a` plus `b`. If we write the code like that, Python calculates `c` as `30`. But we cannot see any output because we didn't call the `c` variable with the `print` function. So, at each step of the script, we use the print function to see the result or even for troubleshooting and debugging the issues in our code. We can call the print function as many times as we need. As in *example 2.1*, we can call the `c` variable and then the `a` variable, so the output is `30` and `10`.

Example 2.1: Print function usage

```
a = 10
b = 20
c = a + b
print ( c )
print ( a )
```

Output:

```
30
10
```

There are many options to use the `print` function. We can write any character as letters, digits or special characters under the parenthesis. To do that, we must write all of them inside quotes:

```
print ( "Hello World" )
```

Output: `Hello World`

We can write multiple values by dividing them with commas. We still use quotes in the beginning and at the end of the value:

```
print ( "Hello", "World" )
```

Output: `Hello World`

We can assign a value to a variable and call it inside a `print` function, like in *Example 2.1*:

```
x = "Hello World"
print ( x )
```

Output: `Hello World`

We can write some values and variables with a dividing comma. As in the following example, values must be inside quotes, but if we call a variable, we cannot use quotes. We just write the full name of a variable:

```
y = "World"
print ( "Hello", y )
```

Output: `Hello World`

Another way is to use the **+** character instead of a comma. If we use a comma, code adds space between the values automatically, but if we use plus, it doesn't add any space, like in the following example:

```
print ( "Hello" + "World" )
```

Output: `HelloWorld`

We write the **Hello World** value to display as output, like in all preceding examples. And we have many options to do that. In the **print** function, we can use characters with percentages and letters. This method is an old-style usage in Python, but it's still supported in Python v3. For example, in the first print function, we write the value as **This apple is red**. So, output is the same as the value. Alternatively, we can divide the red value and assign it to a variable.

```
print ( "This apple is red" )
x = "red"
print ( "This apple is", x)
```

Output: `This apple is red`

We can also call a variable with print function in Python inside a string or value in quotes. To do that, we must use a special method. Since we cannot use variables directly inside quotes, we write a percentage character and the **s** letter to solve it for strings or letters as values. So, when Python sees **%s** as a special character, it understands that it is the value of the **x** variable which is **red**. The usage is also simple. We write **%s** inside quotes, and after that, we write percentage with the variable. We can also write values instead of variables. To do that, we can write the percentage with value as in the second **print** function:

```
x = "red"
print ( "This apple is %s" %x)
print ( "This apple is %s" %"red")
```

Output:

```
This apple is red
This apple is red
```

We can also write **%d** for integers. We will focus on strings, integers and other data types later in this chapter. We can call variables or integers, like in the following examples. When we call an integer, we cannot use quotes. We must write the values like variables:

```
x = 30
print ("10 plus 20 equals to %d" %x)
print ("10 plus 20 equals to %d" %30)
```

Output:

```
10 plus minus 20 equals to 30
10 plus minus 20 equals to 30
```

With Python version 3, we have a new method than percentage sign. We use a curly bracket inside the **print** function, then we write **.format** after quotes with dot and write the values inside parentheses. We can also use multiple variables to call in a string. So, each curly bracket identifies the value in the parenthesis of the format method by order. So, in the second **print** function, after **This**, curly brackets call for the **x** variable, and after **is**, curly brackets call for the **y** variable. The usage of this method is different but easier than that of the percentage sign. The result is the same.

```
x = "apple"
y = "red"
print (" This {} is red " .format (x))
print (" This {} is {} " .format (x,y))
```

Output:

```
This apple is red
This apple is red
```

Another usage of the format method is to write the **f** letter at the beginning of the quote and write the variable in quotes inside curly brackets. This usage makes it easier to write and handle an issue. The result is still the same as the other examples:

```
x = "apple"
y = "red"
print (f"This {x} is red")
print (f"This {x} is {y}")
```

Output:

```
This apple is red
This apple is red
```

In old usage, we must mention if it's string or integer as **`%s`** or **`%d`** or other data type. But with this usage, we don't need to mention the data type. Python understands it by itself. So, for integers, we directly write **`.format`** at the end of the quote or write **`f`** at the beginning of the quote:

```
x = 30
print ( "10 plus 20 equals to {}" .format(x) )
print ( f"10 plus 20 equals to {x}" )
```

Output:

```
10 plus 20 equals to 30
10 plus 20 equals to 30
```

Input ( )

The **`input`** function is also a basic Python function that we use. The **`print`** function displays the output of a value. So, data written in the code, shown to the user. The **`input`** function reads the data that is entered by the user via the keyboard and saves it to a variable. So, it's the opposite of the **`print`** function. The **`input`** function is used when a user needs to add any data to a program. This could be an IP address or password to log in to a device for network engineers.

The usage of the **`input`** function is also similar to that of print. We **`input`** function names in parentheses, and we can assign this function to a variable. Here, we assign the **`input`** function to the **`x`** variable:

```
x = input ( )
```

We can also write something that can be understandable for us. As shown in *Figure 2.1*, when we run the code, it asks for user input in the **`Run`** section in Pycharm. If we write the code in cmd or terminal, the program goes to the following line to wait for the input that you will enter.

`x = input()` indicates that a simple cursor blinks on the interactive command console when the program is executed. Whatever we enter here is assigned to **`x`**. We write **`Hello World`** and press the *Enter* button on the keyboard. So, the input value is set to the **`x`** variable because we assign an **`x`** value to the output of the **`input`** function. We get the value of the **`x`** variable, and it's time to display the **`x`** variable to see the output of the code. So, we write a **print** function to call the **x** variable. Output is **Hello World** as the value of the **x** variable:

```
x = input ( "Enter the value: " )
print ( x )
```

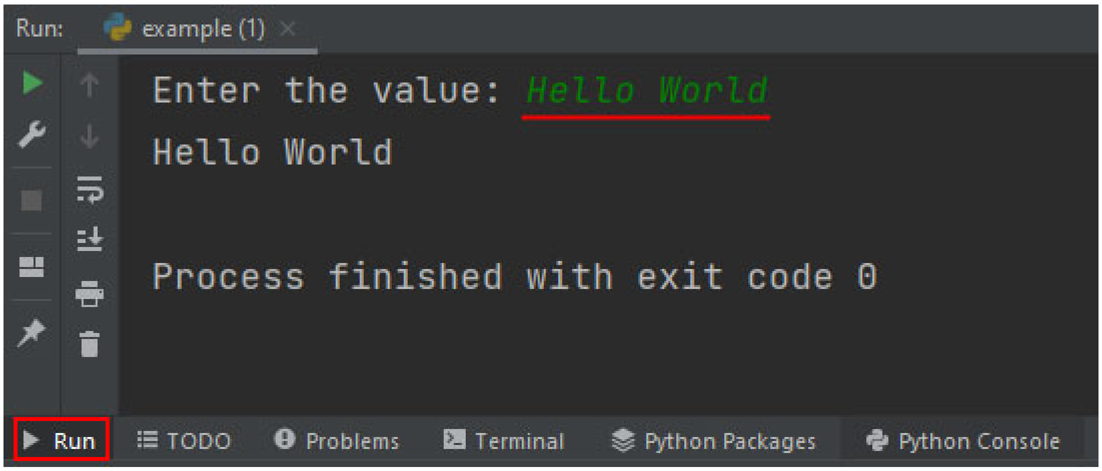


Figure 2.1: *Input Function Example*

In *Example 2.2*, we write an example to get the IP address, username and password data from users. When we run the code, the program asks for three entries for those variables input. Each time we enter data and press *Enter* on the keyboard, the program saves them to variables as IP, user and password. And finally, we can write a print function to display all these values. In the **print** function line, we use **\n** 2 times. **\n** creates a new line in Python; it's similar to the Enter key on the keyboard. So, we can see the output clearly in *Figure 2.2* with three lines of output.

Example 2.2: Getting IP address, username and password information with input function

```
ip = input ( "Enter IP Address: " )
user = input ( "Enter Username: " )
password = input ( "Enter Password: " )
print ( f"IP Address: {ip} \nUsername: {user} \nPassword:
{password}")
```

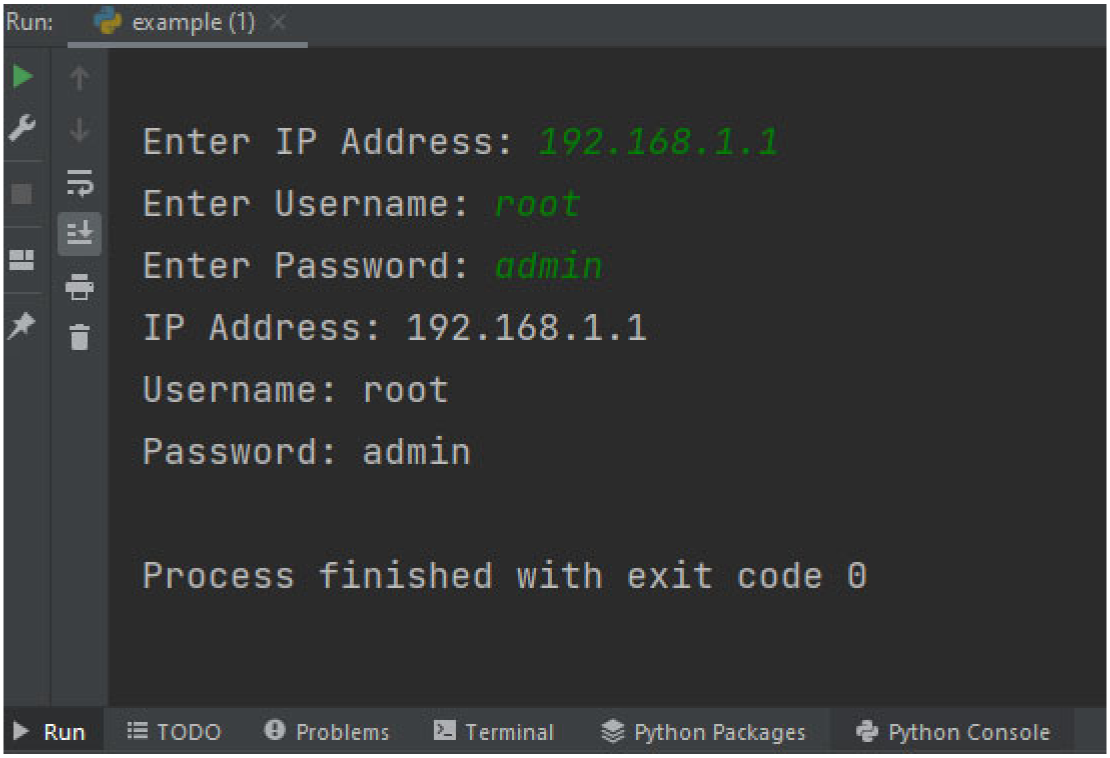


Figure 2.2: *Input Function Output of example 2.2*

Data types

In programming, one of the basic and most important sections are data types. Each value or data has its own type. All data types have different purposes and features, so we need to know each of them to use in our scripts in the correct definition.

Anything written in a single quote, like **`'hi'`**; double quotes, like **`"hi"`**; triple quotes, like **`'''hi''’`**; or **`"""hi"""`** is a string.

```
x = "5" # the value stored in x is string
x = 5 # the value stored in x is integer
y = x # here x is a letter but not a string. It's a variable
```

In Python, we create variables, and those variables store values in one of the data types in Python. It can be anything. The easiest way to find the data type of a variable is to write the **`type()`** function. We need to write a type of function with the variable or value that we want to know the data type of inside parentheses. We can display the output of type with the **`print`** function:

```
a= 10
```

```
b= "Hello World"
print ( type( a ) )
print ( type( b ) )
```

Output:

```
<class 'int'>
<class 'str'>
```

The most common data types that we use in network automation are string, integer, list, dictionary, and range. In addition to these data types, we have float, complex, tuple, set, and frozenset data types and more. These are also used in Python programming, but we rarely used them in our network automation scripts:

- **Text sequence**: String
- **Numeric types**: Integer, Float, Complex
- **Sequence types**: List, Tuple and Range
- **Mapping type**: Dictionary
- **Set types**: Set, Frozenset
- **Boolean operations**: And, Or, Not
- **Binary sequence types**: Bytes, Bytearray, Memoryview

All of these data types have special features to use in coding. We check the basics of some important ones with examples.

- **String**: It's a simple text data type. `x = "Hello World"`
- **Integer**: It's used for positive and negative integer numbers `(…, -1, -2, 0, 1, 2, …) x = 52`
- **Float**: It's used for float numbers. `x = 3.6`
- **List**: It's used to store multiple values or data in a single variable. Each item can be any kind of data type. Strings and numeric data types can only store one value inside a variable.

  ```
  x = [ "cat", "dog", "bird" ]
  ```
- **Tuple**: It's similar to a list. We will check the difference in the comparison chart later in this chapter. Instead of square brackets in the list, we use parentheses for tuples.

  ```
  x = ( "cat", "dog", "bird" )
  ```

- **Dictionary**: We can store multiple items in a dictionary as mapping data types. We have keys and values that are attached together in an item.

```
x = { "Animal" : "Bird", "type" : "Parrot", "color" : "Red" }
```

- **Set**: It has features similar to those of lists. They are created with unordered items and immutable data types.

```
x = { "cat", "dog", "bird" }
```

In *Table 2.1*, we make the comparison table for *list, tuple, set and dictionary*.

Comparison	List	Tuple	Set	Dictionary
Can change value	Mutable	Immutable	Mutable	Mutable
Usage	[items]	(items)	{ items }	{ items }
Keep duplicated values	YES	YES	NO	NO
Orderliness	Ordered	Ordered	Unordered	Unordered
Parameters	Values	Values	Values	Keys and Values

Table 2.1: *Comparison of List, Tuple, Set and Dictionary Data Types*

We can change values in a list, set and dictionary, but we cannot chage them in tuples. We use square brackets for lists, parentheses for tuples, and curly brackets for set and dictionary.

We can store duplicate or the same items in a list and tuple, but in set and dictionary, they must be unique. List and tuple are ordered, while set and dictionary are unordered. So, in lists, we can call the second or third item, but we cannot do it in dictionary. We can call according to the keys.

In list, tuple and set, we have values, and in dictionary, we have keys and values.

String and integer

String is a sequence of characters. It’s a text base data type. String variables can store characters as alphabetic characters, numbers, spaces and also special characters or signs. There are a couple of usages of the string data type.

We can use strings inside double or single quotes. Their usage is the same and output is also the same for Python. If we print the following two variables and run: the code, the output of the `x` variable and the `y` variable is same.

```
x = "Hello World"
y = 'Hello World'
```

We can use double quotes in a double qouted string, but we must add a `\` (backslash) before the double quote. It’s also same for single quotes:

```
x = "Writing codes with \"Python\" is so easy"
x = 'Writing codes with \'Python\' is so easy'
```

We can use strings inside three times single quotes or three times double quotes. If we have a multiline string, we cannot write it in single or double quotes. In this situation, we must write three times single or double quotes, and we can write multiple lines inside them. If we write the following example with only a single qoute or double quotes, the program throws an error.

```
x = """
Hello World
This is a simple Python code.
Thank you !!!
"""
```

The **integer** data type is used for positive and negative integer numbers (…, -1, -2, 0, 1, 2, …). For integers, we directly write the integer number wihout any quotes. For example, if we write the `x` variable and set a value as `10`, `x` automatically becomes an integer. But if we write the value in quotes, it becomes string.

Example 2.3: Find the data type of a variable

```
x = 10
y = "10"
print ("Data type of x = ", type (x))
print ("Data type of y = ",type (y))
```

Output:

```
Data type of x = <class 'int'>
```

```
Data type of y = <class 'str'>
```

String methods

For each data type, Python has different methods or functions to modify or manipulate the data. In the string data type, we have too many methods over 40. We can use them to manipulate the strings. In this chapter, we will focus on the most used methods that can help us with writing automation scripts:

- **`len (string)`**: The **`len`** method is used to find the length of a script. It counts the quantity of all characters. It could be alphabetical characters, digits, space or any special character. To use this method, we need to write **`len`** and then, inside parentheses, we need to write a string or variable that is also a string. As in the following example, **`x`** is a string variable. We understand that the value is inside double quotes. When we write **`len(x)`**, it returns the value of the **`x`** variable, which is **`Hello World`**. This string has 10 letters and **`1`** space, so the length of this string is **`11`**. If we print **`len(x)`**, we can display the output of the returned value as **`11`**, or we can write a string directly inside the **len** method. Both usages have the same result:

  ```
  x = "Hello World"
  print ( len ( x ) )
  print ( len ( "Hello World" ) )
  ```

  Output:

  ```
  11
  11
  ```

- **`string.upper ()`**: Upper methods are used to replace all capital letters. To use this method, we first write the string or the string variable and after the dot, we write the upper method in parentheses. If we use the **`x`** variable as **`Hello World`**, only **`H`** and **`W`** are capital letters. This method changes all the letters to capital letters, but the **`x`** variable is still the same. There is no change in it because all these methods return new values. They don’t change anything in the original variable or string. In the following example, we assign the method output to the **`y`** variable. So, if we display **`x`** and **`y`** with the **`print`** function, we can see that the value of the **`x`** variable is still unchanged, but the value of the **`y`** variable has a value that the upper method returns.

```
x = "Hello World"
y = x.upper()
print ( x )
print ( y )
```

Output:

```
Hello World
HELLO WORLD
```

- `string.lower ()`: This has the exact opposite function of the upper method. It is used to change all letters from capital to small. In the **x** variable, we have **H** and **W** in capital letters, so the new value must be all small letters. If we use the same code as earlier, only replace upper with the lower method, the value of **y** is **hello world**.

```
x = "Hello World"
y = x.lower()
print ( x )
print ( y )
```

Output:

```
Hello World
hello world
```

- `string.strip ()`: The usage of the **strip** method is the same as that of the upper and lower methods. We write string or string variable and then dot and strip in parentheses. This method deletes spaces at the beginning and ending of a string and returns the new value. In the following example, we have spaces at the beginning and end of **x** variable's value. We assign the returned value to **y**, so if we print **y**, we can see that there is no space at the beginning and at the end, but we still have space between words, such as "**Hello**" and "**World**".

```
x = " Hello World "
y = x.strip()
print ( x )
print ( y )
```

Output:

```
Hello World
Hello World
```

- `string.replace ()`: The `replace()` function replaces the first argument in the string with the second argument. To use it, we need to write a string or string variable and replace the method with parentheses, and we need to divide both with a dot. Finally, we need to write old values and new values inside parentheses by dividing them with commas.

In *Example 2.4*, we replace the first argument, **`He`**, with the second argument, **`Te`**, in the **`x`** variable and assign the output to the **`y`** variable. The program checks whether the **`H`** and **`e`** characters are inside the target string, which is **`x`**. However, these letters must be there together with the order.

Example 2.4: Replace characters in a string with the replace method

```
x = "Hello World"
y = x.replace("He", "Te") #'He' is replaced with 'Te'
print ( y )
```

Output:

```
Tello World
```

If we write the following example and try to find "**`Wl`**" together, we will have no match in the **`x`** variable even though we have "**`W`**" and "**`l`**" in different places. So the **`replace`** method cannot change anything on the **`x`** variable, and the result of **`y`** is equal to **`x`**.

```
x = "Hello World"
y = x.replace("Wl", "Te")
print ( y )
```

Output:

```
Hello World
```

- **`string.split ()`**: The **`split`** method is used to split the string into a specific character. After that, it returns an output as a list data type. So, the input is a string, but the output is a list. We again have the **`x`** variable. The target character must be written inside parentheses after **`split`**. This could be a string or a string variable.

  In the following example, we write the **`o`** character, but this could be a word or multiple characters. So, each time the program matches the **`o`** character in the source value, it splits from there, and each part is written with different items in the list. And the **`o`** character is removed. In the following example, we have two of the **`o`** characters. The program matches, and it removes the **`o`** character and divides it into the items in a list. The output is `['Hell', ' w', 'rld']`, which is a list of three items:

```
x = "Hello world"
y = x.split("o")
print(y)
```

Output:

```
['Hell', ' w', 'rld']
```

List

One of the most important sequence data types for network automation is a list data type. We will use it in almost every automation script, so it's quite important to understand the logic of this data type. Some basic features of the list data type are mentioned here:

- Lists are sequences of arbitrary objects. Each object is an item.
- Items are ordered and changeable in a list.
- The list allows duplicate items, so there could be multiple same items in a list.
- All the items in the list are divided by commas.
- Each item has a unique index number in a list. Index numbers start from 0.
- Lists are created in square brackets, and all items are added inside these square brackets.
- We can also create lists inside a list.

```
x = [ item1, item2, item3, item4, … ]
```

In the previous example, we have the `x` variable as a list. There are items divided with commas, and all items are inside square brackets:

```
x = ["Lion", 42, "Panda", "42", "snake", [ "1", " 2 ", " 3 " ]
]
```

Our first item is `Lion` as a string. Then, we have `42`, which is an integer. Then we have `panda` as a string, and then we again have `42`, this time as a string. Since we use it inside double quotes, we have characters as `4` and `2`. Then, there is a string as `snake`. Finally, there is another list inside of this list.

We print **x** square brackets and **0**. Code gets the first item in the list. The item index starts from **0**. Our first item is **Lion**, so the output is **Lion**. In the second example, we print the third item as **x[2]**, which is **Panda**. In the third example, we want to print the second item from right to left or from the end of the list. So, if we write minus 1, the code gets the last item from the list. It receives the item in reverse ordering, from right to left. The **minus 2** is a **snake** item, so we can search for an item in the list from the beginning or end.

```
print ( x [ 0 ] ) Output: Lion
print ( x [ 2 ] ) Output: Panda
print ( x [ -2 ] ) Output: snake
```

This time, we check the data type of items in a list. For that, we use the **type** function. Here, we print the data type of the second item in the **x** list as **x[1]**. It's an integer. If we check the fourth item with **x[3]**, it's a string. If we check the last item in the list, we can write **x[-1]** or **x[5]**, and the result is the same. It's a list.

```
print ( type ( x [ 1 ] ) ) Output: <class 'int'>
print ( type ( x [ 3 ] ) ) Output: <class 'str'>
print ( type ( x [ -1 ] ) ) Output: <class 'list'>
```

Now, we **print x[0:3]**. It means that print items from first to fourth item (but the fourth item is not included), so from the first item to the third. The output is **["Lion", "42" and "Panda"]**:

```
print ( x [ 0 : 3 ] ) Output: ['Lion', 42, 'Panda']
```

We can change all items of a list in reverse ordering. To do that, we write *double colon and minus 1* as **x[::-1]** inside square brackets:

```
print ( x [ ::-1 ] ) Output: [['1', ' 2 ', ' 3 '], 'snake',
'42', 'Panda', 42, 'Lion']
```

- **Replacing items**: Suppose we have a list as **x** variable with three items, which are "**elephant, turtle, hamster**". We can change or replace any of the items on the list. To do that, we write list variables with square brackets. Inside square brackets, we write the index number of the item that we want to change. In the following example, we write **1**. So it's the second item in the list: **turtle**. The second item is replaced with **fish** as a string, so the new value of the **x** variable is **["elephant", "fish" and "hamster"].**

```
x= ["elephant", "turtle", "hamster"]
x [ 1 ] = "fish"
```

Output: `["elephant", "fish", "hamster"]`

- **Check item existence**: We can also check whether or not an item exists in a list. We can check its existence with the `if` condition. In this code, we said that if there is a `hamster` item inside the `x` variable, print `Yes`. Because the `hamster` item is included in the `x` variable, the output is `Yes`. If we write `fish` instead of `hamster`, there is no output because the condition does not match. In the following code, after the `if` condition, there is a space in the following line. It's an indentation in Python. The following line of the statements or functions starts with spaces in Python. If we don't add space, the code understands that we exited from the statement. We will check the indentation mechanism in this chapter's *Statements and conditions* section.

```
x= ["elephant", "turtle", "hamster"]
if "hamster" in x:
print("Yes")
```

Output: `Yes`

- **Create an empty list**: We can create empty lists. There are two options to do that, and both have the same result. We can write a list with empty parentheses and assign it to a variable, like the `x` variable in the following example. Alternatively, we can write empty square brackets and assign it to a variable. If we print the variable, the output is an empty list.

```
x = list( )
y = [ ]
print ( x )
print ( y )
```

Output:

```
[]
[]
```

- **Merge lists together**: We can merge or join different lists. We create `x` and `y` variables as lists. To merge two list variables together, we just add them with a plus sign. In the following example, we create a new variable **z** for the calculation. If we print **z**, the result is a list with two items that come from **x** and **y**.

```
x = ["Lion" ]
y = ["Elephant"]
z = x + y
print(z)
```

Output: `['Lion', 'Elephant']`

List methods

There are many methods in list data type to manipulate it. We will focus on the most commonly used ones that we need in network automation:

- **list.append (item)**: Append method is used to add an item at the end of the list. To use it, firstly, we write the list variable, and after the dot, we write the **append** method. Then, we write the new item inside parentheses that we want to add at the end of the list. In the following example, we have three items: **fruit, vegetable, water**. We want to add **coffee** string to the end of the list, so we write **x.append("coffee")** string inside parentheses. We add a new item with this method. We will use the **append** method often for our scripts in loops to add new items at the end of the lists later in this chapter.

  Example 2.5: Add new items end of the list with the append method

  ```
  x= ["fruit", "vegetable", "water"]
  x.append ( "coffee" )
  print(x)
  ```

  Output: `['fruit', 'vegetable', 'water', 'coffee']`

- **list.insert (index, item)**: The **insert** method inserts an item at a particular index. In the following example, **fruit** is currently at index 0, and **vegetable** is at index 1. When we insert **tea** as index 1, all items after index 0 are shifted. So, in the output, **tea**” is at index 1, and **vegetable** is at index 2.

  ```
  x= ["fruit", "vegetable", "water"]
  x.insert ( 1, "tea" )
  print(x)
  ```

  Output: `['fruit', 'tea', 'vegetable', 'water']`

- `list.remove ( item )`: Remove method is used to remove the first match item in the list. In the following example, we have four items: **fruit**, **coffee**, **water**, **fruit**. We write the **x.remove** method and **fruit** as the target string match. In the list, the **fruit** item is used twice, so there are two matches of this string. However, the **remove** method removes only the first match item in the list; so, the output of **x** variable is **coffee, water, fruit**.

  ```
  x = ["fruit", "coffee", "water", "fruit"]
  x.remove("fruit")
  print(x)
  ```

  Output: `['coffee', 'water', 'fruit']`

- `list.pop ( index )`: This method is used to remove a specific index in the list. Instead of the remove method, the **pop** method uses the index number to delete an item. In the following example, we write **x.pop** with **0** inside parentheses. It means remove item **0**, which is the first item in the list, so **cat** item is removed from the list.

  ```
  x = ["cat", "dog", "monkey"]
  x.pop (0)
  print(x)
  ```

  Output: `['dog', 'monkey']`

  If we don't write any index inside parentheses, the code deletes the last item in the list. In this example, it's **monkey**.

  ```
  x = ["cat", "dog", "monkey"]
  x.pop ()
  print(x)
  ```

  Output: `['cat', 'dog']`

- `del list [ index : ]`: The **del** method is used to delete the specific items in the list. Unlike the **pop** method, **del** can delete a bunch of items. It deletes the specific item. If we add a colon after item, it deletes all the items starting from the specific index. In the following example, we write **1:** inside square brackets. The code deletes all items from the second item to the last one. So, the output of this code is only the first item, which is the **dog** string. We can also use the **del** method with writing the index without a colon; it deletes the specific item in the list, like the **pop** method.

  ```
  x = ["dog", "monkey", "cat"]
  ```

```
del x [ 1 : ]
print(x)
```

Output: `['dog']`

- `list.clear ()`: The `clear` method is used to delete all the items in a list so that the new value of the list is an empty list. In the following example, code deletes all items inside the `x` variable, so the result of `print(x)` is an empty list.

```
x = ["dog", "monkey", "cat"]
x.clear ( )
print(x)
```

Output: `[ ]`

- `list.copy ()`: The `copy` method is used to copy a list into a new list. To copy a list to another, we write `x.copy()`. In the example, we copy the `x` variable to the `y` variable. So the output of `y` is `dog, monkey, cat` as a list.

```
x= [ "dog", "monkey", "cat" ]
y= x.copy ( )
print ( y )
```

Output: `[ "dog", "monkey", "cat" ]`

We cannot just write the equal sign to copy a list to another list, like `a = b`. If we write this, it assigns `a` to `b`. If any changes happen in `a` or `b`, it also changes in the other list, so it's not an independent copy. In the following example, if we delete item-1 in the `x` variable by the `pop` method, the `y` variable is also changed, as shown in the output. So, to copy a list as independent, the `copy` method must be used.

```
x= [ "dog", "monkey", "cat" ]
y= x
x.pop(1)
print ( x )
print ( y )
```

Output:

```
['dog', 'cat']
['dog', 'cat']
```

- `sum ( list )`: The `sum` method is used to calculate the sum of items in a list. The important thing is that all the items must be numeric data type like integer or float. Otherwise, the program throws an error. In the following example, we have a list including integers and float items. We use the **sum** method with variable name as **x** inside parentheses. The sum of the **x** variable is **50.1**.

```
x= [ 4, 5, 7, 9, 10.1, 15 ]
y = sum ( x )
print(y)
```

Output: 50.1

- **len (list)**: The **len** method is used to count the item quantity in a list. It's similar to the **len** method in string. In the following example, we have three items, so we write **len (x)**. The result is **3**.

  ```
  x= ["cat", "monkey", "elephant"]
  y= len ( x )
  print (y)
  ```

  Output: 3

- **list.count (item)**: The **count** method is used to count the items inside a list. In the following example, we write **x.count ("bird")** to count the **bird** string item in the **x** variable. The result is **2** because we have duplicated items added twice in the list. We must write the exact same item in the **count** method. For example, this method cannot find anything if we write **bir** instead of **bird**. If we don't write exact characters in the item, the count result is **0**.

  ```
  x= ["bird", "horse", "elephant","bird", "monkey"]
  y = x.count ("bird")
  print(y)
  ```

  Output: 2

Dictionary

Another important data type for network automation is a dictionary. We can store multiple items in a dictionary as mapping data types. We have keys and values that are attached to an item.

- Dictionary is an unordered data type.
- Dictionary is a changeable and indexed data type.
- Dictionary has items, and each item has keys and values.

- Dictionary items are written inside curly brackets.
- Each key is separated from its value by a colon, and items are separated by commas.
- Keys are unique within a dictionary, but values can be duplicated. So, we cannot create the same key in a dictionary, but we can create the same value.
- Values can be of any data type, but keys must be immutable data like string, number or tuple.

```
{ key1 : value1, key2 : value2, key3 : value4 }
```

In the following example, we can see an **x** variable that is a dictionary. We can understand that all data is inside curly brackets, and each item has two sets of data, as keys and values, which are divided by a colon. From the following example, **animal** and **color** are keys and must be unique. **Lion** and **Yellow** are values.

```
x = { "animal" : "Lion", "color" : "Yellow" }
```

To find any value in a dictionary we can call the key of the value. In the following example, after the **x** variable, we write the key **animal** inside square brackets. This means finding the value that belongs to the **animal** as a key. So, the result is **Lion**.

```
print(x ["animal"] )
```

Output: `Lion`

We can also change values in a dictionary. Like in the list data type, we write the key inside square brackets and equal to the new value. So, the value of the **color** key is replaced with **White**.

```
x ["color"] = "White"
print (x)
```

Output: `{ "animal" : "Lion", "color" : "White"}`

To change keys under a dictionary, we don't have a one-line solution or method, but we have a trick to solve it. If we want to replace the **color** key with a key as **type**, we can write the variable with the new key and equal it to the old key inside the square brackets. However, this code creates another item with the same value. Here, the new key is **type** and the value is still **Yellow**. And if we use the **del** method in the dictionary for the old key as **color**, we can reach what we want as a result. As you can see from the following code, there are two **print** functions. In the first print function, we can see that the key **type** is added to the dictionary. And in the second, we can see that the key **color** is removed from the dictionary.

```
x ["type"] = x ["color"]
print (x)
del x ["color"]
print (x)
```

Output:

```
{'animal': 'Lion', 'color': 'Yellow', 'type': 'Yellow'}
{'animal': 'Lion', 'type': 'Yellow'}
```

We can also add items in a dictionary. We just write a unique key and assign it to a value. In the following example, we create an item with a key as an **age**, and value as an integer **10**. So, the program adds a new item with a key and its value. Dictionaries are unordered. So, we cannot say that the item was added at the end. There is no beginning and end of dictionary items.

```
x ["age"] = 10
print (x)
```

Output: `{"animal":"Lion", "color":"Yellow", "age" : 10 }`

Dictionary methods

Similar to list data type, in the dictionary, we have a different method for every purpose:

- **dict.copy ( )**: The **copy** method is used to copy a dictionary to another dictionary:

  ```
  x = {"animal" : "Lion", "color" : "Yellow", "age" : 7}
  y = x.copy ( )
  print(y)
  ```

  Output: `{'animal': 'Lion', 'color': 'Yellow', 'age': 7}`

- **del dict [key]**: The **del** method is used to delete an item. In the following example, we want to delete item with key as **color**. So, we use the **del** method and write the target key inside square brackets in the **x** variable. In the output, the item with the **color** key and the **Yellow** value is deleted:

  ```
  x = {"animal" : "Lion", "color" : "Yellow", "age" : 7}
  ```

```
del x [ "color" ]
print(x)
```

Output: `{'animal': 'Lion', 'age': 7}`

- `dict.pop ( key )`: The `pop` method is used to delete items with specified keys. It has the same function as the `del` method.

```
x = {"animal" : "Lion", "color" : "Yellow", "age" : 7}
x.pop ( "age" )
print(x)
```

Output: `{'animal': 'Lion', 'color': 'Yellow'}`

- `dict.clear( )`: The `clear` method is used to clear or empty a dictionary. In the following example, we clear the `x` variable with `x.clear()`. The output is an empty dictionary.

Example 2.6: Delete all items inside a dictionary with clear method:

```
x = {"animal" : "Lion", "color" : "Yellow", "age" : 7}
x.clear ( )
print(x)
```

Output: `{ }`

Dictionaries in "for" loop: We can call dictionary items with `for` loops. We will learn about `for` loops later in the chapter. Now let's check the items in the dictionary `for` loop statements. For example, we have 5 values from 1 to 5. To loop these values, the output should be from 1, 2, 3, 4, and finally, 5. So each value is looped from beginning to end.

In the following example, we have a dictionary with three items. We can print keys inside the dictionary with the `for` loop. The code finds keys in the `x` variable and prints them. So, from the beginning, we have three prints because we have three items, which are `animals`, `color`, and `age`.

```
for i in x:
  print (i)
```

Output:

```
animal
color
age
```

In the second example, we want to collect only values. To do that, we must write `x.values ()` to call values. In the output, only values are shown:

```
for i in x.values ( ) :
  print (i)
```

Output:

```
Lion
Yellow
7
```

To find keys and values together, we use **`x.items()`**. And in the loop, we have two items as **`a`** for keys and **`b`** for values. Results are shown as items. Each item is listed with its key and value in order:

```
for a, b in x.items ( ) :
  print (a, b)
```

Output:

```
animal Lion
color Yellow
age 7
```

Statements and conditions

One of the key topics in programming is conditions and statements. We will focus on **`if`**, **`for`**, **`while`**, **`break`**, **`continue`**, **`range`** function, **`for…else`**, nested loops and **`try…except`** statements in this book. All of these will be necessary for our scripts in network automation.

If condition

There are some characters to be used in an if condition. *Table 2.2* lists all the options for if statement equality:

Condition	Sign	Usage
Equal to	= =	a = = b
Not equal to	! =	a ! = b
Greater than	>	a > b
Greater than or equal to	> =	a > = b
Less than	<	a < b
Less than or equal to	< =	a < = b

Table 2.2: If condition sign and usage

To write an if condition, we must write **`if`** followed by the condition. After that, the line must be finished with a colon. If the **`if`** condition is matched, we continue with the body of the condition, which is the **`statement`** in the next line. In the following usage, there are some spaces before the statement. It means we are inside an **`if`** condition. If we write without space, the code gives an indentation error. So, indentation is very important in Python. For **`if`** or any other statement, we must carefully write the code with indentations if it's necessary. For general use, it's simpler to create space with a tab. So, the code is clearer to understand:

```
If Condition :
  Statement
```

In the following example, we create two integer variables: **`a`** is **`33`** and **`b`** is **`200`**. We write that if **`b`** is greater than **`a`**, print **`b is greater than a`**. We write **`if`**, then the condition with a greater sign. After that, we finish the line with a colon. In the next line, we enter the statement with indentation. So, when we run the code, because **`b`** is greater than **`a`**, the condition is matched, and the statement starts. The statement is to print **`b is greater than a`**. It prints this as an output. If the **`b greater than a`** condition is not matched, there is no output because the statement doesn't start as it is passed.

```
a = 33
b = 200
if b > a :
  print("b is greater than a")
```

Output: **`b is greater than a`**

When the **`if`** condition is matched, it continues with its statement, but when the **`if`** condition is not matched, we need to check another condition. In Python, we have **`elif`** and **`else`** statements inside the **`if`** condition. **`elif`** is similar to **`else if`** in other programming languages. We can write multiple **`elif`** statements. It means that if the upper condition is not matched, the **`elif`** condition will be checked. If it is not matched again, it continues with the next condition.

The usage of **`if`** and **`elif`** is the same. In the first condition, we write **`if`**, and for later conditions related to the **`if`** condition, we write **`elif`**. At the end of the **`if`** condition, we can use the **`else`** condition. This means that if all upper conditions are not matched, that statement will continue before exiting the if condition. **`elif`** and **`else`** are optional conditions in the **`if`** statement. In the following structure, if any of the conditions is matched, its statement runs, and after that, it exits from the loop. It doesn't continue to check whether other conditions are matched.

```
if condition :
  Statement
elif condition :
  Statement
  …
else :
  Statement
```

In *Example 2.7*, there are two integer variables: **`a`** is **`200`** and **`b`** is **`33`**. If **`b`** is greater than **`a`**, it prints **`b is greater than a`**, else, if **`a`** is equal to **`b`**, it prints **`a and b are equal`**, else, it prints **`a is greater than b`**. When we checked the values of **`a`** and **`b`**, **`a`** is greater than **`b`** in the example. The first condition fails, so the program checks the second condition, which also fails; the program continues until the **`else`** condition. There is no condition in else. So, in any way, it performs the action for **`else`** condition. Instead of **`else`**, we can write the **`elif a > b :`** condition and write the **`print`** function inside it. Because **`a`** is greater than **`b`**, this condition would match.

Example 2.7: Compare 2 integers with if condition

```
a = 200
b = 33
if b > a :
  print("b is greater than a")
elif a == b :
  print("a and b are equal")
else:
  print("a is greater than b")
```

Output: **`a is greater than b`**

In this example, we have two scripts. The content of these scripts is similar; only the **`if`** conditions are different. In the first script, we use the **`if`** condition each time. So, there are four different **`if`** conditions that are all independent. If we enter the value of the **`x`** variable as **`-1`**, the first and fourth conditions will match. These conditions are connected, so we have two different results in the output:

```
x = -1
if x < 10:
  print("x is less than 10")
if x == 10:
  print("x is equal to 10")
if x > 10:
  print("x is greater than 10")
if x < 0:
  print("x is less than 0")
```

Output:

```
x is less than 10
x is less than 0
```

But in the second code, we use the `if` and `elif` conditions. So, there is only one `if` condition in the following code, and each `elif` statement is dependent on the upper condition. When we enter the `x` variable as `-1` again, this time only one match happens. The first condition is matched. So, after the `print` function runs, the code exits from the `if` condition. So, the output is different from the previous example; we can see the difference in using the `if` condition.

```
x = -1
if x < 10:
  print("x is less than 10")
elif x == 10:
  print("x is equal to 10")
elif x > 10:
  print("x is greater than 10")
elif x < 0:
  print("x is less than 0")
```

Output: `x is less than 10`

For statement

We use `for` loops in repeatable actions. For example, suppose we want to call all the items in a list, and we have 100 items inside a list. We need to write the same code 100 times. But with the `for` loop, the code needs to be written only once and can be put into a `for` loop. It makes the program simpler with less code. We use `for` loops in many scripts for automation.

We always use **for** loops to connect many devices and several different commands. It is a very important section for automation.

The **for** loop is a sequence statement. Each item in the loop runs in order, and the loop continues until it reaches the last item. It returns with the next value till the end. After the last item is proceeded, the loop is finished, and control exits from the loop. In the next line, we enter the statements. This is the body of a **for** loop:

```
for Variable in Iterable :
  Statement(s)
```

There are three things to check in the **for** loop: variable, iterable, and statement.

- **Iterable** is a collection of objects like a list.
- **Variable** is used to get the items from the iterable. In each loop, it gets the next item until it gets all items in iterable.
- **Statement** is the body of the loop. It is written inside the **for** loop, so there is an indentation for the statement. It is executed for each item inside of the iterable.

In the following example, we have three values in the animals list. The **for** loop says that in the animals list, check the **x** variable and print in the body of **for** loop as a statement. So, from the beginning, the for function prints **elephant** and finishes the first round; in the next iteration, it gets **monkey** as a new variable and prints **monkey**; in the last iteration, it gets **cat** as a new variable and prints **cat** in the statement. After that, the loop is finished, and control exits from the loop because there is no more values to check in the animals list.

If we have hundreds of items in a loop, the code checks all of them one by one. So, the **for** loop is checked from the first item to the last, with the **x** variable. In each iteration or loop, the next item in the list is assigned to the **x** variable until it reaches the last item in the iteration or list in the following example. As a result, in the first iteration, **x** is **elephant**; in the second iteration, **x** is **monkey**; and in the next iteration, **x** is **cat**. Statement is **print(x)**, so it prints those three values as output.

```
animals= ["elephant", "monkey", "cat"]
for x in animals :
  print(x)
```

Output:

```
elephant
monkey
cat
```

The following example shows the sum of all items in the **`numbers`** list. Instead of adding each item one-by-one, we can use the **`for`** statement:

```
numbers = [6, 5, 3, 8, 4, 2]
sum = 0
for x in numbers:
  sum = sum + x
print(f"The sum is {sum}")
```

Output: **`The sum is 28`**

When we execute the script, the **`sum`** variable changes in each iteration of the **`for`** statement:

```
sum=0 # Initially sum is 0 as an integer
1st Iteration:
x=6
sum=0+6 #sum is 6

2nd Iteration:
x=5
sum=6+5 #sum is 11

3rd Iteration:
x=3
sum=11+3 #sum is 14

4th Iteration:
x=8
sum=14+8 #sum is 22

5th Iteration:
x=4
sum=22+4 #sum is 26

6th Iteration:
x=2
sum=26+2 #sum is 28
```

While statement

Another statement is the `while` loop. The code finishes or exits from the loop if the while condition is not matched. But if the condition is matched, it continues with the body of the `while` loop. In each iteration or loop, it returns to the beginning to check the `while` condition. It checks until the condition is not matched. The program creates a loop with this statement.

The usage of the `while` loop is similar to that of the `if` condition. We have a `while` loop and test expression, and the line is finished with a colon. In the next line, we write the statements of the loop with indentation:

```
while Test_expression :
  Body_of_while
```

In the following example, we create an integer `x` that equals `0`. In the `while` condition, we write that `x` is less than `6` and finished the line with a colon. It means that the `while` loop will continue until the `x less than 6` condition is false or not matched.

In the body of while, we print out the `x` variable, and then add 1 to `x`. There are two options to write that. We can write `x = x + 1` or `x += 1`. So, we add 1 to the `x` variable in each iteration:

```
x = 0
while x < 6 :
  print(x)
  x += 1
```

Output:

```
0
1
2
3
4
5
```

When we run the code, `x` assigns `0` an integer. The `While` loop checks whether `x` is smaller than `6`. If it's true or matches the condition, it continues with the body of the while. In this case, it's true.

In the body of the `while` loop, the code prints `x`, which is `0`. In the output, we can see that first output is `0`. Then, it adds `1` to `x`. So, the value of `x` is `1` now. The body of `while` is finished for the first loop.

Then, it checks the condition again. **x** is **1**, so it's still lower than **6**. Now it prints **x** as **1** and then adds **1** to **x**. This continues until **x** reaches **6**. When it reaches **6**, the condition is not true or does not match anymore, so the code exits from the loop.

We need to be careful while writing a **while** loop in the script. We can mistakenly write a **while** loop for infinitive times, and it may never end. If we delete the last line in the body of while, which is adding **1** to **x**, the value of the **x** variable is **0** every time. So, the **while** loop never ends because the condition always matches, i.e., **x** is less than **6**.

```
x = 0
while x < 6 :
  print(x)
```

Output:

```
0
0
...
```

We can write the **print** function out of the **while** loop without indentation. This time, the code writes the final value of **x**, which is **5**. After the code exits from the loop, the **print** function can be executed like in the following example. And the value of **x** is **5** when the **while** loop is finished:

```
x = 0
while x < 6 :
  x += 1
print(x)
```

Output:

```
6
```

We can also write the **print** function after adding **1** to **x**. So, in the body of a while, the code adds **1** to **x** and then prints **x**. The result starts from **1** to **6** because we change the order in the body of the **while** loop.

```
x = 0
while x < 6 :
  x += 1
  print(x)
```

Output:

```
1
```

```
2
3
4
5
6
```

Break and continue statement

Break statement: The **`break`** statement is used to exit the loop. For example, we enter the loop, code checks the condition or the statement of the loop, and if it's false or not matched, it exits the loop. But if it's true or matched, we have another option to exit from the loop. In this situation, we can use the **`break`** statement. So, in any part of the loop, if the code executes the **`break`** statement, the code does not continue to the loop and exits immediately.

We mostly use **`break`** statements with **`for`** and **`while`** loops. We put a break statement inside the body of the loop, and we use them for a purpose. For example, when we want the code to exit from the loop if something matches our expectation inside the loop. We can use the **`if`** condition, i.e., if the condition is matched, execute the **`break`** statement and exit from the loop.

`for variable in iterable :`
`body_of_for`
`if condition :`
`break`
	`while Test_expression :`
	`body_of_while`
	`if condition :`
	`break`

Table 2.3: *Break statement usage in for and while loop*

In the following example, we have an animals list with three items: **`lion`**, **`dog`**, **`monkey`**. We create a **`for`** loop and print each item with an **`x`** variable. If we finished the line here, the result is **`lion`**, **`dog`**, **`monkey`**. But we want to finish or exit the loop when the code matches a value as **`dog`** in the list, we write the **`if`** statement that if **`x`** variable equals **`dog`**, break the loop.

In the second iteration, where the value of the **`x`** variable is **`dog`**, the code exits from the loop. Because the if condition is matched, which is if **`x`** equals to **`dog`** string, the condition is matched, and the **`break`** command is executed.

```
animals = ["lion", "dog", "monkey"]
```

```
for x in animals:
  print(x)
  if x == "dog":
   break
```

Output:

```
lion
dog
```

If we write **`if x equals to bird`**, then the break, the code doesn't match the **`if`** statement, so the **`for`** loop continues without breaking. The result is **`lion`**, **`dog`**, **`monkey`** because there is no item named **`bird`** in the animals list.

Continue statement: The **`continue`** statement is used to skip the rest of the code inside a loop for only the current iteration. Loop does not terminate like a **`break`** statement but continues with the next iteration. So, with the **`continue`** statement, we can stop the current iteration of the loop and continue with the next iteration. The usage of the **`continue`** statement is the same as that of the **`break`** statement. The **`break`** statement exits from the loop, but the **`continue`** statement exits only from the current iterable loop.

`for variable in iterable :`
` body_of_for`
` if condition :`
`  continue`
	`while Test_expression :`
	` body_of_while`
	` if condition :`
	`  continue`

Table 2.4: *Continue statement usage in for and while loops*

In the following example, we use the **`continue`** statement instead of the **`break`** statement. With using a **break** statement, the loop finishes when the **dog** item is matched. But we use the continue statement. It only passes when the **`if`** condition is matched with **`dog`**. In the **`for`** loop, **`x`** assigns the first item in the animals list as **`lion`**. In the body of the loop, it checks the **`if`** statement, and **`x`** is **`lion`**. So, it passes the **`if`** condition and prints the **`x`** function. So, the code prints **`lion`**.

Then, **`x`** gets the **`dog`** item. It checks the **`if`** condition, which matches. So, it checks the body of the **`if`** condition, which has a **`continue`** statement. So, this section of the loop is finished, not continue to **`print`** function. Then, **`x`** gets the third item, which is **`monkey`**. It does not match the condition of the **`if`** statement, so it prints **`monkey`** like **`lion`**. As a result, we have the output **`lion`** and **`monkey`**:

```
animals = ["lion", "dog", "monkey"]
for x in animals :
  if x == "dog" :
    continue
  print(x)
```

Output:

```
lion
monkey
```

Finally, the **break** statement finishes all the loops and exits, but the **continue** statement only finishes the current iteration and continues with the next iteration.

Range statement

We often use the **range** function in the **for** loop in network automation to loop integers with specific numbers. It returns a sequence of numbers. It starts from **0**, increments by **1** by default, and ends at a specific number. There are three options to use the range function.

We can write **range** and write the **stop** value **range (stop)** as an integer inside parentheses. In the following example, we enter **5** for the **range** value. So, the **for** statement is executed from the first item of the **range** function, which is **0** and continues until it reaches the first item, which is **4**. As a result, the output is **0**, **1**, **2**, **3** and **4**.

```
for x in range(5):
  print(x)
```

Output:

```
0
1
2
3
4
```

We can write **range** with **start** and **stop** values **range (start, stop)** as integers inside parentheses. The values are divided by a comma. In the following example, we enter **2** as a **start** value and **5** as a **stop** value. So, the output is **2**, **3**, and **4**.

```
for x in range(2, 5):
```

```
  print(x)
```

Output:

```
2
3
4
```

We can write **`range`** with **`start`**, **`stop`** and **`step`** values **`range (start, stop, step)`** as integers inside parentheses. The **`range`** statement starts from the **`start`** value to the **`stop`** value, incrementing by **`1`** by default. But we can modify the incrementing value with the **`step`** parameter. In the following example, it starts with **`2`** and finishes at **`10`**, incrementing by **`3`**. So, the code gets **`2`**, **`5`** and **`8`**.

```
for x in range(2, 10, 3):
  print(x)
```

Output:

```
2
5
8
```

For else statement and nested loops

The for…else statement: There is an option to use the **`else`** statement in **`for`** loop. Normally, the **`else`** statement is used in **`if`** conditions to state that if all conditions are not matched, the loop should continue with the body of the **`else`** statement. But in the **`for`** loop, the **`else`** statement is used when the **`for`** loop finishes. After the last item in iterable is used in the loop, the code passes to the **`else`** statement. It's an optional feature in the Python language:

```
for Variable in Iterable :
  Statement
else:
  Statement
```

In the following example, we have a **`for`** loop with the range as **`3`**. In the body of the loop, we print the **`x`** variable. Then, we have the **`else`** statement to print **`Finally finished`**. After the loop is finished, the code prints **`x`** as **`2`**, continues with the **`else`** statement, and prints "**`finally finished`**".

```
for x in range(3) :
```

```
  print(x)
else:
  print("Finally finished!")
```

Output:

```
0
1
2
Finally finished!
```

Nested loops: Nested loop is a loop inside another loop. The first `for` loop is called the **outer loop**, and the second `for` loop is called the **inner loop**. We just use a `for` loop inside another `for` loop. From the first line, `for` loop, or outer loop, we assign an item to `variable-1` and continue with the body of the first `for` loop. The body of the first loop has another `for` loop, which is the second for loop or inner loop.

In the inner loop, it finds all items in `iterable-2` and continues with the inner loop statement. After the inner loop finishes, the code continues to choose the next item in the outer loop. Then, it again checks for all loop statements in the inner loop.

```
for variable-1 in iterable-1 : #Outer Loop
  for variable-2 in iterable-2 : #Inner Loop
   Statement(s)
```

In *Example 2.8*, we have two lists: `types` and `tools`. Both of them have three items. We write the first loop or the outer loop. It returns all items in the `types` list. In the body of the outer loop, we have another loop, which is the inner loop. The inner loop returns all items in the `tools` list. And the statement prints both variables in inner and outer loops as `x` and `y`. If we run this code, it acts like this:

Example 2.8. Nested loops with outer and inner loop

```
types= ["beautiful", "yellow", "small"]
tools= ["pen", "book", "rubber"]
for x in types:
  for y in tools:
   print(x, y)
```

Output:

```
beautiful pen
```

```
beautiful book
beautiful rubber
yellow pen
yellow book
yellow rubber
small pen
small book
small rubber
```

In the outer loop, **`x`** gets the first item as **`beautiful`**. Then it continues to the second line, which is the inner loop. **`y`** gets the first item of the inner loop, which is the **`pen`**. In the next line or the inner loop's body, the code prints **`x`** and **`y`**, so the output is **`beautiful pen`**. The first iteration of the inner **`for`** loop finishes, but in the outer loop, the statement hasn't finished yet. **`y`** gets the second item in the inner loop as the **`book`** and prints **`beautiful book`**. Finally, **`y`** gets the third item as **`rubber`** and prints **`beautiful rubber`**.

Now the inner loop finishes. It means that the outer loop's first iteration is finished, where it had the item as **`beautiful`**. So, the outer loop continues with the second item. So, in the outer loop **`x`** gets the second item as **`yellow`**. And in the inner loop, from the beginning, **`y`** gets the first item as **`pen`** and prints **`yellow pen`**. This continues until all the items are executed in the outer loop.

In the final iteration of the outer loop, the item gets a **`small`** value and continues the same way with all inner loops. Finally, we have ninw lines of output combining the inner and outer loops.

We use these nested loops many times in our network automation scripts. In the outer loop, we write a list for the device IP addresses, and in the inner loop, we write a list for the command list. The code gets one device IP and connects it to the outer loop; then, it executes all the commands in the inner loop and continues with the second device IP to execute all the commands.

Try…except statement

The **`try…except`** statement is generally used to catch errors in code, debug, to **`catch`** exceptions. For example, we write a code and get an error somewhere in our script that we cannot find. We can add a specific part of the code inside this statement to catch the issue. We can write anything understandable for us, like a **print** function that runs **There is an error in these lines**. So if the code gives an output, we understand that there is an error in that line.

In another example, we have a script that logs in five devices with SSH in the **for** loop. And we cannot reach the third device in the loop because that device has an SSH connection issue. If we run this code, we get an error because the script cannot be finished; it fails. But if we write all statements inside the **try…except** statement, we can continue the code until it is finished and can print the issue device with its IP address that cannot be reachable.

When the loop starts with the third device, it fails, so the **except** statement is run. And it prints the IP address and says it is not reachable. Then, it continues with the fourth device. So, we have two achievements here: our code is finished successfully, and we can catch the third device that cannot be reachable with SSH. So, in real-life scenarios, we can always use a **try…except** statement. The **try…except** statement is similar to **if** statements in some ways. We can also catch failures with the if statement, but it has limits.

If the **try** statement fails, it continues with the **except** statement. If it's successful, it continues the code by passing the except statement. The usage of the **try…except** statement is easy. We write **try** with a colon and then write the body of the **try** statement. After that, we write the **except** statement with a colon and write the body of the **except** statement. Optionally, we can add another statement, which explains that the **try** statement is successful.

```
try:
  Body_of_try
except:
  Body_of_except
else:
  Body_of_else
```

In the following example, we create a string variable, **Network Automation with Python**. In the **try** statement, we print a variable that we already created in the upper line. Then, we write the **except** statement and print the string **Failed**. In the **try** statement, the code runs without a problem, so the output of this code is **Network Automation with Python**.

```
a = "Network Automation with Python"
try:
  print(a)
except:
  print ("Failed")
```

Output: `Network Automation with Python`

In the next example, we change the `print` function in the `try` statement. This time, we print `b` as a variable, but there is no `b` variable in this code. If we don't write a `try…except` statement in this code, the code gives an error, like `name 'b' is not defined`, and even if we have some other codes after `print b`, the program doesn't continue to execute them because of the failure. But, if we write it with the `try…except` statement, the code still fails in the `try` statement, so it continues with the `except` statement. This time, it is not passed in the `except` statement. So, the output prints the function of a string, which is `Failed`.

```
a = "Network Automation with Python"
try:
  print(b)
except:
  print ("Failed")
```

Output: `Failed`

In the *Example 2.9*, we write two print functions: `print (a)` and `print (b)`. When we run this code, the code processes the first line in the `try` statement, which is `print (a)`. There is a value of the `a` variable above the code, so it gives the output `Network Automation with Python`. Then, it continues to the next line. Now, it's `print b`. It continues with an `except` statement, which has only one line of code, that is, the print function of the `Failed` string.

Example 2.9. Finding the issue code with try…except statement

```
a = "Network Automation with Python"
try:
  print(a)
  print(b)
except:
  print ("Failed")
```

Output:

```
Network Automation with Python
Failed
```

In this example, we change the order of the body in the **`try`** statement. The code starts executing the first line. There is no **`b`** variable in the code, so it catches an error and then continues with the **`except`** statement. It doesn't check the next lines in the **`try`** statement. So, the output of this example is different from that of the previous example:

```
a = "Network Automation with Python"
try:
  print(b)
  print(a)
except:
  print ("Failed")
```

Output: **`Failed`**

In the final example, we use the **`else`** statement additionally. We write the body of the **`else`** statement after the **`except`** statement. When we run this code in a **`try`** statement, it prints a variable, and then it continues with the **`else`** statement. It bypasses the **`except`** statement because there is no issue in the **`try`** statement.

If the **`try`** statement is successful, it continues with the **`else`** statement. In the example, we write the **`print`** function as a **`Successful`** string in the body of else. As a result, the output has two lines, which are **`Network Automation with Python`** and **`Successful`**.

```
a = "Network Automation with Python"
try:
  print(a)
except:
  print ("Failed")
else:
  print ("Successful")
```

Output:

```
Network Automation with Python
Successful
```

Conclusion

In this chapter, we learned about the basic functions of Python: **`print`** and **`input`** We compared the list, set, tuple, and dictionary data types, and we introduced the basics and methods of string, integer, list and dictionary data types. We also wrote several example scripts for these methods. We introduced statements and conditions deeply and wrote example scripts for **`if`**, **`for`**, **`while`**, **`break`**, **`continue`**, **`range`**, **`for…else`**, nested loops and **`try…except`** statements. We learned the usage of these statements and learned their syntax. We focused on the tricks and the important parts to use these statements.

In the next chapter, we will continue with file handling, **`RE`** module, and some advanced topics of Python, like functions and classes. After that, we will introduce the connection modules with SSH and telnet protocols to log in to real network devices. So, we will be ready to collect logs from network and system devices and modify them for our purposes.

Multiple choice questions

1. What will be the output of the following code?

```
x = 4
for i in range(x):
  x += 1
  print (x)
```

   a. 5 6 7 8
   b. 1 2 3 4
   c. 4 5 6 7
   d. 2 3 4 5

2. Which of the following is not a dictionary feature?
   a. Ordered
   b. Changeable
   c. Indexed
   d. Each item has keys and values

3. What will be the output of the following code?

```
x = "3 + 5"
print (x)
```

a. 8
b. “8”
c. 3+5
d. “3+5”

4. What will be the output of the following code?

```
x = "In google search, Python is the best for in all
scripting"
x = x.replace ("in", "X")
print (x)
```

a. google search, Python is the best for X all scriptXg
b. In google search, Python is the best for X all scripting
c. In google search, Python is the best for X all scriptXg
d. google search, Python is the best for in all scriptXg

5. What will be the output of the following code?

```
x = [2, 33, 222, 14, 25]
print (x[-2])
```

a. Error
b. 25
c. 14
d. 222

Answers

1. a
2. a
3. c
4. c
5. c

Questions

1. Write a script to calculate the perimeter of the rectangle from length and width parameters.
2. Write a script to convert degrees Fahrenheit to degrees Celsius.

   Formula: *Celsius = (5 / 9) * (Fahrenheit - 32)*
3. Write a script to find the grade of a student according to input, like 70, 90, and 50 scores.

   a. If the score is between 90 and 100, grade "AA"
   b. If the score is between 70 and 90, grade "BB"
   c. If the score is between 60 and 70, grade "CC"
   d. If the score is below 60, grade "FF" (It can also same as otherwise it's "FF"; else statement can be used.)

CHAPTER 3
Python Networking Modules

This chapter will focus on file handling in Python language. We will use new modules, like the OS module to modify files and directories, the **`RE`** module to manipulate logs, and netmiko, paramiko, and telnetlib modules to connect devices. We will focus on object-oriented programming in Python language as functions, classes, and modules.

Structure

In this chapter, we will cover the following topics:

- File handling
  - Open function
  - OS module
  - Word files
  - Excel files
- RE modules
  - RE module functions
  - Special sequences
  - Sets in RE module
- Advanced topics of Python
  - Functions
  - Creating modules
  - Classes

Objectives

We will explore Word, Excel, and text files in this chapter. We will also open, close, and modify files with the OS module. We will learn about the **RE** module so that we can manipulate logs of network devices and get the specific data needed in network automation. Further on, we will move on to advanced topics in Python, which are functions and classes. And we will create custom modules to import to the scripts.

File handling

We always display outputs with the print function, but in a more advanced way of showing results, we will use Word, Excel, and even text files. We will create, modify, and delete files according to our expectations in network automation scripts.

Open function

In Python, the **open** function is used for file handling. With this function, we can open, read, append, write, create, and close files. We can change the mode with some parameters to handle a file with our expectations. The files can be in text or log format. Refer to *Table 3.1*:

Mode	Description
"r"	Opens a file for reading, gives error if the file does not exist, (default value)
"w"	Opens for writing, truncating the file first
"x"	Creates the specified file, returns an error if the file exists
"a"	Opens a file for appending at the end, creates the file if it does not exist
"b"	Change the mode of the file from text to binary mode

Table 3.1: *The Open function parameters*

- **Read mode**: We use the "**r**" parameter in the **open** function to read a file. This is the default mode of the **open** function. In the next example, we try to open a "**test.txt**" file, which is in the same directory as our script. We read the file and display the file as output with a **print** function.

  First, we assign an **open** function to the **files** variable. We write the target file with its extension and the mode parameter inside the parentheses of the **open** function. In this case, even if we don't write the **r** parameter, the code works fine because the default mode of the **open** function is read mode as **r**. The target file can be in any text file format that the **open** function supports.

In the first line, we open the file in reading mode. In the next line, we read the opened file with the **files.read()** function and assign it to another variable **file_read**. In this **files.read()** code, we call the **files** variable. This variable equals **open("test.txt", "r")**. In the last line, we print the **file_read** variable. So, we can display a text file with this script.

```
files = open("test.txt", "r")    #Open a file, same as,
files = open ("test.txt")
file_read = files.read()        #Read a file
print(file_read)             #Print file that we read
```

Before running the previous code, make sure to create a **test.txt** file in the same directory as the code and fill the file with some strings:

```
test.txt
Hello World
This is Python Script
```

When we run the code, it shows the string of the **test.txt** file. The output will be as follows:

```
Hello World
This is Python Script
```

We called the **files** variable with the **read** function. Instead of writing the previous code, we directly wrote **open(test.txt), 'r').read()**, like in the given example. So, we didn't call any variable; we directly wrote the code that works with the **read** function. But the following code is more complicated to write and understand. So, we always assign some codes to variables and call those variables with other functions. When we write the preceding code, the code translates it by itself as the example following. So, both the codes are the same. As we said, the earlier version is much better.

```
file_read = open("test.txt", "r").read()
print(file_read)
```

- **Append mode**: We can append or add new entries to the current file. If there is no file, it creates a file. For the append feature, we use the **a** parameter. Its usage is similar to that of the read mode. We open a file with the **`a`** parameter with the **`open`** function, and then we add the **`write`** function with the new value. In the following example, after we add strings to the file, we also read the file and print it. We write the **`read`** function to read it after appending a string to display the final content of the **`test.txt`** file. The **`write`** function doesn't change anything in the original file. It only adds new entries after the last character in the original content. It doesn't go to the next line, as shown in the following output:

```
files = open("test.txt", "a")
files.write("Hello World")
files = open("test.txt", "r")
print(files.read())
```

Output:

```
Hello World
This is Python ScriptHello World
```

- **Write mode:** We can also overwrite a file. Append doesn't change anything in the original content, it only adds new lines. But the **`w`** parameter, which is also the **`write`** mode, deletes all the original content and writes its new value. The usage of overwriting is similar to that of append. We use the **`write`** function on both of them. Only the **`open`** function parameter is changed. In append, we use **`a`** and in overwrite, we use **`w`** as the write mode.

```
files = open("test.txt", "w")
files.write("This is new content !!!")
files = open("test.txt", "r")
print(files.read())
```

Output: **`This is new content !!!`**

- **Read by characters:** If we run the **`read`** function with empty parentheses, it reads the entire file content, as in the following examples. If we write a number, like **`10`** in the following example, it only reads the first **`10`** characters instead of all the characters in the file. So, we can read some parts of the content in the target file.

```
files = open("test.txt")
print(files.read(10))
```

- **Close function**: The **`close`** function is used to close a file. To use it, we call a **`close`** function with empty parentheses with the variable which we open the target file as **`files`**.

```
files = open("test.txt")
print(files.readline())
files.close()
```

Output:

```
Hello World
```

- **Create mode**: Create mode is to create a file. We use the **`x`** parameter in the **`open`** function to create a new file. When we run the following example, we must see a new file as **`test2.txt`** created in the same directory as our script running.

```
files = open("test2.txt", "x")
```

OS module

Python consists of modules and functions. One of the basic modules is the **`os`** module. It's generally used for operating system work, like deleting files and folders, changing the name of a file, or changing the directory of a file. There are also other features of this module:

- **Delete a file**: We can delete files on our PC by Python scripting. To do that, we can use the **`os`** module. To use the **`os`** module, we must import it as **`import os`**. After that, we need to call the **`remove`** function from the **`os`** module to delete a file. To call a function from its module, we use **`module_name.function_name`**. So, in our example, it's **`os.remove`**. Inside the **`remove`** function, we write the target file with its extension. When we run the code, we can see that the file is deleted in the current directory.

```
import os
os.remove("test.txt")
```

- **Create a folder**: We can create a directory or folder with the **`mkdir`** function. We need to import the **`os`** module before using this function.

```
import os
os.mkdir("testfolder")
```

- **Delete a folder**: We can delete a directory or folder with the **`rmdir`** function. We need to import the **`os`** module before using this function.

```
import os
os.rmdir("testfolder")
```

- **Getcwd function**: The `getcwd` function is used to find the full path of the script running.

  ```
  import os
  print(os.getcwd())
  ```

  Output:

  ```
  D:\Examples\test
  ```

- **Listdir function**: The `listdir` function is used to find all the content, including files and directories, in the current path of the script. We can specify the path inside parentheses if it's different from the current path. The code returns a list with all content. It's working as a `dir` command on Windows or `ls` command on Linux.

  ```
  import os
  print(os.listdir())
  ```

  Output: `['example.py', 'test2.txt']`

Word files

Python-docx module is used to create and modify word files in Python. It's a third-party module that is not built-in. So, to use this module, we need to install it with the `pip install Python-docx` command. We can create word documents, add headings, add paragraphs, change styles like bold or italic, add pictures and tables, and add rows in the table. We can save all these changes to a word file. We can do it without even opening a word file, only with Python code.

To call a document, we use the document function from the `docx` module `docx.document()`. To call each function, firstly, we must call the `document()` function. Instead of writing this function each time, we assign this function to the `document` variable. So each time we write `document`, it means `docx.document()`.

There are also other docx module functions, as shown in *table 3.2*. If you need more functions to check for a specific purpose, you can check their official website with the following link:

https://Python-docx.readthedocs.io/

Function	Description
`docx.Document ( )`	Call document function to use for other docx functions
`add_heading`	Add a new header in the document with the option to change the size from 0 to 9
`add_paragraph`	Add a new paragraph
`add_run`	Append characters (words, sentences) in a paragraph, with the osption to change the style to bold or italics
`add_picture`	Add a picture (JPEG or PNG format) in a document, with the option to change the size
`add_table`	Add table in a document in any size
`cell ( )`	Add text inside a table
`add_row`	Add a row in the table
`save (file_name)`	Save all changes in the code to word with a file name

Table 3.2: *Python-docx Module Functions*

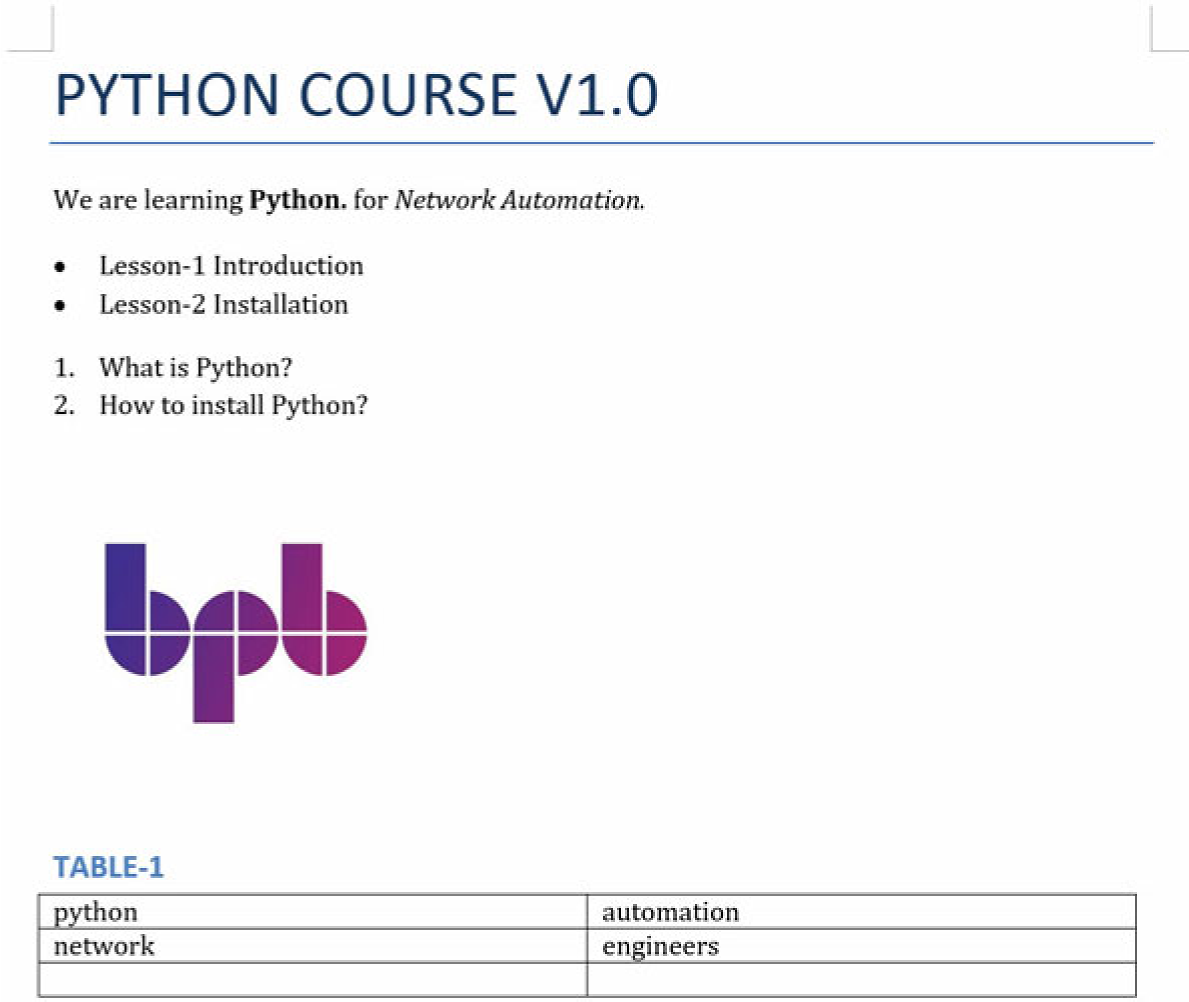
We can create word files, like in *Example 3.1*. For adding images, we need to add a JPG file to the same directory with our script. When we execute the code in *Example 3.1*, Python creates a word file named **`test.docx`** as we save with this name in the last line of our code. When we open the word file, we can see the following output. It starts with a big size header, followed by a paragraph including default, bold and italic styles. Then, we have a bullet list and a numbered list. Finally, we have a table where some cells are filled with the inputs. *Figure 3.1* is created by our script. In later projects, we can create any kind of Word file by writing Python scripts according to our demands.

Example 3.1: Create a Word file and modify with Python

```
import docx                        # Import Python-docx module
document = docx.Document()   # Call document function to call
other functions
document.add_heading('PYTHON COURSE V1.0', 0)       # Add
heading to word document
p = document.add_paragraph('We are learning ')     # Create a
new paragraph
p.add_run('Python. ').bold = True             # Add characters
in bold
```

```
p.add_run('for ')                        # Add characters in
default style
p.add_run('Network Automation.').italic = True        # Add
characters in italic
# Add 2 lines of bullet style text
document.add_paragraph('Lesson-1 Introduction', style='List
Bullet')
document.add_paragraph('Lesson-2 Installation', style='List
Bullet')
# Add 2 lines of Numbered list
document.add_paragraph("What is Python?", style='List Number')
document.add_paragraph("How to install Python?", style='List
Number')
document.add_picture('logo.jpg', width=docx.shared.Inches(2))
# Add Picture
document.add_heading('TABLE-1', 2)              # Add Heading
with size "2"
table = document.add_table(rows=2, cols=2)     # Add Table with
2 rows and 2 columns
table.style = document.styles['Table Grid']
cell = table.cell(0, 0)             # Fill Table by cells
cell.text = "Python"
cell = table.cell(0, 1)
cell.text = "automation"
row = table.rows[1]            # Fill Table by cells in
alternative way
row.cells[0].text = 'network'
row.cells[1].text = 'engineers'
row = table.add_row()                # Add new row to table
document.save('test.docx')               # Save all changes to
docx file
```

Refer to *Figure 3.1*:



PYTHON COURSE V1.0

We are learning **Python.** for *Network Automation.*

- Lesson-1 Introduction
- Lesson-2 Installation

1. What is Python?
2. How to install Python?

TABLE-1

python	automation
network	engineers

Figure 3.1: *Output of Example 3.1*

Excel files

The **`openpyxl`** module is used to create and modify an Excel file. It's also a third-party module, like the Python-docx module. So, we need to install the **`openpyxl`** module using **`pip install openpyxl`**. After the installation, we can import the **`openpyxl`** module. Another option is that instead of importing all the modules, we can only import specific functions of a module. In the following example, we import the **`Workbook`** function from the **`openpyxl`** module:

```
from openpyxl import Workbook
```

When we call the **`workbook`** function, we don't write **`openpyxl.Workbook()`** because we already called it in the previous line. If we only write **`import`**

Workbook, we must write **openpyxl.Workbook ()** instead of writing **Workbook()**.

We assign the **workbook()** function to the **workbook** variable. Then, we assign the **workbook** variable with active function to the **sheet** variable. This two-function assignment is required to write codes more clearly in the later sections. As the official document of the **openpyxl** module (**https://openpyxl.readthedocs.io/**) says; there is no need to create a file on the filesystem to get started with **openpyxl**. We just import the **Workbook** class and start work. So, with a workbook, we create an Excel file.

```
workbook = Workbook ()
sheet = workbook.active
```

After that, we add values in Excel blocks. In the following example, we choose **A1** block and assign its value as **Python**, B1 is assigned **Scripting**, **A2** is assigned **For Network**, and **B2** is assigned **Automation**.

```
sheet [ "A1" ] = "Python"
sheet[ "B1" ] = "Scripting"
sheet[ "A2" ] = "For Network"
sheet[ "B2" ] = "Automation"
```

We can also change the sheet name with the title function. In the following example, we change it to **Test Page**:

```
sheet.title = "Test Page"
```

We create our Excel file and modify it. Finally, we can save it to a file, like in the **Python-docx** module. We use the **save** function with the **workbook** variable that we created in the beginning. Inside the **save** function, we write the filename with its extension.

```
workbook.save ( filename="test.xlsx" )
```

When we execute the code, Python creates a file with the mentioned features and saves it in the same directory as our script. In *Example 3.2*, you can find the full code of the preceding example.

Example 3.2: Create an Excel file and modify it with Python

```
from openpyxl import Workbook

workbook = Workbook ()
sheet = workbook.active
sheet [ "A1" ] = "Python"
sheet[  "B1" ] = "Scripting"
```

```
sheet[  "A2" ] = "For Network"
sheet[  "B2" ] = "Automation"
sheet.title = "Test Page"
workbook.save ( filename="test.xlsx" )
```

We can also read values from an existing Excel file. This time, we import the **load_workbook** function from the **openpyxl** module.

```
from openpyxl import load_workbook
```

Then, we create a variable as **test.xlsx** string, which is an Excel file name and extension that we created in *Example 3.2*. Then, we call the **load-workbook** function with the filename:

```
filename="test.xlsx"
wb=load_workbook ( filename )
```

Like in *Example 3.2*, we use the **activate** function and assign it to the **sheet** variable.

```
sheet=wb.active
```

We create two variables: **b1** and **b2**. In the first line, we directly write sheet with **A1** inside square brackets. **A1** is the block name and number in the Excel file. In the second line, we call the **cell** function with writing row and column by numbers as **row=1** and **column=1**. In both instances of usage, we find the same block in the Excel file. The usage is different, but the result is the same.

```
b1=sheet['A1']
b2=sheet.cell ( row=1, column=1 )
```

After we got the values, we printed the **b1** and **b2** variables. If we can directly write the variable, we cannot see the value in the block. We see **<Cell 'Test Page' .A1>** in the output, so we must write **b1.value** to get the value in the specific block. The output is **Python** as string, which is the **A1** block value of the Excel file that we created in *Example 3.2*.

```
print( b1.value )
print( b2.value )
print( b2 )
```

Example 3.3: Read data from the Excel file

```
from openpyxl import load_workbook
filename="test.xlsx"
wb=load_workbook ( filename )
```

```
sheet=wb.active
b1=sheet['A1']
b2=sheet.cell ( row=1, column=1 )
print( b1.value )
print( b2.value )
print( b2 )
```

Output:

```
Python
Python
<Cell 'Test Page'.A1>
```

RE modules

The **`RE`** module is one of the most important modules in network automation for filtering data and logs. We can also find specific characters in files. **`Re`** means *regular expression*. **`RE`** module is a third-party module, so we need to install the module with **`pip install regex`**.

RE module functions

There are many **`re`** module functions. As listed in *Table 3.3*, we will focus on four main functions of the re module in this book. They will be the most useful ones for network automation. We must **`import re`** module to use all functions in the **`re`** module:

Function	Description
`findall()`	Returns all matches in a list
`search()`	Searches the string for a match, and returns the first match
`split()`	Splits the string with a specific character
`sub()`	Replaces the matched character with new values

Table 3.3: *RE module functions*

- **`findall()`**: The **`findall`** function is used to find all matches in a specific variable. When we use the **`findall`** function, we first write the characters or variables that we are searching for inside the parentheses. After a comma, we write the source string. So with the

findall function, we can find the specific values and return them in a list. If no match is found, it returns an empty list as an output. So, the input or source must be a string or byte data type. The result is always a list data type.

```
import re
re.findall (  Find_the_Characters  ,  Source_String  )
```

In the following example, we import the **re** module. We create a **string** variable as a **test**. Then, we write **re.findall** in parentheses. We write **o** and **n** characters as search parameters and test as the source string inside parentheses. We assign this function to the **x** variable and print it.

In this example, we try to find **o** and **n** characters together. The **test** variable is a string, which is **On Friday, I will study Python for Network Automation**. In this string, we have 3 of **o** and **n** together. But in the first letter, **O** and **n**, **O** is capital. It cannot match our condition. Since the Re module functions are case sensitive, the condition must match the same characters. There are two **on** in the string with the condition, so the function finds two of **o** and **n** in the test string. It returns an output as a list. If it doesn't find any matches, it creates an empty list.

```
import re
test = "On Friday, I will study Python for Network
Automation."
x = re.findall (  "on"  ,  test  )
print(x)
print(type(x))
```

Output:

```
['on', 'on']
<class 'list'>
```

- **search()**: The **search** function is used to check for the first match in the source string.

```
import re
re.search (  Find_the_Characters  ,  String_Name  )
```

We have a **test** variable, which is **I am learning Python for network automation**. We write the **re.search()** function with **o** and **n** to find the target and **test** as a source variable. If we execute this code, the result will be **<re.Match object; span=(18, 20), match='on'>**.

We have **o** and **n** two times, but the search function only gets the first match, which is in **Python word**.

```
import re
test = "I am learning Python for network automation"
x = re.search ( "on" ,  test )
print( x )
```

Output: **<re.Match object; span=(18, 20), match='on'>**

In the preceding output, **match** is the value that we are searching for, and **span** shows where the first matched value is. In this example, span is **18** and **20**. It's the character index in the source string. The matched characters are between the 18th and 20th characters. Finally, the matched value is **on**.

If we print the **x.start()** function, it shows the matched value in the first place, which is the 18th character. So, the result is **18**:

```
print( x.start( ) ) Output: 18
```

If we print the **x.end()** function, it shows the match value end place, which is the 20th character. So, the result is **20**:

```
print( x.end( ) ) Output: 20
```

If we want to know how many characters are there in total in the source string, we can use the **x.endpos()** function:

```
print( x.endpos ) Output: 43
```

If we want to check only the **span** value, we can write **x.span()**. The output shows it:

```
print( x.span( ) ) Output: (18, 20)
```

- **Split()**: The **split** function is used to split the string input into a list by dividing with specific characters. We write the **re.split()** function. Inside the parentheses, we write the target characters to divide by, and the string or **string** variable as input or source. There is an optional parameter to choose how many times the **split** function splits the matched value with the condition. By default, it divides for each match.

```
import re
```

```
re.split (Find_characters, String_name, (optional)
Number_of_times)
```

In the following example, we have a **`test`** variable as a string, which is **`Network Automation`**. We write the **`re.split("o","test")`** function. Inside the parentheses, we write the **`o`** string to match and the **`test`** variable as a **`string`** value. If we print **`x`**, the output of the code returns a list. So, we divide the string each time by a **`split`** function that finds the **`o`** character. We have three instances of **`o`** in the **`test`** variable, so three times divided, finally, we get 4 different items in a list.

```
import re
test= "Network Automation"
x = re.split ( "o", test)
print(x)
```

Output: `['Netw', 'rk Aut', 'mati', 'n']`

In the following example, we have the same split function, but this time we provide **`the number of times`** optional value as **`1`**. So, the code finds all matches, but it only divides from the first match. Even though we have three matches, only one of them is split. We have two items in the output list instead of four. If we write a higher value than the matched count, like in the example, we have three matches but write five in the function, the optional value will make no sense. It is eventually divided thrice.

```
import re
test= "Network Automation"
x = re.split ( "o", test, 1 )
print(x)
```

Output: `['Netw', 'rk Automation']`

- **`Sub()`**: The **`sub`** function is used to replace the matches with the new values. We write the **`re.sub()`** function; inside the parentheses, we write the original or current value, then the new value, and finally, the source or input string or a variable that needs to be used for the **`sub`** function. There is also an option to choose the number of times to replace matches, like in the **`split`** function, as an optional parameter. Instead of other functions like **`findall`** or **`split`** in the **`re`** module, the **`sub`** function's output is in string data type.

```
import re
```

```
re.sub (Find_characters, Replace_characters, String_name,
(optional) Number_of_times)
```

In the first example, we have the same test variable **Network Automation**. Inside the **re.sub()** function, we write the current value as **o**, then we write **x** as a new value, and finally, we write the source variable. In the output, **o** is replaced with **x** thrice in the **test** variable, and the output is a string data type.

```
import re
test= "Network Automation"
x = re.sub ("o"  ,   "x"  ,   test )
print(x)
```

Output: **Netwxrk Autxmatixn**

In the second example, we use the same parameters and add the optional parameter as **2**. So, the **sub** function only replaces the first two matches of **o** with **x**.

```
import re
test= "Network Automation"
x = re.sub ("o"  ,   "x"  ,   test, 2 )
print(x)
```

Output: **Netwxrk Autxmation**

Special sequences

In the RE module, there are special characters called the "**RE Special Sequences**". They can find all spaces or digits or only get the target characters. So, they are very powerful to manipulate strings or find the exact part from any kind of log. All these special sequences are used with the backslash sign.

Special sequences	Description
\A	Returns a match if the specified characters are at the beginning of the string
\d	Only returns the digits in the string
\D	Only returns non-digit values in the string
\s	Only returns spaces in the string

`\S`	Only returns characters except spaces in the string
`\w`	Returns a match where the string contains any word characters (characters from "a to z", "A to Z", digits from "0 to 9", and underscore)
`\W`	Returns a match where the string does not contain any word characters
`\Z`	Returns a match if the specified characters are at the end of the string

Table 3.4: *Re module special sequences*

From *Table 3.4*, there are lower case and capital letters with backslash signs; these are opposites of one another. For example, lower **d** is used to find digits, but capital **D** is used to find non-digits.

```
import re
test = "You can learn Python Scripting in 10 Weeks."
```

We have a string variable **test**, which is **You can learn Python Scripting in 10 Weeks**. In the first example, we write **\d**. It finds all the digits in the **test** string. We have **1** and **0** as digits in the **string** variable, so it creates a list with items for each match. There are two items on that list.

```
x = re.findall("\d", test)
print(x)
```

Output: `['1', '0']`

In the second example, we write **\D**. It finds and returns anything like characters from **a to z**, spaces, and signs instead of digits. **\D** is opposite of the **\d**.

```
x = re.findall("\D", test)
print(x)
```

Output: `['Y', 'o', 'u', ' ', 'c', 'a', 'n', ' ', 'l', 'e', 'a', 'r', 'n', ' ', 'P', 'y', 't', 'h', 'o', 'n', ' ', 'S', 'c', 'r', 'i', 'p', 't', 'i', 'n', 'g', ' ', 'i', 'n', ' ', ' ', 'W', 'e', 'e', 'k', 's', '.']`

In the next example, we use **\s**. It finds all the spaces in the string and writes each of them in a list with different items. We have seven spaces in the string, so we have seven space items in output:

```
x = re.findall("\s", test )
```

```
print(x)
```

Output: `[' ', ' ', ' ', ' ', ' ', ' ', ' ']`

In this example, we use `\w`, which finds any characters from `a to z`, `A to Z`, digits, or underscore.

```
x = re.findall("\w", test)
print(x)
```

Output: `['Y', 'o', 'u', 'c', 'a', 'n', 'l', 'e', 'a', 'r', 'n', 'P', 'y', 't', 'h', 'o', 'n', 'S', 'c', 'r', 'i', 'p', 't', 'i', 'n', 'g', 'i', 'n', '1', '0', 'W', 'e', 'e', 'k', 's']`

In the next example, we write `S\w`. It finds a character starting with `S`, then any characters a to z, A to Z, 0 to 9, or underscore, so the output is `Sc`.

```
Test = "You can learn Python Scripting in 10 Weeks."
X = re.findall("S\w", test)
print(x)
```

Output: `['Sc']`

In the next example, we use `\D` to find anything except digits. In the example, we said the match must start with `S`, and the result must include the `S` character.

```
x = re.findall("S\D", test)
print(x)
```

Output: `['Sc']`

This time, we use `\D` with a plus sign. The function matches until it reaches a digit and stops. So the result of `S\D+` is `Scripting in` until the first digit match.

This time, in the first line, we use `S\w+` with a plus sign for the `x` variable. The plus sign means that it continues until the match condition fails. In this example, after `S`, we have `c`. After that, we have `r`. It's also a letter character. `\w` matches characters from a to z, A to Z, digits, and underscore. This continues with `i,p,t,i,n,g`. There is a space after `g`, so this does not match the condition of `\w`. The result will be `Scripting`, starting with `S` and finishing with `g`.

```
x = re.findall("S\D+", test)
print(x)
```

Output: `['Scripting in ']`

In the second line, we write **`S(\w+)`** as the **`y`** variable and put **`\w+`** inside parentheses. So, we said that the match starts with **`S`** and continues if the next character is a to z, A to Z, digits, or underscore. If it fails to match the condition, finish the function. It is the same as the first line until now. Here, we find a match for **`Scripting`**, but we write parentheses for **`\w+`** matches. The function only gets the part inside the parentheses, so it doesn't get the **`S`** character because it's outside the parentheses. The output is **`cripting`**, without the **`S`** character. So the function catches all the matches but only returns the values inside the parentheses:

```
x = re.findall("S\w+", test)
y = re.findall("S(\w+)", test)
print(x)
print (y)
```

Output:

```
x => ['Scripting']
y => ['cripting ']
```

Sets in the RE module

In addition to special sequences, we have the *sets* in the **`RE`** module. Similar to special sequences, sets can match specific predefined characters to manipulate strings easier. **`RE`** module sets return a value for the match condition.

Sets are used by parameters. Without sets, it will only match the exact match together with the order. For example, if we write a match **`o,n`**, it will check all the **`o,n`** characters in a string together by order. If we write **`o,n`** in a set, it will check all strings with **`o`** or **`n`**. So, we can say that sets are the **`or`** parameters to check the strings.

- All **`RE`** module sets are always written with square brackets. We can write any of the alphabetic characters for sets.
- We can write any of the characters inside the square brackets. If we try to match values from **`a`** to **`p`**, we don't need to write all characters between **`a`** and **`p`**. Instead of this, we add hyphens between the characters and write them inside parentheses. We can also use hyphen signs in digits.

- We can use double sets. So, in the first set, we check digits from 0 to 5, and in the second, we check 0 to 9. So, this match starts from "0", "0" to "5", "9".
- If we try to find the matches with the **except** statement, we use `^` characters.

There are some example usages of sets in *Table 3.5*:

Sample sets	Description
`[abc]`	Returns the value that matches of "a", "b" or "c" in the string
`[a-p]`	Returns the value that matches characters in the alphabetic order from "a" to "p"
`[^abc]`	Returns the value that matches anything except "a","b" or "c"
`[012]`	Returns the value that matches 0, 1 or 2 as the digits
`[0-9]`	Returns the value that matches all digits from 0 to 9
`[0-5][0-9]`	Returns the value that matches all digits from 00 to 59
`[a-zA-Z]`	Returns the value that matches any alphabetical character from "a to Z"

Table 3.5: *Re module sets examples*

In the following example, we have the same string as the **`test`** variable. If we write the **`findall`** function with **`o`** and **`n`**, it checks **`o`** and **`n`** together in the string.

```
test = "You can learn Python Scripting in 10 Weeks."
x = re.findall("on", test)
print(x)
```

Output: `['on']`

In the second example, we write the same match with square bracket, which is a set. It checks **`o`** or **`n`** in target variable. If there is no square bracket, it checks **`o`** and **`n`**. But in this example, it's **`o`** or **`n`**. So we have two of **`o`** and five of **`n`** in the string. The result has seven items:

```
x = re.findall("[on]", test)
print(x)
```

Output: `['o', 'n', 'n', 'o', 'n', 'n', 'n']`

In the third example, we check digits from 0 to 9 with hyphen sign with sets. We have two items: **`1`** and `0`.

```
x = re.findall("[0-9]", test)
print(x)
```

Output: `['1', '0']`

Advanced topics of Python

We can write our automation scripts in a basic or more advanced way. If we use advanced features of Python in our scripts, they are more stable, require less code, and are easy to troubleshoot. Functions and classes are essential for advanced usage of the Python programming language, so we add these in the following scripts. We can also create custom-designed modules to call them anywhere in our code.

Functions

Functions are one of the most important parts of Python. They make our scripts simple and clean. For example, we have some scripts with many lines. We can write these codes each time we must use them, but it's not effective and not clear coding. So we create a function for that code once, and each time we need that code, we call the function. We used many functions. For example, we use the split function in the regular expression module. Like in the remodule functions, let's create a function. Remember that functions are reusable anytime and anywhere.

To define a function, we write **`def`** and we write the function name in parentheses. The line finishes with a colon. We write the body of the function in the following lines with indentation. When we write a function name with parentheses anywhere in the same code, we can call it:

```
def  Function_Name ( ) :
  Body_of_Function
function_Name ( )
```

In the following example, we create a **`test`** function in parentheses. Inside the function, we just write the **`print`** function with the value of the **`Network Automation`** string. After that, we write test in parentheses. We call this function in a different part of the code. When we call the **`test`** function, in that part of the code, it runs the **`test`** function and prints **`Network`**

Automation as the output. If the later lines are not in function indentation, we can understand that the body of the function is already finished:

```
def  test ( ) :
  print("Network Automation")
test ( )
```

Output: `Network Automation`

Functions with parameters

We create functions with parentheses. When we define a function, we can write variables inside parentheses. Then, in any part of the code, we call the function with the value of the variable.

```
def  Function_Name ( Variable ) :
  Body_of_Function
function_Name ( Value_of_Variable )
```

We define the **test** function with parentheses. Inside parentheses, we enter a variable as **platform**. Inside the **test** function, we have only one line as a print function. There is a string **I am learning Python for** plus a **platform** variable. Then, outside of the **test** function, we write the **test** function with the **platform** variable. In the first line, we write **Network Automation**. So when we call this **test** function, it prints a function and writes **Network Automation** when it sees the **platform** variable. In the second line, we change the variable to **myself**. So we call the **test** function twice, and we have two different outputs.

```
def test (platform) :
  print("I am learning Python for " + platform )
test ("Network Automation")
test ("myself")
```

Output:

```
I am learning Python for Network Automation
I am learning Python for myself
```

Suppose we call this function 10 times. We write the function once, and we call it 10 times in the code. If we don't write a function, we need to write this information again and again when we need it. It will create maintenance problems and need too much coding. It's not good coding. So in our codes, we try to create functions for repeatable codes.

In the second example, we define the **test** function and the variable of this function as **x**. Inside the function, we print the value for **x** multiplied by **2**. Outside the function, we call the **test** function with the value of **x** as **10**. So when we run this code, the output will be **20** because we have a **print** function for **10** multiplied by **2**, which is **20**.

```
def test ( x ) :
  print(x*2)
test( 10 )
```

Output: `20`

Functions with default parameters

In the previous example, we added a parameter but didn't set any default value on it. In the next example, we can add a default value to the **platform** parameter. If we call the **test** function without any values, it gets the default value. But if we call it with a value, it uses the new value.

```
def test (platform =  "Network Automation") :
  print("I am learning Python for " + platform )
test ( )
test ("myself")
```

Output:

```
I am learning Python for Network Automation
I am learning Python for myself
```

Call variables from functions

We can call variables outside of functions.

Example 3.4: Different usage of function variables		

Case-1:
`def test():`
` a=10`
` b=20`
` c= a+b`
`print (c)`
Output:
` print (c)`
`NameError: name 'c' is not defined`
	Case-2:
	`def test():`
	` a=10`
	` b=20`
	` c= a+b`
	` return c`
	`print (c)`
	Output:
	` print (c)`
	`NameError: name 'c' is not defined`
		Case-3:
		`def test():`
		` a=10`
		` b=20`
		` c= a+b`
		` return c`
		`x= test()`
		`print (x)`
		Output: `30`

In *Example 3.4*, **case-1**, we have three variables: **`a`** equals **`10`**, **`b`** equals **`20`**, and **`c`** equals **`a+b`**, which is **`30`**. When we try to print the **`c`** variable outside of the **`test`** function, we get an error that **`c`** is not defined. This is because we called the **`c`** variable outside of the **`test`** function.

In *Example 3.4*, **case-2**, **`c`** is inside a function. Any variable in a function has a local scope. Therefore, when **`c`** is printed outside, it says **`c is not defined`**.

In *Example 3.4*, **case-3**, we call the **`test`** function outside the function, and we need to assign it to a variable, which is **`x`** here. If we print the **`x`** variable, the code calls the **`test`** function and prints the return value. We can reach any variable from a function in this way. In this example, we can also write **`print("test()")`**, and we will get the same result.

Example 3.5: Global and local variables of functions

Case-4:

```
def test():
 a=10
 b=20
 c= a+b
 return c
 return b     #Code is not
 reachable
x= test()
print (x)
```

Output: 30

Case-5:

```
def test():
 a=10
 b=20
 c= a+b
 all = [a,b,c]
 return all
x= test()
print (x[1])     #Call "b"
print (x[2])     #Call "c"
```

Output:

20
30

Case-6:

```
def test():
 global c
 a=10
 b=20
 c= a+b
test()
print(c)
```

Output: 30

In *Example 3.5*, **case-4**, if we try to reach multiple variables from the function, we cannot write multiple returns. This is because when the execution comes to the first return, it understands that there is an exit from the function. So, the function finishes after the first return line. Any code after the **`return`** is not executed. In this example, the **`return b`** line is not executed.

In *Example 3.5*, **case-5**, to solve this issue, we can create a list inside a function and add all the variables that we try to reach outside the function. Then, we can write return the **`list`** variable. Outside the function, we can call the **`test`** function and assign it to **`x`**. If we try to reach the **`b`** variable, we can call **`x[1]`**, which is the second item of the **`all`** list.

In *Example 3.5*, **case-6**, as an alternative to using **`return`**, we can call variables outside the function with **`global`** variables. Inside the function, we can write **`global`** with the **`target`** variable so that we can call this variable from outside of the function. After we call the **`test`** function, we can call a global variable in this code.

Creating modules

When we create a function, we can only use it in the same Python file by default. We can create customized modules with functions, so we can import those modules and call our functions.

In the following example, the **`testmodule.py`** file, we create a function as **`test`**. We have the body of the function as one line of the **`print`** function. We save the file with the **`.py`** file extension, which is a Python file format.

When we create another Python file for **`example.py`**, we cannot directly call the **`test`** function from the **`test`** module Python file. We must import the module or file first. After that, we can call a function from that module with **`module_name.function_name`**. The **`test`** function has one parameter, so we can write one parameter to call the test function **`Network Automation`**, like in the example. We can call a function from another file, so we create a module.

Example 3.6: Create a module and call a function from that module

```
testmodule.py
def test (platform) :
  print("Hello " + platform )
example.py
import testmodule
testmodule.test ("World")
```

Output: **`Hello World`**

In this example, we have two different ways to call modules. We always used the first example until now. In the second example, we import the **`test`** function again. We don't write **`testmodule.test`** to call it; we directly write the **`test`** function. For example, if we need a single or a couple of functions from the module, we can only call those functions. We can use **`from module_name import function`**. When we call a function, we only write the function name, as in the following example, without the module name. Both usages are the same.

```
example.py
import testmodule
testmodule.test ("World")
```

Output: `Hello World`

```
example.py
from testmodule import test
test ("World")
```

Output: `Hello World`

Classes

Programming language has a philosophy of writing codes once and reusing them efficiently. **Object-Oriented Programming** (**OOPs**) is a very important section in programming. We use classes in almost all our Python scripts. Classes are code templates for creating objects. To create a class, we just write **`class class_name`**. We can write class names with or without parentheses. The line is finished with a colon. In the next line, it starts for the body of the class with indentation. We use classes in network automation to make the code simpler, understandable for other engineers, easy to troubleshoot and reusable.

```
Class Class_name():
  Body_of_class
```

In the following example, we have a **`test`** function, inside which we run the **`print`** function. If we run this code, there will be no output. We must call the function to execute it:

```
def test ( ) :
  print("This is a function")
test ( )
```

Output: `This is a function`

We write a **`test`** class that is a format similar to functions. Inside the class, we run the **`print`** function. If this is a function, we need to call this function in the code. But for class, we don't need to call it. If we run the following code, the result is the **`This is a class`** string. In classes, we don't need to call class instead of functions:

```
class test ( ) :
  print ("This is a class")
```

Output: `This is a class`

Conclusion

In this chapter, we learned file handling and RE modules to manipulate the logs we collect from network devices. We can divide logs or find specific keywords from the logs with the **RE** module. We learned to create custom functions, classes, and modules for more advanced usage of Python language. In later scripts, we always use those scripts. We created Word and Excel files without opening them and made them with scripts.

In the next chapter, we will log in to network devices with SSH and telnet protocols. We will collect logs from network devices and modify the data we receive into a more readable format, like collecting CPU levels, version, and model information.

Multiple choice questions

1. How can you delete the 'test' folder with the OS module?
    a. `os.remove ("test.txt")`
    b. `os.remove ("test")`
    c. `os.rmdir  ("test.txt")`
    d. `os.rmdir  ("test")`
2. How can you read five lines from a text file?
    a. `x = open("test.txt")`
       `print (x.read(5))`
    b. `x = open("test.txt")`
       `print (x.readline(5))`
    c. `x = open("test.txt")`
       `print (x(5))`
    d. `x = open("test.txt")`
       `print (x.readline())`

3. How can you find all the digits in x variable?
    a. `re.findall("0123456789", x)`
    b. `re.findall("[09]",x)`
    c. `re.findall("[0-9]", x)`
    d. `re.findall("\s", x)`
4. How can you import a function from a module?
    a. `import FUNCTION_NAME from MODULE_NAME`
    b. `import MODULE_NAME`
    c. `from FUNCTION_NAME import MODULE_NAME`
    d. `from MODULE_NAME import FUNCTION_NAME`

Answers

1. d
2. b
3. c
4. d

Questions

1. Find the phone numbers of the string given, including country codes:
   `x = "+44-1234567 (AA TELEKOM)/+33-7654321(BB TELEKOM)/+11-1111111(CC TELEKOM)"`
   `Output: ['+44-1234567', '+33-7654321', '+1-1111111']`
2. Find all the small letters of the following string:w
   `x = "This is a Network Automation Example created by Python."`

CHAPTER 4

Collecting and Monitoring Logs

This chapter will focus on connection modules and script examples. We will use netmiko, paramiko, and telnetlib modules to log in to network and system devices by **SSH** (**Secure Shell**) and telnet protocols. We will use these modules to collect data from multiple devices and modify it after logging in. We will also create a custom IP address validation tool and subnet calculator.

Structure

In this chapter, we will cover the following topics:

- Connection modules
    - SSH connection
    - Telnet connection
- Collecting logs
    - Collecting version and device information
    - Collecting CPU levels
    - Finding duplicated IP address
    - Collecting logs with multithreading
- Tools and calculators
    - IP address validator
    - Subnet calculator

Objectives

We must log in to the network and system devices to make automation by SSH and telnet protocol. We often use paramiko, netmiko, and telnetlib modules to connect devices with these protocols. We can connect one or more devices by the `for` loop and execute many commands in one script, and we can also create custom scripts to collect data from devices or create custom tools like a subnet calculator. Additionally, we can use the parallelism feature of Python with the multithreading module to log in to many devices simultaneously.

Connection modules

To simulate connection scripts in the next part of this book, we can use real devices and network simulators like **GNS3**, which is a free tool. It's recommended to use test devices or simulators to test the scripts. Real network devices have traffic, so it could be risky at the beginning of learning automation. For Cisco or Juniper devices and more vendors, GNS3 can run properly, and we can use the **eNSP** simulator for Huawei.

For later scripts, at least one network device is necessary, but it's better to test on multiple devices and improve yourself more deeply.

There are many options for networking modules in Python. Some of the most popular and powerful ones are paramiko, netmiko, NAPALM, nornir, and socket. There are plenty of options available for networking. You can choose any of them for network automation; each has its own advantages and disadvantages. During the course of the book, we will mainly focus on paramiko and netmiko modules.

There are also automation softwares that can be installed on a PC or server to make automation easier. Some of the most popular automation tools are Red Hat's **Ansible**, or Python modules such as **Paramiko, RE,** and **threading**. There are other popular tools as well, like Puppet, Saltstack, and Chef.

We can download and install GNS3 and VM Tool by following these steps:

1. We can download the free GNS3 tool from its official website. We must create a free account to download the tool, and we can install the tool on Windows, MAC, or Linux.

   https://www.gns3.com/software/download

2. After downloading and installing the tool on a PC, we need to download the GNS3 VM from the following link. You need to choose the specific VM to use. In this book, we use **`VMware Workstation Player`**, free for non-commercial use. So, we download the GNS3 VM for **`VMware Workstation and Fusion`**.

   https://gns3.com/software/download-vm

3. Finally, we must download and install the VMware Player tool from its official website.

https://www.vmware.com/products/workstation-player.html

4. After all installations are finished, we must open the VMware player and import the GNS3 VM from our PC in *Figure 4.1* by clicking on **`Open a Virtual Machine`**.

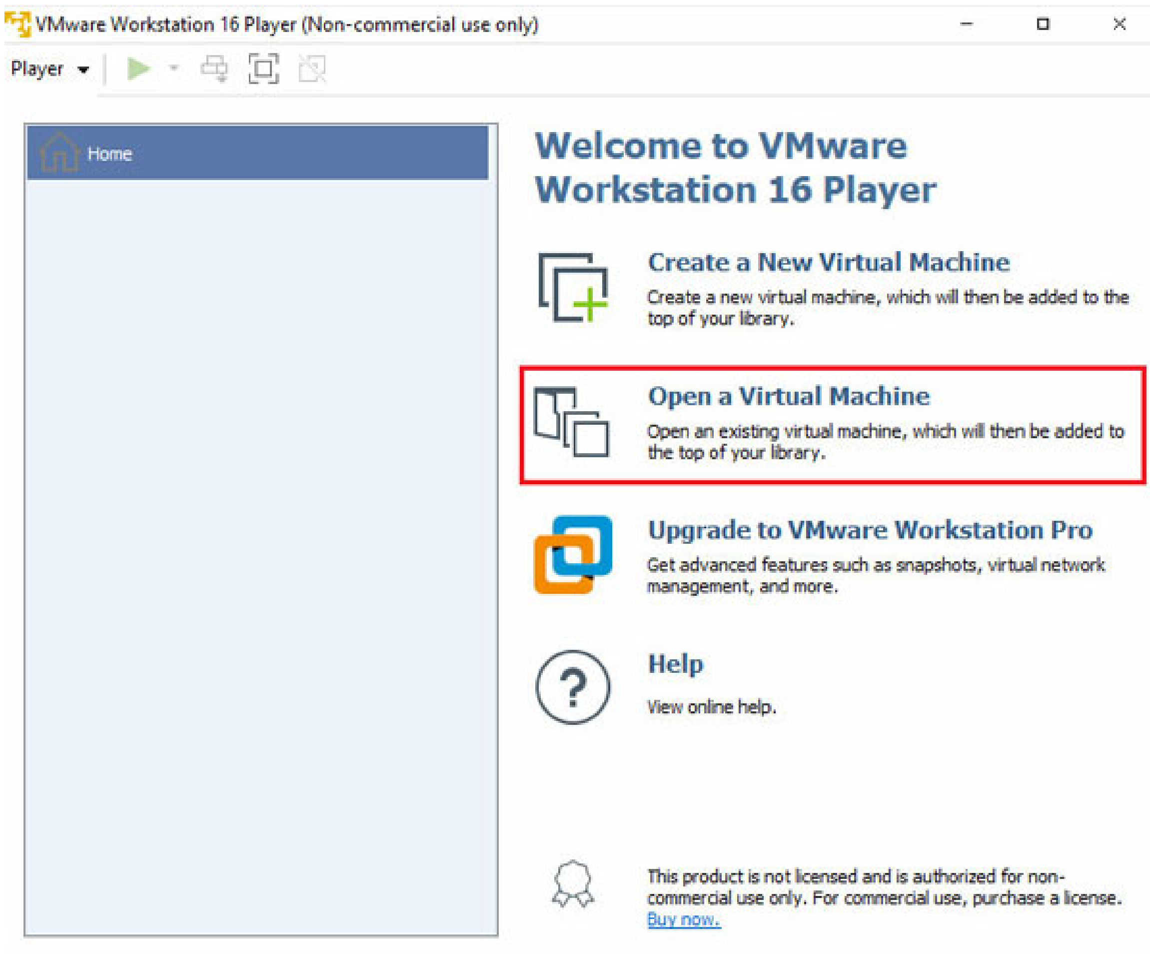


Figure 4.1: *Importing GNS3 VM to VMware Tool*

5. After we open the GNS3, from the **`Edit`** tab, we must open **`Preferences`**. In *Figure 4.2*, we must click on **`Enable the GNS3 VM`** and choose the VM engine to correct the VM tool. In the **`Settings`** tab, we must see **`GNS3 VM`** as the VM name. Remember that the VMware tool must be opened. We can apply the changes and close the window:

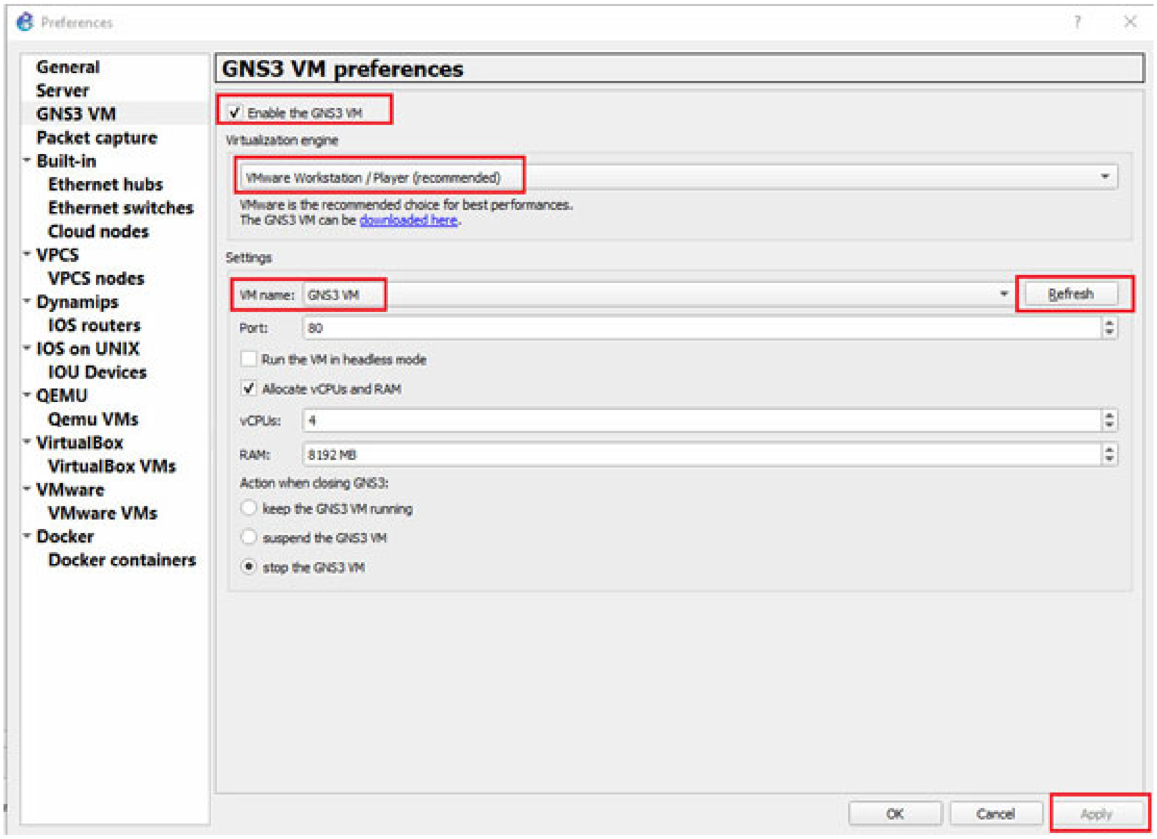


Figure 4.2: *GNS3 Configuration*

6. To verify whether the previous step is successful, we can check whether the LED indicator of the GNS3 VM turns green, as shown in [*Figure 4.3*](#). If it's not displayed or the LED indicator is red, the GNS3 VM installation has failed.

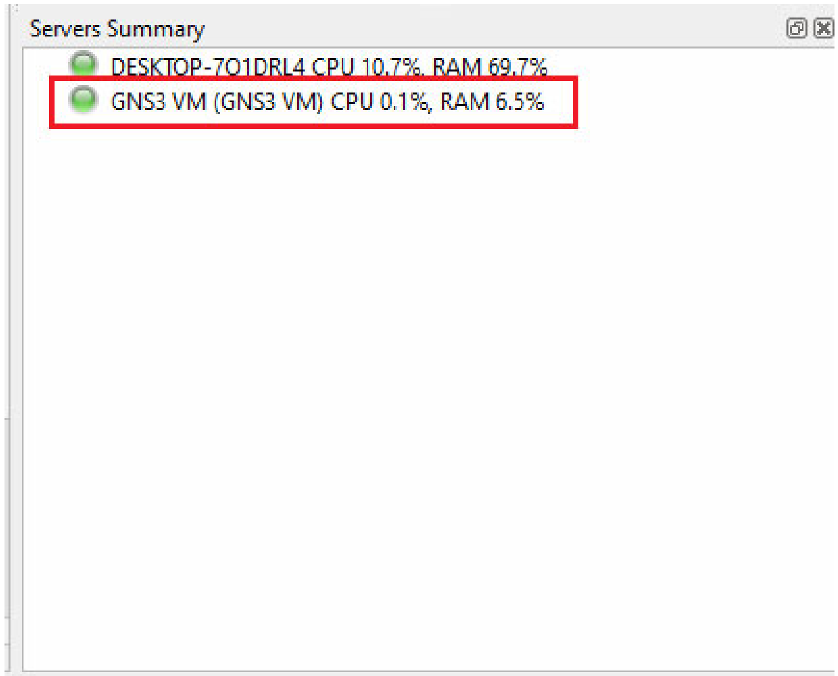


Figure 4.3: *Validation of GNS3 VM Installation*

7. If the LED indicator of the GNS3 VM is green, we can add appliances. We can enter the **`File`** tab and the **`Import Appliance`** button, and then we can import the GNS3 appliance file from our PC. We can download the appliance file from the following links. Each router model has a different appliance file, so you must download the correct appliance file according to your Cisco device file.

   https://gns3.com/marketplace/appliances

8. You must have the Cisco router ISO file to combine the appliance file. You can download the Cisco ISO files from Cisco's official website with your account.

   http://www.cisco.com

9. After we install the correct appliance with the ISO file, the router is added to the router list in the GNS3. We can create a new project and add the new router. Then, we can start the router and log in to configure it.

SSH connection

One of the most used protocols to log in to network and system devices is the SSH protocol. It's a secure connection protocol with many encryption options and protocol versions, such as version 1 and version 2, and SSH is much more secure than the telnet protocol. We often use the paramiko and netmiko modules to log in to the devices by the SSH protocol.

Paramiko module For SSH

There are options to make SSH and **Secure File Transfer Protocol** (**SFTP**) connections to any network or system devices with the paramiko module. This module has no support for telnet and **File Transfer Protocol** (**FTP**) connections. We can log in to any device with a username, password, and port number. We can execute `show or display` commands to monitor network devices and collect logs and can also execute configuration commands to change the configuration. Paramiko has some codes to connect a network device and execute commands as it is a third-party module. Hence, we need to install the paramiko module to project with `pip install paramiko`.

After the installation, we must import the paramiko module to our code.

```
import paramiko
```

We need to call the `SSHClient` function from the `paramiko` module for the SSH connection. This function is a high-level representation of a session with an SSH server. We assign this function to the `client` variable in the following code. When we write the `client` variable in the code, we understand that we call the `SSHClient` function.

```
client = paramiko.SSHClient()
```

When we try to log in to a network device for the first time, the our PC sends a message to the network device to trust or not. This is an SSH protocol security step. Once we click on the option to trust this device, it will never ask that question again. In paramiko, we push to change the `SSH authentication key` to `Trust ALL` with the `set missing host key policy` function and insert the `AutoAddPolicy` function from the paramiko module. So, we can add untrusted hosts and write the following code to pass that step with paramiko.

```
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

We write the IP address, port number, username, and password by order with the `connect` function. Thus, the code generates the connect information with the `connect` function.

```
client.connect (IP ADDRESS, PORT, USERNAME, PASSWORD)
```

We write another paramiko module with the `client` variable as the `invoke_shell` function. With this function, we request an interactive shell session on this channel and assign this function to the `commands` variable. Later, we will use this variable to execute commands on the network devices.

```
commands = client.invoke_shell ( )
```

SSH connection to any network or system device is ready. Since we are inside the device, it's time to **send** or **execute** commands. These can be **show** or **display** commands or any configuration commands in CLI according to vendor or OS type. So we **send** commands to the device with the **send** function, and then we use the invoke shell function to do that. After that, we write the **commands.send()** function. Thus, the **commands** variable is assigned to the **invoke_shell** function.

Inside of the **send** function, we write the command to run on the device. It sends the command but does not push the *Enter* button. So, after the command, we must write **\n**, which goes to the next line in Python programming.

Go to the next line means pushing the *Enter* button. So we send and run the commands on the devices:

```
commands.send ("COMMAND \n")
```

Finally, after we send the command to the device, we need to receive some output. We collect the outputs with the **Recv()** function. This function gets the data from the currently active channel. If we write 20 as a nbytes value, the code contains only the first 20 characters in the output. But, when we run **show running-configuration** in a Cisco device, the output is very long, so we need to enter high values to receive the output the device displays. Hence, we can enter **commands.recv(1.000.000)** as a 1 million value or much higher.

```
output = commands.recv ( NBYTES )
```

However, the received data format is in nbytes, so we need to change it to a human-readable format as **UTF-8** with the **decode** function. In the next example, we will execute the **decode** function with UTF-8 format and assign it to the **output** variable. If we print **output**, we can see all the output sent to the network devices. The basic log collection or sending command of the **paramiko** module is finished with this line:

```
output = output.decode ( "utf-8" )
```

We use seven different functions to log in to a device with SSH, which is a little bit complicated, but we only change a few parts of these lines in the later scripts, like the **connect** and **send** functions; the others remain the same. You don't need to understand at the beginning which function is used for what purposes. Rather, it's better to copy them and write the remaining part of your code. When you write many scripts repeatedly, you will easily understand what happens when we try to log in to a device in the background with the **paramiko** module. If you want more details about the **paramiko** module, you can check out the website www.paramiko.org, which is the official paramiko module web page.

Connect 1 device with Paramiko

We can log in to a device with SSH and collect logs. Before writing the code in pycharm, in this basic *Example 4.1*, we try to log in to a Cisco device and collect version information with the `show version` command. After that, we try to display the output of that command in Pycharm:

1. We start by importing the `paramiko` module and the `time` module. Time module creates delays in seconds in the code:
   ```
   import paramiko
   import time
   ```
2. We write an `SSHClient` function for SSH connection:
   ```
   client = paramiko.SSHClient()
   ```
3. We write `set_missing_host_key_policy` to pass the first login authentication process:
   ```
   client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
   ```
4. We enter the IP address, port number `22`, the default port number of SSH protocol, username as root, and password as a test:
   ```
   client.connect("10.10.10.1", 22, "admin", "cisco")
   ```
5. We write an `invoke_shell` function to request an interactive shell session on this channel:
   ```
   commands = client.invoke_shell()
   ```
6. It's time to send commands to the device and send the `show version` command with `\n`:
   ```
   commands.send("show version \n").
   ```
7. We call the `sleep` function from the `time` module. When we run a piece of code, it executes the whole code very fast. We `send` commands to the device and get the data with the receive function. For example, when we execute the `show run` command in a Cisco device, it takes time to display the output; it's not instant.
8. So, we should add a delay after the line that we send the command to the device. We can collect the entire output with this delay. If we did not add any time delays, some parts of the logs might not have been collected. That's why we write some delays after the `send` command. In this example, it is one second as `1`.

9. After the **send** function, the program waits for 1 second and then continues with the **receive** function. So we can get all the output correctly. We can change the **sleep** function value at any time:

   ```
   time.sleep(1)
   ```

10. We send the command and wait for a second. Now it's time to receive the data. We write the nbytes value as **1000000**, so this code will receive outputs with these nbytes. The value is quite enough to receive all the output correctly.
11. Then, we decode the **nbyte** format to **UTF-8**, which is the human-readable format. Finally, we can print the output as a string. Code displays the output of the **show version** command of one device:

    ```
    output = commands.recv(1000000)
    output = output.decode("utf-8")
    print (output)
    ```

In *Example 4.1*, we can log in to a single network or system device by the **paramiko** module and execute any commands in the device.

Example 4.1: Connect to a single device with Paramiko

```
import paramiko
import time
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect("10.10.10.1", 22, "admin", "cisco")
commands = client.invoke_shell()
commands.send("show version \n")
time.sleep(1)
output = commands.recv(1000000)
output = output.decode("utf-8")
print (output)
```

Running configuration commands with Paramiko

We can also execute configuration commands in the network devices with the paramiko module:

1. To do that, we use the **send** function, like in the previous example. In *example 4.2*, we try to configure a description of an interface in Cisco Router.

2. Then we show the configuration of this interface: whether or not a description is created. The beginning, until the **`send`** function, is the same as in *Example 4.1*. After that, we run the **`configure terminal`** command to enter configuration mode.
3. Then we run **`interface gigabitethernet 0/1`** to enter interface mode and run a **`description`** command to add or change the description of this interface.

```
commands.send( "configure terminal \n")
commands.send( "interface gigabitethernet 0/1 \n")
commands.send( "description TEST\n")
commands.send( "do show run interface gigabitethernet 0/1
\n")
```

4. After that, we run the "**`show`**" command to check the related interface configuration, and we run the **`sleep`** function and get the output.

```
time.sleep(.5)
output = commands.recv(1000000)
output = output.decode("utf-8")
print (output)
```

5. We didn't enter the **`sleep`** function after each command. If you are experienced in the CLI, many configuration commands are set instantly, but some of them are set gradually. However, the **`show`** command displays some outputs, so it always takes time.
6. It's recommended to put some delays between commands. We will discuss an alternative to using the **`sleep`** function in the later chapters. We will write a script that checks whether output is finished, and the program waits if not. We use it in a **`for`** loop. In this example, we will keep this as simple as as we can.
7. You may think that we had to write a multiline code just to run one command in a device. Running a command in CLI instead of a run script is much faster in this situation.
8. However, if we run 10 commands in 100 devices and get some specific data from them, doing this task in CLI is a waste of time. That's why we use network automation as network engineers.

Example 4.2: Running configuration commands in a single device with Paramiko

Configuration change:

```
conf t
interface g0/1
description TEST
do show run interface gigabitethernet 0/1
```

Python code:

```
import paramiko
import time
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect("10.10.10.1", 22, "admin", "cisco")
commands = client.invoke_shell()
commands.send( "configure terminal \n")
commands.send( "interface gigabitethernet 0/1 \n")
commands.send( "description TEST\n")
commands.send( "do show run interface gigabitethernet 0/1 \n")
time.sleep(1)
output = commands.recv(1000000)
output = output.decode("utf-8")
print (output)
```

Connect to multiple devices with Paramiko

In *Example 4.1* and *Example 4.2*, we could log a device in with SSH and collect many logs. But we have not tried to log in multiple devices yet; if we try with those examples, we must write the same `paramiko` functions and send the same commands multiple times. It's not a good way to write code, and it may introduce more errors in the code.

If we have 100 devices and 20 commands to run, or even a single command to run, we must create 100 sessions in Secure CRT or in another SSH connection tool. If you have a much larger network, over 1,000 devices, it's almost impossible. In this situation, we can use Python scripting to make things easier for us.

We will just modify some parts in our last example to automate the code and use loops for repeatable actions in the programming. Connecting and running multiple commands in devices are repeatable actions, and computers are much better than us for repeatable things. By using a for loop, we can connect many devices.

In *Example 4.3*, we will use two loops as nested loops, i.e., inner and outer loops. One of the loops is used for IP addresses to log devices in to each loop, and the other one is used for the commands that we run in a single device each time. So if we check the connection timeline, we must log in to a device, and then we must run the commands. So, the first loop, which is the outer loop, is used to connect to the device, and the second loop, which is the inner loop, is used to send commands.

1. Import `paramiko` and `time` modules.
   ```
   import paramiko
   import time
   ```
2. Create a list of `hosts` for IP addresses of the devices. In the following code, we have three IP addresses. For commands, we also create another list as `command_list` for the commands that run in each device:
   ```
   hosts = ["10.10.10.1","10.10.10.2","10.10.10.3"]
   command_list = ["conf t","int g0/0","description NEW-TEST"]
   ```
3. Enter the outer loop and check the IP address from the hosts list and make the connection. Then, continue with the body of the `for` loop, which includes the inner loop. In the outer loop, write the function to connect the device by executing the **invoke_shell** function. In the inner loop, we get commands from `command_list`.
4. It chooses the first item in `command_list` and runs the body of the `for` loop. After it finishes, it gets to the second iteration or item from the inner loop. The inner loop continues until all items are chosen.
5. After all inner loop iteration is finished, the first statement of the outer loop is finished. So, we collect and execute all these commands in the first device. After that, outer loop gets the second item or iteration and continues with the body of the `for` loop. This continues until all items in the outer loop are completed. When we print `output`, we can see all outputs for each loop.
   ```
   #Outer Loop
   for ip in hosts:
     client = paramiko.SSHClient()
     client.set_missing_host_key_policy(paramiko.AutoAddPolicy()
     )
     client.connect (ip,22,"admin","cisco")
     commands = client.invoke_shell()
     #Inner Loop
   ```

```
for command in command_list:
  commands.send("{} \n".format(command))
  time.sleep(1)
  output = commands.recv(1000000)
  output = output.decode("utf-8")
  print (output)
```

Example 4.3: Connect to multiple devices with Paramiko

```
import paramiko
import time

hosts = ["10.10.10.1","10.10.10.2","10.10.10.3"]
command_list = ["conf t","int g0/0","description NEW-TEST"]

for ip in hosts:
  client = paramiko.SSHClient()
  client.set_missing_host_key_policy(paramiko.AutoAddPolicy()
)
  client.connect (ip,22,"admin","cisco")
  commands = client.invoke_shell()

  for command in command_list:
   commands.send("{} \n".format(command))
   time.sleep(1)
   output = commands.recv(1000000)
   output = output.decode("utf-8")
   print (output)
```

Netmiko module for SSH

Netmiko is another third-party connection module like paramiko. The official project document (**www.github.com/ktbyers/netmiko**) says that netmiko is a multi-vendor library that simplifies paramiko SSH connections to network devices.

The netmiko module has numerous features that are better to use than the paramiko module, such as:

- It can support more than 40 vendors like Cisco, Juniper, Huawei, and Nokia. Netmiko is created on top of the paramiko module.
- It is based on paramiko and supports SSH, telnet, and SCP connections. Instead of paramiko, we can log in to many vendor network devices by telnet in the `netmiko` module.

- Netmiko has simplified code. The **paramiko** module had many functions to run in the code, code lines are fewer in netmiko, making it easier to understand.

Netmiko module supports almost all the network devices. As the official page says, there are three categories for support: regularly tested, limited tested, and experimental. However, major network vendors are supported by the netmiko module. For a full and updated list, you can visit the following link:

www.ktbyers.github.io/netmiko/PLATFORMS.html

Connect a single device with Netmiko

In *Example 4.4*, we will write a basic **netmiko** module to log in to a single Cisco router. For other vendors, only some parameters change:

1. First, we must install and import the **netmiko** module. Instead of importing the whole module, we can import the "**Netmiko**" function from the **netmiko** module.

   ```
   from netmiko import Netmiko
   ```

2. Then, we must write device information inside the device variable as a dictionary. We write host as IP address, username, password, and device type as vendor type. Optionally, we can choose timer delay between commands as a **global_delay_factor**, in seconds, as we did with the time module in the paramiko examples. These keys are predefined in the **netmiko** module, and we add values to specific keys. There are also many keys to this part, which you can check on the **netmiko** module official website.

3. For the IP address, we enter the **host** key. We add the **password** and **username** keys for password and username. We must enter the device type with the **device_type** key. For other devices, you can check device type usage. We will use **juniper_junos** as the value for Juniper and **huawei** as the value for Huawei.

   ```
   device = {
     "host": "10.10.10.1",
     "username": "admin",
     "password": "cisco",
     "device_type": "cisco_ios",
     "global_delay_factor": 0.1,
   }
   ```

4. Then, we must call the **netmiko** function with the device list as the device variable. We write two stars before the **dictionary** variable.

   ```
   net_connect = Netmiko(**device)
   ```

5. Now, it's time to send configuration commands to the Cisco device. We create a list as a **command** variable and enter the commands by order. First, we enter the interface and change the description of that interface. You can see that there is no **configure terminal** in the list. Netmiko understands the device type we write in the dictionary as Cisco, so it automatically enters configuration mode.

   ```
   config= ["interface GigabitEthernet0/0", "description TEST"]
   command = "show run interface GigabitEthernet0/0"
   ```

6. Then, we create a **config_output** variable and call the **send_config_set** function to send the configuration commands. Inside parentheses, we write the **command** variable as a list of commands, and the code enters the configuration mode automatically. We cannot run the show commands inside this function. If we want to do that for Cisco, we must enter **do** before the show letter. Otherwise, we can run show commands directly with the **send_command** function.
7. Usage is also the same with the **send_config_set** commands. We cannot run **show** commands in Cisco configuration mode or we must write do before the show command.

   ```
   config_output = net_connect.send_config_set(config)
   show_output = net_connect.send_command(command)
   ```

8. Finally, we call the **disconnect** function to close the SSH session and print the output. Netmiko has clear code to run according to the **paramiko** module. It automatically does many things in the background, and we almost write only the device information and command lists.
9. In the netmiko module, commands are automatically run with the **send_config_set** function one-by-one in order. So if we try to connect multiple devices, we don't need to create nested loops. Only one **for** loop is enough for the device list. For the command list, netmiko does the loop action for us.

   ```
   net_connect.disconnect()
   print(config_output )
   print(show_output )
   ```

Example 4.4: Connect to a single device with Netmiko

```
from netmiko import Netmiko
```

```
device = {
  "host": "10.10.10.1",
  "username": "admin",
  "password": "cisco",
  "device_type": "cisco_ios",
  "global_delay_factor": 0.1,
}
net_connect = Netmiko(**device)
config= ["interface GigabitEthernet0/0", "description TEST"]
command = "show run interface GigabitEthernet0/0"
config_output = net_connect.send_config_set(config)
show_output = net_connect.send_command(command)
net_connect.disconnect()
print(config_output)
print(show_output)
```

Output:

```
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
Router-1(config)#interface GigabitEthernet0/0
Router-1(config-if)#description TEST
Router-1(config-if)#end
Router-1#
Building configuration…
Current configuration : 148 bytes
!
interface GigabitEthernet0/0
  description TEST
  ip address 10.10.10.1 255.255.255.0
  duplex auto
  speed auto
  media-type rj45
  no cdp enable
end
```

Connect to multiple devices with Netmiko

In *Example 4.5*, we will log in to multiple devices with the **`netmiko`** module. There are three devices, and all of them are added to a list called **`device_list`**. We create a **`for`** loop to call all devices one by one and write the same netmiko functions inside the loop.

Example 4.5: Connect to multiple devices with Netmiko

```
from netmiko import Netmiko

device1 = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device_list = [device1, device2, device3]

for host in device_list:
  net_connect = Netmiko(**host)

  config = ["interface g0/0", "description TEST-NETMIKO"]
  command = "show version"

  config_output = net_connect.send_config_set(config)
  show_output = net_connect.send_command(command)

  net_connect.disconnect()
  print("Config is starting from here:", config_output)
  print("Logs are starting from here:", show_output)
```

Telnet connection

There are some options for telnet connections. We are focusing on telnetlib and netmiko modules to log in to network devices with the telnet protocol in this book. As netmiko is more stable in SSH connections, netmiko is also better for telnet connections.

Telnetlib module for telnet

We can make SSH connections with paramiko and netmiko modules. Almost all network devices use SSH to make connections, and it's more secure, but we also have a telnet connection protocol. So if we need some devices to connect with telnet, we have the **`telnetlib`** module in Python.

In *Example 4.6*, we will write a telnet connection script to log in a single device with the **`telnetlib`** module. We can connect many devices with a **`for`** loop:

1. First, we must install and import the **`telnetlib`** module into the code.

   ```
   import telnetlib
   ```

2. Then, like paramiko, we will assign IP address, username and password to new variables.

   ```
   ip = "10.10.10.1"
   user = "admin"
   password = "cisco"
   ```

3. Then, we will make the telnet connection and send the username and password. After that, we can send any command. We will use the **`telnet`** function from the **`telnetlib`** module and enter the IP address and port name. Optionally, we will have a timeout count in the **`telnet`** function, and the code will never end if we don't set it before running the **`read_all`** function. Even though timeout is optional, we must enter the value.

   In SSH protocol, we set a username and password, and then directly log in to a device; the authentication process takes place in the background. But in telnet, we must enter username first, and then the password in CLI.

   So, we send username and password values as a command. When the code match to **Username:** string, it sends the username. When the code match to **Password:** string, it sends the password.

   We write the **`read_until`** function and the string. Inside parentheses, there is a **`b`** letter and a **`string`**. **`b`** is for the **`byte`** data type. We cannot use these functions directly with string; we must use the **`byte`** data type. We write **`b`** before the string, so it is a **`byte`**. The output gives an error if we remove **`b`** in these functions. It says **`argument should be integer or bytes-like object, not string'`**.

   With the **`read_until`** function, we can wait for the code until we see the output. We wait until the code matches the **`Username:`** value. If the code catches it, it continues with the following line. The next line sends a command with the **`write`** function. We send the username as a variable, but the **`user`** variable is a string. We must convert it to byte data type. To convert the variable, we write **`variable.encode`** and **`ASCII`** mode inside parentheses. Then, we write the username. After that, we must push the **`enter button (or go to the next line)`** with **`\n`**. We use it as bytes again. We wait for `Password:` output. We use the same `read_until` function. Then, we send command with the `write` function.

```
tel = telnetlib.Telnet(ip, 23, timeout=1)
tel.read_until(b"Username:")
tel.write(user.encode('ascii') + b"\n")
tel.read_until(b"Password:")
tel.write(password.encode('ascii') + b"\n")
```

4. We use the `write` function to send commands and an `exit` command to close the telnet session.

```
tel.write(b"show ip interface brief\n")
tel.write(b"exit\n")
```

5. Finally, we can read the output and print it. To read all results, we run the `read_all` function. We use the `decode` function with ASCII inside parentheses to translate the output to string.

```
print(tel.read_all().decode('ascii'))
```

When we run the telnet connection script, the code may give an error or never finish, so we need to add the `try…except` statement in the `while` statement to avoid any problems. The issue code is in the `telnetlib.py` Python file, the path to which is given as follows. We need to change the `telnetlib.py` file as in *Table 4.1*:

```
C:/Users/USER_NAME/AppData/Local/Programs/Python/Python310/Lib/telnetlib.py
```

<table>
<tr><th>BEFORE:</th><th>AFTER:</th></tr>
<tr><td><pre>def read_all(self):
 """Read all data until EOF; block until
 connection closed."""
 self.process_rawq()
 while not self.eof:
  self.fill_rawq()
  self.process_rawq()
 buf = self.cookedq
 self.cookedq = b''
 return buf</pre></td><td><pre>def read_all(self):
 """Read all data until EOF; block until
 connection closed."""
 self.process_rawq()
 while not self.eof:
  try:
   self.fill_rawq()
   self.process_rawq()
  except:
   break
 buf = self.cookedq
 self.cookedq = b''
 return buf</pre></td></tr>
</table>

Table 4.1: *Changing telnetlib module*

In *Example 4.6*, we can log in to a network device with the `telnetlib` module. We can also log in to multiple devices by sending multiple commands by adding the **for** loop in *Example 4.6*.

Example 4.6: Connect to a single device with the telnetlib module

```
import telnetlib
ip = "10.10.10.1"
user = "admin"
password = "cisco"
tel = telnetlib.Telnet(ip, 23, timeout=1)
tel.read_until(b"Username:")
tel.write(user.encode('ascii') + b"\n")
tel.read_until(b"Password:")
tel.write(password.encode('ascii') + b"\n")
tel.write(b"show ip interface brief\n")
tel.write(b"exit\n")
print(tel.read_all().decode('ascii'))
```

Connect to multiple devices with telnetlib

In *Example 4.7*, we will write a script to log in to multiple devices and execute various commands with the **telnetlib** module. It's similar to the **paramiko** module. We will create nested loops. In the first loop, we will log in to the devices, and in the second loop, we will execute the commands in the devices.

Example 4.7: Connect to mutiple devices with telnetlib module

```
import telnetlib

host = ["10.10.10.1","10.10.10.2","10.10.10.3"]
user = "admin"
password = "cisco"
command = ["terminal length 0","show ip interface brief","show
clock","exit"]

for ip in host:
  tel = telnetlib.Telnet(ip, 23, timeout=1)
  tel.read_until(b"Username:")
  tel.write(user.encode('ascii') + b"\n")
  tel.read_until(b"Password:")
  tel.write(password.encode('ascii') + b"\n")

  for config in command:
   tel.write(config.encode("ascii") + b"\n")
   print(tel.read_all().decode('ascii'))
```

Netmiko module for telnet

We can make telnet connection with **netmiko** module. Not all brands support telnet connection in netmiko, but netmiko supports major vendors like Cisco, Juniper, and Huawei.

1. We write almost the same code for the SSH connection. First, we import the **netmiko** module.

   ```
   from netmiko import Netmiko
   ```

2. Then, we enter the device information. As a device type, we add **_telnet** for each device type to connect by telnet. Normally, to connect a Cisco device, the SSH device type is **cisco_ios**. For telnet connection, we write **cisco_ios_telnet**. There is one more thing here: we add the global delay factor in SSH connections, and it's an optional parameter to add delay for connections. If we don't set this parameter for telnet connection, we can also log in to the device. However, there is a possibility not to send the username and password to the device during connection because the device connection is slow and the program can create an error. It's better to set a global delay factor for telnet connection min to half a second.

   ```
   device = {
     "host": "10.10.10.1",
     "username": "admin",
     "password": "cisco",
     "device_type": "cisco_ios_telnet",
     "global_delay_factor": 0.5
   }
   ```

3. We call the **netmiko** function. Then, we create a list of commands by order. We send configurations with the **sending_config_set** function.

   ```
   net_connect = Netmiko(**device)
   command = ["interface g0/0", "description TEST"]
   output = net_connect.send_config_set(command)
   print(output)
   ```

Example 4.8: Connect to devices with netmiko module with telnet protocol

```
from netmiko import Netmiko

device = {
  "host": "10.10.10.1",
  "username": "admin",
```

```
  "password": "cisco",
  "device_type": "cisco_ios_telnet",
  "global_delay_factor": 0.5
}
net_connect = Netmiko(**device)
command = ["interface g0/0", "description TEST"]
output = net_connect.send_config_set(command)
print(output)
```

Output:

```
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
Router-1(config)#interface g0/0
Router-1(config-if)#description TEST
Router-1(config-if)#end
Router-1#
```

Collecting logs

In this section, we collect logs from network devices. Examples are based on the Cisco devices, but they can be used by any vendor to replace the commands to execute on a device. We collect the device's software version, model information, and CPU levels. We can also search for data in devices and try to find the duplicated IP addresses in the network. Additionally, we can use the advanced feature of Python language, called **multithreading**. Thus, we use this module to log in to multiple devices simultaneously as it saves time.

Collecting version and device information

In *Example 4.9*, we try to collect IP address, software version, model information, vendor type, and hostname information from the Cisco routers and save them to an Excel file. To do that, we log in to the three Cisco routers with the `netmiko` module and execute the `show version` command. We match the specific data with the `findall` function from the `RE` module and add it to the different lists. In the final part, we save all the information in an Excel file.

1. First, we import the required modules. We call the `Netmiko` function from the `netmiko` module and import the `RE` module. And finally, we call the `DataFrame` function from the `pandas` module. The `pandas` module is

often used for data analysis and science. We use the **DataFrame** function to create an Excel file from a list.

```
from netmiko import Netmiko
import re
from pandas import DataFrame
```

2. We add three devices as Cisco routers with the following information. And we create a list of those three devices as **host** variable. After that, we create a **string** variable as **command** that we run in the devices as **show version**.

```
device1 = {"host": "10.10.10.1", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
host = [device1, device2, device3]
command = "show version"
```

3. In the following code, we write a single-line **for** loop. This is another usage of the **for** loop. In this example, we used it to create five different lists for each variable name. After that, we created the main **for** loop of the code:

```
ip_list, version_list, model_list, vendor_list,
hostname_list=([] for i in range(5))
for ip in host:
```

4. We write the **try**...**except** statement to match whether we can log in to the device. If we cannot log in to the device, the **except** statement is executed, and it continues with the next iteration in the loop. If we can log in to the device, the code executes the lines in the **try** statement.

```
try:
......
except:
print(f"***Cannot login to {ip['host']}")
```

5. In the **try** statement, we connect the device with the **Netmiko** function and execute the command; we save all logs to the **output** variable. So before connecting to the device, we add a **print** function as **Try to**

Login with the IP address of the current iteration. Hence, we can see the process in the output, and if we have multiple devices to log in, it's better to give this information in order to track the process.

```
print(f"\n---Try to Login:{ip['host']}---\n")
net_connect = Netmiko(**ip)
output = net_connect.send_command(command)
print(output)
```

6. After connecting the device and running the commands, we are still inside the **try** statement. We collect the version, model, vendor, and hostname information from the **output** variable and use the **findall** function from the **RE** module. Then, we create four different variables and assign them to the target match.

   For example, one of Cisco ASR router's **show version** outputs is shown in *Table 4.2* (**www.ciscolive.com**). For the version information, we must match **5.3.3[Default]**, so we write the **findall** function with **Version (.*)**, and it reaches our target. For the model information, we write **Cisco (.*)\(revision**, for which the result is **ASR9K Series**. We write Cisco to match the vendor data. And finally, for the hostname, we write **(.*) uptime is** in the following output. The result is **Router-1**.

```
Router-1#show version
Cisco IOS XR Software, Version 5.3.3[Default]
Copyright (c) 2016 by Cisco Systems, Inc.
ROM: System Bootstrap, Version 10.45(c) 1994-2014 by
Cisco Systems, Inc.
Router-1 uptime is 0 weeks, 3 days, 7 hours, 38 minutes
System image file is "XXXX.vm"
Cisco ASR9K Series (revision …) processor with
```

Table 4.2: *Example output of the show version command in Cisco Device*

```
version = re.findall("Version (.*),", output)
model = re.findall("Cisco (.*)\(revision", output)
vendor = re.findall("Cisco", output)
hostname = re.findall("(.*) uptime is", output)
```

7. We collected the specific data in the previous code, and we can add it to a list. We already created five lists at the beginning of the code, so we use the **append** function in list variables to get the first item in the list as **item 0**.

```
ip_list.append(ip['host'])
version_list.append(version[0])
```

```
model_list.append(model[0])
vendor_list.append(vendor[0])
hostname_list.append(hostname[0])
```

8. Finally, we need to save these lists into an Excel file where we use the **DataFrame** function from the **pandas** module. We write a dictionary with keys as the strings and values as the lists we created; this dictionary is written inside the **DataFrame** function. After we assign this to the **df** variable, we call the **to_excel** function to write the collected list variables to the Excel file. We can set the Excel file name and sheet name and also the index with this function, but we don't use it in this example

```
df = DataFrame({"IP Address": ip_list, "Hostname":
hostname_list, "Vendor Type": vendor_list, "Model":
model_list, "Version": version_list})
df.to_excel("Version List.xlsx", sheet_name="Vendors",
index=False)
```

Example 4.9: Collecting device information and saving it in an Excel file

```
from netmiko import Netmiko
import re
from pandas import DataFrame

device1 = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
host = [device1, device2, device3]
command = "show version"

ip_list, version_list, model_list, vendor_list, hostname_list =
([] for i in range(5))

for ip in host:

  try:
   print(f"\n---Try to Login:{ip['host']}---\n")
   net_connect = Netmiko(**ip)
   output = net_connect.send_command(command)
   print(output)
   version = re.findall("Version (.*),", output)
```

```
        model = re.findall("Cisco (.*)\(revision", output)
        vendor = re.findall("Cisco", output)
        hostname = re.findall("(.*) uptime is", output)

        ip_list.append(ip['host'])
        version_list.append(version[0])
        model_list.append(model[0])
        vendor_list.append(vendor[0])
        hostname_list.append(hostname[0])

    except:
        print(f"***Cannot login to {ip['host']}")
df = DataFrame({"IP Address": ip_list, "Hostname": hostname_list,
"Vendor Type": vendor_list, "Model": model_list, "Version":
version_list})
df.to_excel("Version List.xlsx", sheet_name="Vendors",
index=False)
```

When we execute *Example 4.9*, the code creates an Excel file. When we open it, we see that all device information is filled in the Excel file, as illustrated in *Figure 4.4*:

	A	B	C	D	E
1	IP Address	Hostname	Vendor Type	Model	Version
2	10.10.10.1	Router-1	Cisco	ASR9K Series	Version 5.3.3[Default]
3	10.10.10.2	Router-2	Cisco	ASR9K Series	Version 5.3.3[Default]
4	10.10.10.3	Router-3	Cisco	ASR9K Series	Version 5.3.3[Default]

Figure 4.4: *Output of Example 4.9*

Collecting CPU levels

In *Example 4.10*, we will try to find the CPU levels of the Cisco devices in 5-second, in 1-minute, and in 5-minute values. When we run **`show processes CPU`** in the Cisco command line, there is a line in the output: **`CPU utilization for five seconds: 19%/0%; one minute: 20%; five minutes: 16%`**. It shows all the CPU values as we try to collect them. So in this code, we will try to find CPU data from the output.

1. After we import the modules, we create a **`command`** variable, assign the **`show processes CPU`** string, and create five empty lists to use in the following code:

```
from netmiko import Netmiko
import re
from pandas import DataFrame
device1 = {"host": "10.10.10.1", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
host = [device1, device2, device3]
command = "show processes cpu"
ip_list, cpu_list_5s, cpu_list_1m, cpu_list_5m, cpu_list_risk
= ([] for x in range(5))
```

2. We create a **for** loop again, write the **try…except** statement, and write the following codes inside the **try** statement:

```
for ip in host:
  try:
  …..
  except:
   print(f"***Cannot Login to {ip['host']}")
```

3. We log in to the device and execute the command. After that, we assign all device logs to the **output** variable as a string:

```
   print(f"\n---Try to Login:{ip['host']}---\n")
   net_connect = Netmiko(**ip)
   output = net_connect.send_command(command)
```

4. We try to find 5 seconds, 1 minute, and 5 minutes of CPU levels of the device with the **findall** function in the following code:

```
   cpu_5s = re.findall("CPU utilization for five seconds:
   (\d+)",output)
   cpu_1m = re.findall("one minute: (\d+)",output)
   cpu_5m = re.findall("five minutes: (\d+)",output)
```

5. After we find the specific data, we need to append it to the empty lists, which we created at the beginning of the code. We will collect the data as digits, so it's better to add the **%** character at the end of the CPU level, like

in the following code. We will also append the IP address of each device. So in Excel, we can see which device has which CPU level.

```
ip_list.append(ip['host'])
cpu_list_5s.append(cpu_5s[0]+"%")
cpu_list_1m.append(cpu_1m[0] + "%")
cpu_list_5m.append(cpu_5m[0] + "%")
```

6. Additionally, we can add the **if** condition to the code and alert the user if the CPU usage is higher than 90% with the message **Fatal CPU Level**. If it's between 70 and 90, we can alert them with the message **High CPU Level**, or we can inform them with the message **No Risk**. So we use the if condition and convert the string to an integer with the **int()**. We can add the risk value to the **cpu_list_risk** variable.

```
if int(cpu_5m[0]) > 90:
cpu_risk = "Fatal CPU Level"
elif 70< int(cpu_5m[0]) <90:
cpu_risk = "High CPU Level"
else:
cpu_risk = "No Risk"
cpu_list_risk.append(cpu_risk)
```

7. Finally, as we did in *Example 4.9*, we will create a dictionary with the **DataFrame** function and save all the output to the Excel file.

```
df=DataFrame({"IP Address":ip_list,"CPU Levels for 5
Seconds": cpu_list_5s,"CPU Levels for 1 Minute":cpu_list_1m,
"CPU Levels for 5 Minutes":cpu_list_5m,"CPU
Risk":cpu_list_risk})
df.to_excel("CPU Levels.xlsx",index=False)
```

Example 4.10: Collecting CPU levels

```
from netmiko import Netmiko
import re
from pandas import DataFrame
device1 = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
host = [device1, device2, device3]
```

```
command = "show processes cpu"
ip_list, cpu_list_5s, cpu_list_1m, cpu_list_5m, cpu_list_risk =
([] for x in range(5))
for ip in host:
  try:
   print(f"\n---Try to Login:{ip['host']}---\n")
   net_connect = Netmiko(**ip)
   output = net_connect.send_command(command)

   cpu_5s = re.findall("CPU utilization for five seconds:
   (\d+)",output)
   cpu_1m = re.findall("one minute: (\d+)",output)
   cpu_5m = re.findall("five minutes: (\d+)",output)

   ip_list.append(ip['host'])
   cpu_list_5s.append(cpu_5s[0]+"%")
   cpu_list_1m.append(cpu_1m[0] + "%")
   cpu_list_5m.append(cpu_5m[0] + "%")

   if int(cpu_5m[0]) > 90:
    cpu_risk = "Fatal CPU Level"
   elif 70< int(cpu_5m[0]) <90:
    cpu_risk = "High CPU Level"
   else:
    cpu_risk = "No Risk"

   cpu_list_risk.append(cpu_risk)
   df=DataFrame({"IP Address":ip_list,"CPU Levels for 5 Seconds":
   cpu_list_5s, "CPU Levels for 1 Minute":cpu_list_1m, "CPU
   Levels for 5 Minutes":cpu_list_5m,"CPU Risk":cpu_list_risk})
   df.to_excel("CPU Levels.xlsx",index=False)
  except:
   print(f"***Cannot Login to {ip['host']}")
```

When we execute *Example 4.10*, the code creates an Excel file. When we open it, we can see that all the device CPU information in 5 seconds, 1 minute, and 5 minutes is filled in the Excel file illustrated in *Figure 4.5*:

	A	B	C	D	E
1	IP Address	CPU Levels for 5 Seconds	CPU Levels for 1 Minute	CPU Levels for 5 Minutes	CPU Risk
2	10.10.10.1	12%	12%	13%	No Risk
3	10.10.10.2	14%	12%	13%	No Risk
4	10.10.10.3	10%	12%	13%	No Risk

Figure 4.5: *Output of Example 4.10*

Finding duplicated IP address

In *Example 4.11*, we will try to find the duplicated IP addresses in our network. We have three devices to check an IP address, and we log in to each one to search for the target IP address. If we find a duplicated IP address, the code gives the output of the duplicated IP, duplicated host IP, and the interface information of the duplicated IP address.

1. We import **`netmiko`** and **`re`** modules. After that, we add three devices' information as a **`host`** list. We create a variable called **`check_ip`**. We add the target IP address that we are looking for in the network, and we create an empty list that we will use later in the code. Finally, we create the **`command`** variable./p>

```
from netmiko import Netmiko
import re
device1 = {"host": "10.10.10.1", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
host = [device1, device2, device3]
check_ip = "10.10.10.2"
duplicated_list = []
command = "show ip interface brief"
```

2. We create a **`for`** loop for all devices in the network. After that, we write our code inside the **`try`** statement. We will continue with the following code if we can log in to the device./p>

```
for ip in host:
```

```
print(f"\n---Try to Login: {ip['host']} ---\n")
try:
.........
except:
 print(f"***Cannot login to {ip['host']}")
```

3. We connect to the device and collect the logs. We search for the target IP address inside the logs with the **findall** function and assign it to the **duplicate_ip** variable.

```
net_connect = Netmiko(**ip)
output =net_connect.send_command(command)
duplicate_ip = re.findall(check_ip,output)
```

4. If the **duplicate_ip** variable is empty, we have no duplicate IP address in the network. Otherwise, we have a duplicated IP address. We use the **while** statement in the following code. If the **duplicate_ip** variable is not empty, continue with the **while** statement.
5. So, if we have a match of the IP address in the network, the **while** statement is accurate, and the code executes the following codes. First, it tries to find the interface information with the **(.*){check_ip}** match. Afterward, it appends the target IP address to the **duplicated_list** variable. Finally, it assigns the duplicated device IP address to the **duplicate_device** variable and finishes the **while** loop with the **break** statement.

```
while duplicate_ip:
 interface = re.findall(f"(.*){check_ip}",output)
 duplicated_list.append(check_ip)
 duplicate_device = ip["host"]
 break
```

6. Finally, if the **duplicated_list** variable is not empty or has an item inside, continue with the **print** function by writing the duplicated IP, the duplicated device's IP address, and the interface of the duplicated IP address. Otherwise, it prints that the target IP address is not duplicated in the network and can be used in the network without a problem.

```
if duplicated_list:
  print(f"---------\nDuplicated IP: {check_ip} \nDuplicated
  Device IP Address: {duplicate_device} \nInterface:
  {interface[0]} \n -----------")
else:
  print(f"{check_ip} IP address is suitable for use")
```

Example 4.11: Finding duplicated IP address

```
from netmiko import Netmiko
import re

device1 = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device2 = {"host": "10.10.10.2", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
device3 = {"host": "10.10.10.3", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1}
host = [device1, device2, device3]
check_ip = "10.10.10.2"
duplicated_list = []
command = "show ip interface brief"

for ip in host:
  print(f"\n---Try to Login: {ip['host']} ---\n")
  try:
   net_connect = Netmiko(**ip)
   output =net_connect.send_command(command)
   duplicate_ip = re.findall(check_ip,output)

   while duplicate_ip:
    interface = re.findall(f"(.*){check_ip}",output)
    duplicated_list.append(check_ip)
    duplicate_device = ip["host"]
    break
  except:
   print(f"***Cannot login to {ip['host']}")

if duplicated_list:
  print(f"---------\nDuplicated IP: {check_ip} \nDuplicated Device
  IP Address: {duplicate_device} \nInterface: {interface[0]} \n --
  ---------")
else:
  print(f"{check_ip} IP address is suitable for use")
```

Collecting logs with multithreading

When we log in to devices, we always use the **for** loops. Code connects to the devices in each for loop one by one. If we have 100 devices, the loop executes

100 times by order, and if one device connection and collect log takes 30 seconds, the total time to collect all devices' data is 3000 seconds or 50 minutes.

It's too much time to collect data from many devices. We need a solution to connect all devices at the same time and collect the data simultaneously.

We can use several options to solve this issue. In *Example 4.12*, we use the multithreading module of the Python language. So, if we have 100 devices, we can collect data on all devices in 30 seconds instead of 50 minutes. Multithreading is a powerful feature of Python that executes the code on all devices simultaneously. In the programming language, multithreading can also be called **parallelism**.

In *Example 4.12*, we will use the `paramiko` module to connect devices. We create two text files to get the device's IP addresses and the command lists.

1. We import the `paramiko`, `time`, `re`, and `threading` modules to execute our code:/p>

```
import paramiko
from time import sleep
import re
import threading
```

2. We create two functions: `SSH_Thread()` and `ssh_conn()`.We open the **ip_list.txt** file in *Table 4.3* to read it. We divide each IP address with lines and create a `host` variable as a list. Each item on that list is an IP address, so we have a list that includes IP addresses.

ip_list.txt
10.10.10.1
10.10.10.2
10.10.10.3

Table 4.3: *Content of the "ip_list.txt" file*

```
def SSH_Thread():
  with open("ip_list.txt") as r:
   host = r.read()
  host = re.split("\n", host)
```

3. Inside the `for` loop, we call the `Thread` function from the `threading` module and assign it to the `trd` variable. Inside this function, we have two parameters: `target` and `args`. The `target` parameter is the callable object to be invoked.

In this example, we call the **`ssh_conn`** function, and **`args`** is the argument tuple for the target invocation. In this example, we write ip as the **`for`** loop variable. We must write **`args=(ip,)`** the variable and comma after, otherwise the program gives an error. We need to call the **`start`** function to start the thread's activity.

```
for ip in host:
 trd = threading.Thread(target=ssh_conn, args=(ip,))
 trd.start()
```

4. The first function is finished, and threading is done in that function. Now, we can execute the commands inside the devices and save them to text files. We create a **`command_list`** text file in *Table 4.4* and read it. We split each command by line.

```
command_list.txt
show ip interface brief
show clock
show arp
show ip route
```

Table 4.4: *Content of the "command_list.txt" file*

```
def ssh_conn(ip):
 with open("command_list.txt") as c:
  command_list = c.read()
 command_list = re.split("\n", command_list)
```

5. We write the paramiko connection functions: IP as the **`ip`** variable, and username and password. We call the **`ssh_conn`** function with a parameter as the **`ip`**, so threading connects each device at the same time with this parameter. As we use the **`ip`** parameter in the **`connect`** function of paramiko to connect to the devices with the IP address, we create an empty **`result`** string variable to use later in the code./p>

```
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy()
)
client.connect(ip, 22, "admin", "cisco")
commands = client.invoke_shell()
result = ""
```

6. We create a **`for`** loop to execute all the commands in the device. After that, we add some delays by the **`sleep`** function delays to get all the

output of the command. As there is no mechanism to wait until the output is finished in paramiko, we save it to the **result** variable.

```
for comm in command_list:
 commands.send(f"{comm} \n")
 sleep(1.5)
 output = commands.recv(1000000).decode("utf-8").replace
 ("\r", "")
 result += str(output)
 print(output)
```

7. We save the output of the **result** variable in a log file by naming the device's IP address.

```
with open(f"{ip}.log", "a") as wr:
 wr.write(result)
```

8. We must call a function to execute in the code. So, we call the **SSH_Thread** function to execute the code.

```
SSH_Thread()
```

Example 4.12: Collecting logs with multithreading

```
import paramiko
from time import sleep
import re
import threading

def SSH_Thread():
  with open("ip_list.txt") as r:
    host = r.read()
  host = re.split("\n", host)

for ip in host:
    trd = threading.Thread(target=ssh_conn, args=(ip,))
    trd.start()

def ssh_conn(ip):
  with open("command_list.txt") as c:
    command_list = c.read()
  command_list = re.split("\n", command_list)

  client = paramiko.SSHClient()
  client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
  client.connect(ip, 22, "admin", "cisco")
  commands = client.invoke_shell()
```

```
  result = ""

  for comm in command_list:
   commands.send(f"{comm} \n")
   sleep(1.5)
   output = commands.recv(1000000).decode("utf-8").replace("\r",
   "")
   result += str(output)
   print(output)

  with open(f"{ip}.log", "a") as wr:
   wr.write(result)
SSH_Thread()
```

Tools and calculators

We can create custom tools with Python scripts. We will write two examples in this section: *IP address validator* and *Subnet calculator*. We can check whether an IP address is a valid IPv4 address. We can also calculate parameters such as the subnet, network and broadcast address, and all available hosts in the subnet that the user enters.

IP address validator

In *Example 4.13*, we create a simple tool to verify or validate an IPv4 address. In this script, we ask the user to enter an IP address and check whether it is valid or invalid.

1. We ask the user to enter an input with input function.

   ```
   enter_ip = input("\nEnter an IP address: ")
   ```

2. We split the input string with dividing dots. It creates a new list variable as the **`ip`**. We create an integer variable as **`valid`** that we use in the **`for`** loop.

   ```
   Ip = enter_ip.split(".")
   valid = 0
   ```

3. The structure of an IPv4 address is X.X.X.X, with four numbers from 0 to 255 and 3 dots between each number. So, the input cannot have any alphabetic characters or special characters, and all inputs must be digits with three dots. Otherwise, it cannot be a valid IPv4 address.

We create an **if** statement to find whether the input has four numbers divided by commas. If it doesn't match the condition, the code gives an output saying it's an invalid IP address. If it matches, we continue with the body of the **if** statement.

```
if len(ip) == 4:
…..
else:
  print ("This is NOT a VALID IP Address")
```

4. If we have four items in the list and can fill the **if** statement. We write a **for** loop. So, we can check all four items one-by-one. Each item must be between 0 and 256, including 0, but a user can enter a non-digit character. To check it, we can write **int(x)**.

   This code automatically converts a string to an integer if all the characters are digits. Otherwise, it throws an error.

   So with this code, we can catch digits. However, when the user enters a non-digit character, we try to continue without error, so we use the **try… except** statement. If the code gives an error in the **try** statement, the code directly continues with the **except** statement. In the except statement, an error message is shown because the user input has a non-digit character.

   Inside the **if** condition, we add the **valid** variable as 1. If the integer of the current iteration or **x** is not between **0** and **256**, the code gives an error. If all four items match the condition, the **valid** value equals four after the loop finishes, code continue. It displays the IPv4 address as a valid IP address.

   We also use the **break** statement in the **for** loop. If one of the **x** values is invalid, the code exits from the **for** loop after giving the **not valid** error message.

```
  try:
    for x in ip:
      if 0 <= int(x) < 256:
        valid = valid + 1
      else:
        print("This is NOT a VALID IP Address")
        break
    if valid == 4:
      print(f"{enter_ip} is a VALID IP Address")
```

```
    except:
     print ("This is NOT a VALID IP Address")
```

Example 4.13: Creating an IP address validator

```
enter_ip = input("\nEnter an IP address: ")
ip = enter_ip.split(".")
valid = 0
if len(ip) == 4:
  try:
   for x in ip:
    if 0 <= int(x) < 256:
     valid = valid + 1
    else:
     print("This is NOT a VALID IP Address")
     break
   if valid == 4:
    print(f"{enter_ip} is a VALID IP Address")
  except:
   print ("This is NOT a VALID IP Address")
else:
  print ("This is NOT a VALID IP Address")
```

Subnet calculator

In *Example 4.14*, we try to create our subnet calculator. We input the IP address and the subnet mask, and the code gives the subnet, wildcard, total available host in that subnet, network and broadcast address, and finally, the IP address range of this subnet.

1. We enter the IP address and create an empty list. Code divides the string with dots and adds each string to the **ip** variable.
   ```
   enter_ip = input("\nEnter an IP address: ")
   octet_list = []
   ip = enter_ip.split(".")
   ```
2. We create a `for` loop to check all octets, regardless of whether or not they are digits. We must be sure that there are no non-digit characters inside all items in the `ip` variable. Code converts each item from string to an integer, then adds to the `octet_list`. If it fails, it executes the `continue`

statement, and the following code gives an error as the **`Invalid IP Address`**.

```
for octet in ip:
  try:
   octet_list.append(int(octet))
  except:
   continue
```

3. An IPv4 address must have four parts and all the parts must be between `0` and `255`. For example, `192.168.1.1` is a valid IPv4 address, but `192.260.2.3` is not, because the second part is bigger than 255. If the condition is not matched, the code throws an eror saying invalid IP address.

```
if len(octet_list) == 4 and 0 < octet_list[0] < 255 and  0 <=
octet_list[1] <= 255 and  0 <= octet_list[2] <= 255  and  0
<= octet_list[3] <= 255 :
….
else:
  print("ERROR: INVALID IP ADDRESS")
```

4. Inside the `if` condition, we ask for second input as the subnet mask. The subnet mask must be from `1` to `32`. Otherwise, the code gives an error. We use the `try`…`except` statement to check it./p>

```
  mask = input("\nEnter a Subnet Mask (1 to 32):  address: ")
  try:
  …….
  except:
   print("ERROR: INVALID IP ADDRESS")
```

5. Inside the `try` statement, we convert the input to an integer. If the condition does not match, the code finishes and gives an invalid IP address.

```
    number = int(mask)
    if  0 < number <= 32:
    ……
    else:
     print("ERROR: INVALID IP ADDRESS")
```

6. We are inside the main code of calculation of the parameters. First, we need to find the main subnet classes of the input mask 0, 8, 16, and 24.

```
      a = int(int(mask) / 8)
```

7. After that, we need to find the subclasses of the input mask, like 18, 25, and 26.

```
b = int(mask) % 8
octet1 = 2 ** 8 - 2 ** (8 - b)
```

8. We need to find the network and broadcast addresses with the following calculation. After that, we can find the minimum available host by adding 1 to a network address, and we can find the maximum available host by deleting 1 from the broadcast address.

```
z = octet_list[a]        #Find  the octet to change
k = int(z / 2 ** (8 - b))
net = ((2 ** (8 - b)) * k)  #network address calculation
brod = ((2 ** (8 - b)) * (k + 1)) - 1  #Broadcast address
calculation
min_host = net + 1       #Min avaliable host
max_host = brod - 1      #Max avaliable host
```

9. After that, we need to find the subclass value. We had the `if` condition in the following part. We check the `a` variable value from `0` to `3`. If a equals `0`, the subnet is `X.0.0.0`. `X` must be an integer between `0` and `255`. If a equals 1, the subnet is `255.X.0.0`. We also write the wildcard with the `y` string. When we find the subnet, we can find other parameters easily. The calculation is related to network address calculation.

```
if a == 0:
 subnet = "x.0.0.0"
 wildcard = "y.255.255.255"
 total_host = ((256-octet1)*(256**3))-2
 network = "{}.{}.{}.{}".format(net,0,0,1)
 broadcast = "{}.{}.{}.{}".format(brod,255,255,255)
 min_host = "{}.{}.{}.{}".format(net,0,0,2)
 max_host = "{}.{}.{}.{}".format(brod,255,255,254)
elif a == 1:
 subnet = "255.x.0.0"
 wildcard = "0.y.255.255"
 total_host = ((256-octet1)*(256**2))-2
network = "{}.{}.{}.{}".format(octet_list[0], net,0,1)
broadcast = "{}.{}.{}.{}".format(octet_list[0],
brod,255,255)
min_host = "{}.{}.{}.{}".format(octet_list[0], net,0,2)
```

```
max_host = "{}.{}.{}.{}".format(octet_list[0],
brod,255,254)
elif a == 2:
 subnet = "255.255.x.0"
 wildcard = "0.0.y.255"
 total_host = ((256-octet1)*256)-2
network = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1],net,1)
broadcast = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1],brod,255)
min_host = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1],net,2)
max_host = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1],brod,254)
elif a == 3:
 subnet = "255.255.255.x"
 wildcard = "0.0.0.y"
 total_host = (256-octet1)-2
network = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1], octet_list[2],net)
broadcast = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1], octet_list[2],brod)
min_host = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1], octet_list[2],min_host)
max_host = "{}.{}.{}.{}".format(octet_list[0],
octet_list[1], octet_list[2],max_host)
```

10. Then, we replace `x` and `y` with `octet1` and `255-octet1` in which the value is subclass value.

```
subnet_new = subnet.replace("x", str(octet1))
wildcard = wildcard.replace("y", str(255 - octet1))
```

11. After finding all the information, we print it, as shown in the following code:

```
print("-------------\nIP Address: {}".format(enter_ip))
print("Subnet Mask: {}".format(mask))
print("Subnet: {}".format(subnet_new))
print("Wildcard: {}".format(wildcard))
print("Total Host: {}".format(total_host))
print("Network Address: {}".format(network))
```

```
        print("Broadcast Address: {}".format(broadcast))
        print("IP Address Range: {} - {}".format(min_host,
        max_host) )
```

Example 4.14: Subnet calculator

```
enter_ip = input("\nEnter an IP address: ")
octet_list = []
ip = enter_ip.split(".")          #Divide ip address to octets by
"." dot character
for octet in ip:                  #Check all octets are digits (not
contain any non-digit character)
  try:
   octet_list.append(int(octet))    #Convert each item in list to
   integer, if fail, continue
  except:               #So if fail,octet list will not be 4
  anymore.
  continue              #And below if condition will not be matched
if len(octet_list) == 4 and 0 < octet_list[0] < 255 and  0 <=
octet_list[1] <= 255 and  0 <= octet_list[2] <= 255  and  0 <=
octet_list[3] <= 255 :
  mask = input("\nEnter a Subnet Mask (1 to 32):  address: ")
  try:
  number = int(mask)  #Convert input to integer, if fail, continue
   if  0 < number <= 32:
   # Find 0/8/16/24 main classes
    a = int(int(mask) / 8)
   # Find sub-class like 18,25,26, etc.
    b = int(mask) % 8
   octet1 = 2 ** 8 - 2 ** (8 - b)  #Find subclass value
   z = octet_list[a]       #Find  the octet to change
    k = int(z / 2 ** (8 - b))
   net = ((2 ** (8 - b)) * k)  #network address calculation
    brod = ((2 ** (8 - b)) * (k + 1)) - 1   #Broadcast address
    calculation
   min_host = net + 1      #Min avaliable host
   max_host = brod - 1     #Max avaliable host
   # Find subclass value
    if a == 0:
     subnet = "x.0.0.0"
```

```
 wildcard = "y.255.255.255"
 total_host = ((256-octet1)*(256**3))-2
 network = "{}.{}.{}.{}".format(net,0,0,1)
 broadcast = "{}.{}.{}.{}".format(brod,255,255,255)
 min_host = "{}.{}.{}.{}".format(net,0,0,2)
 max_host = "{}.{}.{}.{}".format(brod,255,255,254)
elif a == 1:
 subnet = "255.x.0.0"
 wildcard = "0.y.255.255"
 total_host = ((256-octet1)*(256**2))-2
 network = "{}.{}.{}.{}".format(octet_list[0],net,0,1)
 broadcast = "{}.{}.{}.{}".format(octet_list[0],brod,255,255)
 min_host = "{}.{}.{}.{}".format(octet_list[0],net,0,2)
 max_host = "{}.{}.{}.{}".format(octet_list[0],brod,255,254)
elif a == 2:
 subnet = "255.255.x.0"
 wildcard = "0.0.y.255"
 total_host = ((256-octet1)*256)-2
  network = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],net,1)
  broadcast = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],brod,255)
  min_host = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],net,2)
  max_host = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],brod,254)
elif a == 3:
 subnet = "255.255.255.x"
 wildcard = "0.0.0.y"
 total_host = (256-octet1)-2
  network = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],octet_list[2],net)
  broadcast = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],octet_list[2],brod)
  min_host = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],octet_list[2],min_h
  ost)
  max_host = "{}.{}.{}.
  {}".format(octet_list[0],octet_list[1],octet_list[2],max_h
```

```
    ost)
  subnet_new = subnet.replace("x", str(octet1))  #Replace x
  value in subnet with octet1
  wildcard = wildcard.replace("y", str(255 - octet1))
  print("-------------\nIP Address: {}".format(enter_ip))
  print("Subnet Mask: {}".format(mask))
  print("Subnet: {}".format(subnet_new))
  print("Wildcard: {}".format(wildcard))
  print("Total Host: {}".format(total_host))
  print("Network Address: {}".format(network))
  print("Broadcast Address: {}".format(broadcast))
  print("IP Address Range: {} - {}".format(min_host,max_host) )
 else:
  print("ERROR: INVALID IP ADDRESS")
except:             # So if fail,octet list will not be 4
anymore.
 print("ERROR: INVALID IP ADDRESS")
else:
  print("ERROR: INVALID IP ADDRESS")
```

Conclusion

In this chapter, we learnt about connection modules. For SSH, we use netmiko and paramiko, and for telnet, we use the telnetlib and netmiko modules. We connected network devices and collected logs for specific purposes like collecting software versions and CPU levels of devices, logged in to multiple devices simultaneously with multithreading and created an IP validation tool and a subnet calculator.

In the next chapter, we will focus on configuring network devices. We will be configuring interfaces, SNMP, and OSPF protocols, and we will replace the old configuration with the new configuration parameters. We will be saving the multiple devices' configuration with a single script.

Multiple choice questions

1. Which module can log in with telnet protocol?
   a. netmiko module
   b. paramiko module

c. os module
d. re module

2. What is the **`device_type`** parameter to log in to Cisco devices by the netmiko module?
    a. cisco
    b. csc_ios
    c. cisco_ios
    d. cisco_systems
3. Which module is used to connect to multiple devices simultaneously?
    a. Import os
    b. Import telnetlib
    c. Import multithreading
    d. Import threading

Answers

1. a
2. c
3. d

Questions

1. Write a script to collect the ARP table from the Cisco device and write it to different text files for each device with the **`show arp`** command.
2. Write a script to collect logs from the Cisco device using multithreading and netmiko module.

CHAPTER 5
Deploy Configurations in Network Devices

This chapter will focus on the configuration of network devices with different modules and functions. We will use the jinja2 template, YAML files, NAPALM module, and nornir automation framework. As advanced usage, we will use these modules and templates to configure multiple devices in a more automated way.

Structure

In this chapter, we will cover the following topics:

- Configure network devices
    - Configuration of interfaces
    - Replacing configurations on files
- Configure devices with Jinja2 template
    - Introduction to Jinja2 template
    - Introduction to YAML language
    - Rendering Jinja template with the YAML file
    - Configure devices with Jinja
    - If statement in Jinja
- Configure devices with Napalm module
    - Collect logs from devices with NAPALM
    - Configure devices with NAPALM
- Configure devices with Nornir module
    - Configure inventory in Nornir

- Connection to devices with Nornir-Netmiko
- Connection to devices with Nornir-NAPALM
- Configure devices by Nornir and Jinja Template

Objectives

With the knowledge of previous chapters, we can easily log in to devices and configure them with the **`netmiko`** and **`paramiko`** modules. In basic usage, we can handle easy tasks. When we need to take up more complex tasks, we must use some modules or frameworks to handle these by adding automation. We use configuration templates like Jinja to configure multiple devices with fewer lines of code. We use the NAPALM module to connect devices in a more straightforward mode. We create a nornir automation platform to develop our automation scripts with faster connection types.

Configure network devices

As we did in the previous chapter, we can create automation scripts for collecting data from any network or system device. We can also modify, implement and configure those devices with Python scripts. We can deploy 10 or even 100 devices with a simple script. We use the netmiko module, which performs better than the **`paramiko`** module, to deploy configurations in network devices. As we did in the previous chapter, there is always an option to use multithreading to configure devices in parallel.

There are different options to create data for configuration. We can directly write the command set inside the script, which is the basic usage of scripting. But for advanced use, scripts must be more flexible. To do that, we can write the commands in another text file and get all the commands from that file. So, our code will be much more apparent. We can also create an Excel file and get the data from there. We have the option to get different data for different devices. For example, we can configure each interface with the different IP addresses on different devices. IP addresses cannot be identical in the network, so they must be unique. For different purposes, we will create various kinds of scripts.

1. In *Example 5.1*, we try to log in to devices with the **`netmiko`** module again. For netmiko, we have a format to add devices, and we need to

add a unique IP address. However, the username, password, delay, or device models can be identical or unique. We enter too many parameters and many lines in the examples. We can create a **for** loop for those parameters to avoid too many repeatable codes.

We import the **netmiko** module and then write our code. In the previous examples of the **netmiko** module, we always added devices with different variables: **device1**, **device2** and **device3**. In those examples, it seemed to be no problem. But if we have 100 devices, adding each variable is not good. So, we can create a loop for this repeatable code. We create a variable named **ip_list** and fill it with the device IP addresses.

```
from netmiko import Netmiko
ip_list = [ "10.10.10.1", "10.10.10.2", "10.10.10.3",
"10.10.10.4"]
```

2. We create a **for** loop to get the IP addresses from the **ip_list** variable. We add the **ip** variable as the value of the **host** key. So, in each iteration, the value of the host key changes according to the **ip_list** items. It means that, in each loop, we have a different device to log in.

```
for ip in ip_list:
  ip = {
   "host": f"{ip}",
   "username":"admin",
   "password":"cisco",
   "device_type": "cisco_ios",
   "global_delay_factor": 0.1
  }
```

We can also change other parameters. For example, if we have Cisco, Juniper and Huawei devices in our network, we need to change the **device_type** parameter for different vendors. So, we can create another variable by adding the vendor names and creating an **if** condition in the loop to choose the specific vendor. We can use the same logic for the username and password.

3. As we did in the previous examples, we write our main code in the **try…except** statement. In the example, we have a **10.10.10.4** device that we cannot log in. So, when we execute the code, it gives an error

that we cannot log in. Inside this statement, we write our netmiko code, and we send the **show** command to the device and get the output.

```
try:
 print(f"\n---Try to Login: {ip['host']} ---\n")
 net_connect = Netmiko(**ip)
 output = net_connect.send_command("show interface
 description")
 print(output)
except:
 print(f"***Cannot login to {ip['host']}")
```

Example 5.1: Creating device information in the loop

```
from netmiko import Netmiko
ip_list = [ "10.10.10.1", "10.10.10.2", "10.10.10.3",
"10.10.10.4" ]
for ip in ip_list:
  ip = {
   "host": f"{ip}",
   "username":"admin",
   "password":"cisco",
   "device_type": "cisco_ios",
   "global_delay_factor": 0.1
  }
  try:
   print(f"\n---Try to Login: {ip['host']} ---\n")
   net_connect = Netmiko(**ip)
   output = net_connect.send_command("show interface
   description")
   print(output)
  except:
   print(f"***Cannot login to {ip['host']}")
```

Instead of creating a variable and calling it in the loop, we can create a text file and add all the IP information there. So, we can open the text file and add all lines as different items in a variable.

In *Example 5.2*, we import the **re** module. We open the **host_info** text file and read it. After that, we create the **ip_list** variable and get each line as a unique item in this variable.

```
host_info.txt
10.10.10.1
10.10.10.2
10.10.10.3
```

10.10.10.4

```
import re
with open("host_info.txt") as r:
  host = r.read()
ip_list = re.split("\n", host)
```

The remaining parts of this example are the same as *Example 5.1*. So, if we have many devices, it's better to use this method to get the IP information.

Example 5.2: Getting device information from text file

```
from netmiko import Netmiko
import re
with open("host_info.txt") as r:
  host = r.read()
ip_list = re.split("\n", host)
for ip in ip_list:
  ip = {
   "host": f"{ip}",
   "username":"admin",
   "password":"cisco",
   "device_type": "cisco_ios",
   "global_delay_factor": 0.1
  }
  try:
   print(f"\n---Try to Login: {ip['host']} ---\n")
   net_connect = Netmiko(**ip)
   output = net_connect.send_command("show interface
   description")
   print(output)
  except:
   print(f"***Cannot login to {ip['host']}")
```

Configuration of interfaces

In *Example 5.3*, we create a netmiko script to execute commands to configure hostnames, OSPF, and interface configurations in Cisco routers. We collect all three-device data from the Excel file with the **`xlwings`** module. It's a third-party Python module that must be installed with the **`pip install xlwings`** command in the terminal. This module opens an Excel file. After that, it reads the data in all columns, writing the range **`A1:A14`** with the **`range`** function.

So, we add all Router-1 configurations in the A column, Router-2 is in B, and Router-3 is in C. We write a **`for`** loop to get each column range and device connection data from the **`netmiko`** module simultaneously. So, we create the **`for`** loop with two iterables. After that, we connect to the devices in each loop and send the **`configuration`** variable. The value in the **`configuration`** is a list of data in Excel from **`A1`** to **`A14`**.

	A	B	C
1	hostname Test-R1	hostname Test-R2	hostname Test-R3
2	interface GigabitEthernet0/1	interface GigabitEthernet0/1	interface GigabitEthernet0/1
3	ip address 20.20.20.1 255.255.255.0	ip address 20.20.20.2 255.255.255.0	ip address 20.20.20.3 255.255.255.0
4	no shutdown	no shutdown	no shutdown
5	interface GigabitEthernet0/2	interface GigabitEthernet0/2	interface GigabitEthernet0/2
6	ip address 30.30.30.1 255.255.255.0	ip address 30.30.30.2 255.255.255.0	ip address 30.30.30.3 255.255.255.0
7	no shutdown	no shutdown	no shutdown
8	interface GigabitEthernet0/3	interface GigabitEthernet0/3	interface GigabitEthernet0/3
9	ip address 40.40.40.1 255.255.255.0	ip address 40.40.40.2 255.255.255.0	ip address 40.40.40.3 255.255.255.0
10	no shutdown	no shutdown	no shutdown
11	router ospf 10	router ospf 10	router ospf 10
12	network 20.20.20.0 0.0.0.255 area 0	network 20.20.20.0 0.0.0.255 area 0	network 20.20.20.0 0.0.0.255 area 0
13	network 30.30.30.0	network 30.30.30.0	network 30.30.30.0

	0.0.0.255 area 0	0.0.0.255 area 0	0.0.0.255 area 0
14	network 40.40.40.40 0.0.0.255 area 0	network 40.40.40.40 0.0.0.255 area 0	network 40.40.40.40 0.0.0.255 area 0

Table 5.1: *Configuration template of an Excel file "Config_file.xlsx"*

In *Table 5.1*, all three device configurations are saved in the **Config_file.xlsx** Excel file with the same **script** directory. When we execute the script in *Example 5.3*, it configures each device with the specific template in columns A, B, and C. With an Excel template, we can configure many devices with the first installation in a simple solution.

Example 5.3: Deploy configuration template from the Excel file

```
import xlwings
from netmiko import Netmiko
excel = xlwings.Book("Config_file.xlsx").sheets['Sheet1']
column = ["A","B","C"]
host = ["10.10.10.1", "10.10.10.2", "10.10.10.3"]
for x,ip in zip(column,host):
  print(f"---Connected to {ip}---")
  configuration = excel.range(f"{x}1:{x}14").value
  device = {"host": ip, "username": "admin", "password":
  "cisco","device_type": "cisco_ios"}
  net_connect = Netmiko(**device)
  net_connect.send_config_set(configuration)
```

In the output of *Example 5.3*, in each connection, we display the IP address with a **print** function. So, we can understand which device has a connection now. So, if we have 100 devices, we can see how many device configurations are finished and how many of them remain.

Netmiko is designed to connect network devices more smartly. So, when we send configuration with the **send_config_set** function, it automatically enters the configuration mode, and after all commands finish, it automatically exits to user mode. In this example, it's Cisco routers, so it enters configuration mode with the **configure terminal** and exists with the **end** command. We have no need to enter those commands, but if we use the **paramiko** module, we need to enter them.

In *Example 5.4*, we get the IP address and assigned it to the **ip_list** variable. This time, we call the commands from a text file. We call the

send_config_from_file function to run all content in the file. We write the filename with its extension as a string inside parentheses.

```
output =
net_connect.send_config_from_file("command_list.txt")
```

So, the code is much clearer to handle, and the script gets all the information to log the device in and executes commands from files.

Example 5.4: Configuration of interfaces from text files

```
from netmiko import Netmiko
import re
with open("host_info.txt") as r:
  host = r.read()
ip_list = re.split("\n", host)
for ip in ip_list:
  ip = {
   "host": f"{ip}",
   "username":"admin",
   "password":"cisco",
   "device_type": "cisco_ios",
   "global_delay_factor": 0.1 }
  try:
   print(f"\n---Try to Login: {ip['host']} ---\n")
   net_connect = Netmiko(**ip)
   output =
   net_connect.send_config_from_file("command_list.txt")
   print(output)
  except:
   print(f"***Cannot login to {ip['host']}")
```

For later usage of this code, we only need to change two files, and we do not need to change anything in the code.

host_info.txt 10.10.10.1 10.10.10.2 10.10.10.3	command_list.txt interface GigabitEthernet0/1 description Test no shutdown no cdp enable

Replacing configurations on files

In *Example 5.5*, we have a router's interface configuration text file. It could be a complete configuration to modify for further usage. In this scenario, we change the interface description, port **`shutdown`** status, and IP address information.

We remove the description line in the **`GigabitEthernet0/0`** interface in *the following code*. We also replace the **`no ip address`** line with the **`ip address 192.168.10.10 255.255.255.0`** line, and we remove the **`shutdown`** line under the **`GigabitEthernet0/1`**.

1. We create an **`old_config.txt`** text file with the following content. After that, we create a **`command_change`** variable. We write the old and new configurations in this variable. **`command_change[0]`** is the old value, and the **`command_change[1]`** is the new value. Also, **`command_change[2]`** is the old value, and **`command_change[3]`** is the new value. Even-numbered items represent old values, and odd-numbered items represent new values.

```
command_change = [
"""interface GigabitEthernet0/0
description TEST""",
"interface GigabitEthernet0/0",
"""interface GigabitEthernet0/1
no ip address
shutdown""",
"""interface GigabitEthernet0/1
ip address 192.168.10.10 255.255.255.0"""
]
```

2. We count the **`command_change`** lists items with the **`len`** function. In this example, the value of the **`item_count`** is four.

```
item_count = len(command_change)
```

3. We open the old configuration file with the **`open`** function, read it and assign it to the **`new_config`** variable.

```
with open("old_config.txt") as old_config:
  new_config = old_config.read()
```

4. We create a loop with a **`range`** function. **`x`** is **`0`** in the first iteration and 2 in the second iteration because in the **`range`** function, we add two in each loop as we write in the code. After that, we use the **`replace`** function to change the old value with the new one. So, we write the

old value as **command_change[x]** and the new value as **command_change[x+1]**. So in the first loop, it's "new_config = new_config.replace(command_change[0],[1])".

```
for x in range(0, item_count, 2):
  new_config = new_config.replace(command_change[x],
  command_change[x + 1])
```

5. We write the output of the **new_config** variable with the **open** function to the **new_config.txt** file.

```
with open("new_config.txt", "w") as new:
   new.write(new_config)
```

Example 5.5: Replacing configurations on files

```
command_change = [
"""interface GigabitEthernet0/0
description TEST""",
"interface GigabitEthernet0/0",
"""interface GigabitEthernet0/1
no ip address
shutdown""",
"""interface GigabitEthernet0/1
ip address 192.168.10.10 255.255.255.0"""
]
item_count = len(command_change)
with open("old_config.txt") as old_config:
  new_config = old_config.read()
for x in range(0, item_count, 2):
  new_config = new_config.replace(command_change[x],
  command_change[x + 1])
with open("new_config.txt", "w") as new:
  new.write(new_config)
```

In the following text files, we replace the configuration with a basic configuration change script. We can use this script for the migration or swap project as a network engineer:

```
old_config.txt
interface GigabitEthernet0/0

description TEST
```

```
ip address 10.10.10.1 255.255.255.0
duplex auto
speed auto
media-type rj45
!
interface GigabitEthernet0/1
no ip address
shutdown
duplex auto
speed auto
media-type rj45
!
new_config.txt
interface GigabitEthernet0/0
ip address 10.10.10.1 255.255.255.0
duplex auto
speed auto
media-type rj45
!
interface GigabitEthernet0/1
ip address 192.168.10.10 255.255.255.0
duplex auto
speed auto
media-type rj45
!
```

Configure devices with Jinja2 template

Jinja2 template is one of the popular Python modules in network automation. We use this third-party module to create configuration templates and execute them with YAML format files. With these two concepts, we can easily configure thousands of network and system devices with many options.

Introduction to Jinja2 template

We use variables and call them with their values to execute the network commands. Or we write full configuration command for a device and send it to network devices. If we configure a couple of lines in network or system devices, it's normal to use that, like in the previous examples.

But when we configure many parts and devices, we need to make it more automated. For example, if we need to configure 10 interfaces in a single device, as a primary way, we can write all the configurations in a file and execute all of them. On the other hand, we can create a sample interface, and with a loop, we can define different values for them. This can be done with basic Python knowledge, but it's still more work. Instead of writing this kind of script, we have another solution: the *Jinja* template.

Jinja is a template engine for Python. It's fast, has a syntax similar to Python, and has many features to use in many areas. It's a simple and powerful language for templating usage. As network engineers, we use this template to create configuration templates to be executed in the network and system devices. We must download jinja to use in scripts by running the **`pip install Jinja2`** command in the terminal, as we did for the other third-party modules in Python.

In the following code template, there is a sample jinja template for network devices. There is a hostname configuration and two different port configurations. Hostname, interface IP address and subnet mask are changeable variables. So, in this template, we can configure many devices by setting the following variables. Each variable is inside double curly brackets:

```
hostname {{name}}
!
interface GigabitEthernet 0/1
ip address {{ip_address}} {{subnet_mask}}
no shutdown
!
interface GigabitEthernet 0/2
ip address {{ip_address}} {{subnet_mask}}
no shutdown
!
```

Introduction to YAML language

The official website of **Yet Another Markup Language** (**YAML**) says *YAML is a human-friendly data serialization language for all programming languages*. So, it's a programming language, and we need to learn the basics of this language to use with the jinja template. YAML is a modern and primary language used in different languages or tools. Ansible, one of the most popular network automation tools, executes all the scripts in YAML format. So, it's essential to understand that clearly.

- YAML is easy to use, human-friendly, and easily readable format.
- Like in many programming languages such as Python, indentations are very important in the YAML language.
- Unlike in other languages, we cannot use *tab* characters in the YAML language. If we use a tab, the script throws an error; so, we use spaces instead of tab.
- In YAML, we use the # character to write a comment, as in Python.
- YAML is a case-sensitive language, so characters of lower and upper case are different values.
- YAML files are specified with an extension of `.yaml` or `.yml`.

Like in other programming languages, there are several data types in the YAML language: string, boolean, integer, float, list, and dictionary.

For example, we can create a list with a `-` character. In the following example, there are four items: `lion`, `elephant`, and `dog` are strings, and `5` is an integer. All of them make up a list with four items:

```
- lion
- elephant
- dog
- 5
```

Output: `[ 'lion', 'elephant', 'dog', 5 ]`

In this example, we create a block mapping or a dictionary. `lion` is a dictionary variable name, `age` is the key, and `5` is the value of that key. Keys and values are divided by a colon, and keys and their values create dictionaries, as in Python language.

```
- lion:
  age: 5
  color: yellow
```

```
  type: wild
```

Output: `lion: {age: 5, color: yellow, type: wild}`

Rendering Jinja template with a YAML file

When we write scripts using the jinja template, we use both the jinja and YAML modules at the same time. We create the jinja template file in text format and the YAML file in the YAML format. We call or load both YAML and text files inside the code. Finally, we render the template in the script to get the data combined with the YAML file.

In *Example 5.6*, we create a script to execute the jinja template file with the data in the YAML file and render it in the main script.

1. We use both jinja and YAML modules. Instead of importing all modules, we only import the necessary functions. For jinja, we use the **`Environment`** and **`FileSystemLoader`** functions, and for YAML, we use the **`safe_load`** function.

   ```
   from jinja2 import Environment, FileSystemLoader
   from yaml import safe_load
   ```

2. `Environment` is the core object in the jinja module, and it contains the variables as configurations. Inside the parentheses, we use the loader parameter, and it's a template `loader` for this environment. The value of the loader is the **`FileSystemLoader`** function, which loads the template in the file system. We can write a dot to target the current directory or write the full path. In the following code, we enter the dot inside the quote, which is in the parentheses. It means '*load the current file path in the PC as an environment*'. If we need to call the jinja template file from a different directory, we need to write the directory name inside the quote. After that, we assign this value to the **`env`** variable to use later in the code.

   ```
   env = Environment(loader=FileSystemLoader("."))
   ```

3. We need to call the jinja file with the **env** variable.We use the **`get_template`** function to do that. Inside this function, we write the jinja template file, which is **`commands.txt`** in this example. After that, we assign this value to the **`template`** variable.

   ```
   template = env.get_template("commands.txt")
   ```

4. Now, the file is ready to be combined with the data in the YAML file. So, inside the **info.yml** file, we write YAML format variables parameters to assign in the jinja template. First, we need to read the YAML file in Python. We have already imported the **safe_load** function from the **pyyaml** module. We open the file and read it with the **safe_load** function. After that, we assign all the output to the **data** variable.

```
with open("info.yml") as r:
  data = safe_load(r)
```

5. Finally, we combine the template and the YAML data by rendering the script with the **render** function. We display the output of this code.

```
print(template.render(data))
```

In the following code, we write one line of string in the **commands.txt** file. We write double curly brackets and write the variable name inside it as **language**. When we execute the code, Python replaces this variable with the data in the YAML file.

commands.txt

```
We try to learn {{ language }}
```

In the following code, we write the key and its value as **language: Python** in the **info.yml** file. So, code uses this data to combine with the jinja template.

info.yml

```
language: Python
```

When we run the code, we see Python instead of **{{ language }}**. So the code gets the Python data or value of the language key and replaces it with the **{{ language }}** variable in jinja template.

Output: `We try to learn Python`

This example is the basic usage of the jinja template with the YAML file. In the following examples, we try to create more features of jinja, like creating loops inside the template for repeatable actions.

Example 5.6: Rendering jinja template with the YAML file

```
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("commands.txt")
```

```
with open("info.yml") as r:
  data = safe_load(r)
print(template.render(data))
```

Configure devices with Jinja

We already got the data, merged with the jinja template, and rendered it in the previous example. In *Example 5.7*, we use the same script as the one in *Example 5.6*. This time, we configured a hostname and interface in a Cisco router.

In *Example 5.7*, we create two files that are the jinja template and the YAML file. In the jinja template, we write the whole configuration in the following code. We write variable names in double curly brackets for the changeable strings. These are hostname information, interface description, IP address, and subnet mask. So, we need to create four sets of data for these variables. After that, we execute the commands on the router with the **netmiko** module.

commands.txt

```
hostname {{name}}
interface GigabitEthernet0/1
description {{description}}
ip address {{ip_address}} {{subnet_mask}}
no shutdown
```

In the following code, we create four different keys and their values in the YAML data file. These values replace the specific variables in the jinja template.

info.yml

```
name: Router-1
description: Test_Interface
ip_address: 192.168.10.10
subnet_mask: 255.255.255.0
```

We import the **jinja2**, **yaml**, **netmiko**, and **re** modules in the main script. After that, we write the device information to log in.

```
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
from netmiko import Netmiko
import re
```

```
ip = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor":
0.1}
```

We write the jinja template commands as we did in *Example 5.6*.

```
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("commands.txt")
with open("info.yml") as r:
  data = safe_load(r)
command = template.render(data
```

When we check the **command** variable, it's a string of commands. We need to change it to a list by each line. So, if we have 10 lines of commands, we need to change it to a list with 10 items. We use the **split** function from the **RE** module by dividing lines with the **\n** string.

```
command = re.split("\n", command)
```

Finally, we log in to the device and send the configuration commands with **netmiko** functions.

```
print(f"\n---Try to Login: {ip['host']} ---\n")
net_connect = Netmiko(**ip)
output = net_connect.send_config_set(command)
print(output)
```

When we execute the script in *Example 5.7*, we can see a simple router configuration with its parameters.

```
Output of the Jinja Template:
hostname Router-1
interface GigabitEthernet0/1
description Test_Interface
ip address 192.168.10.10 255.255.255.0
no shutdown
```

Example 5.7: Configure a single interface with Jinja

```
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
from netmiko import Netmiko
import re
ip = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor":
0.1}
```

```
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("commands.txt")
with open("info.yml") as r:
  data = safe_load(r)
command = template.render(data)
command = re.split("\n", command)
print(f"\n---Try to Login: {ip['host']} ---\n")
net_connect = Netmiko(**ip)
output = net_connect.send_config_set(command)
print(output)
```

In the previous example, we execute one interface configuration in a router. If we have repeatable configurations, such as configuring many interfaces with the same parameters in routers, we can use `for` loops in the jinja template.

To create a `for` loop in the jinja template, we need to write `for` loop like in Python. It must be between `%` and curly brackets as `{% … %}`. The loop must be finished with the `{% endfor %}` line. Inside the `for` loop, we write the string and variables together, like in the jinja templates as in the previous examples. We call the data from the YAML file as writing `{{ ITERABLE["var1"] }}`.

```
Sample Jinja Template:
{% for ITERABLE in OBJECT %}
Hello {{ ITERABLE["var1"] }}, it's{{ ITERABLE["var2"] }}.
{% endfor %}
Sample YAML file:
ITERABLE:
- var1: World
  var2: Python
```

When we execute the code, the output will be `Hello World, it's Python`.

In *Example 5.8*, we use the same Python script as in *Example 5.7*. But we changed the YAML file and the jinja template. In this scenario, we configure the hostname of the router and three interfaces with their description, IP address, and status as `no shutdown`. Instead of writing three interfaces configuration in the jinja template and YAML file, we write the `for` loop in the jinja template.

In the YAML file, we write the hostname key and value. After that, we write the **`interfaces`** dictionary. There are three lists in this dictionary, and each list has a dictionary with keys and values, for example, **`name`** and **`GigabirEthernet0/1`**. So, we enter the keys and values for each interface.

We write the jinja file with the **`for`** loop. We have data for hostnames, interface name, interface description, IP address, and subnet mask.

Example 5.8: Configure multiple interfaces with Jinja

```
commands.txt:
hostname {{hostname}}
{% for int in interfaces %}
interface {{ int["name"] }}
description {{ int["description"] }}
ip address {{ int["ip_address"] }} {{ int["subnet_mask"] }}
no shutdown
{% endfor %}
info.yml:
hostname: Router-1
interfaces:
- name: GigabitEthernet0/1
  description: Service_Interface
  ip_address: 172.16.10.10
  subnet_mask: 255.255.255.0
- name: GigabitEthernet0/2
  description: MGMT_Interface
  ip_address: 10.0.0.10
  subnet_mask: 255.255.255.0
- name: GigabitEthernet0/3
  description: Dowlink_Interface
  ip_address: 1.1.1.1
  subnet_mask: 255.255.255.0
```

When we execute the Python code in *Example 5.6*, it executes the following configurations in Router-1. We can configure dozens of interfaces with a single template; that's why the jinja template is so powerful for automation.

Output:

```
R1(config)#hostname Router-1
Router-1(config)#
```

```
Router-1(config)#interface GigabitEthernet0/1
Router-1(config-if)# description Service_Interface
Router-1(config-if)# ip address 172.16.10.10 255.255.255.0
Router-1(config-if)# no shutdown
Router-1(config-if)#
Router-1(config-if)#interface GigabitEthernet0/2
Router-1(config-if)# description MGMT_Interface
Router-1(config-if)# ip address 10.0.0.10 255.255.255.0
Router-1(config-if)# no shutdown
Router-1(config-if)#
Router-1(config-if)#interface GigabitEthernet0/3
Router-1(config-if)# description Dowlink_Interface
Router-1(config-if)# ip address 1.1.1.1 255.255.255.0
Router-1(config-if)# no shutdown
Router-1(config-if)#
Router-1(config-if)#end
Router-1#
```

In *Example 5.9*, we configure interface information in multiple devices. We have three Cisco routers, as we did in the previous examples:

1. We import `jinja`, `yaml`, `netmiko` and `RE` modules and create a list of IP addresses to log in to the routers.
   ```
   from jinja2 import Environment, FileSystemLoader
   from yaml import safe_load
   from netmiko import Netmiko
   import re
   ip_list = [ "10.10.10.1", "10.10.10.2", "10.10.10.3" ]
   ```
2. We execute the `Environment` function for the current directory and get the template from the text file. After that, we get the data from the YAML file.
   ```
   env = Environment(loader=FileSystemLoader("."))
   template = env.get_template("commands.txt")
   with open("info.yml") as r:
     data = safe_load(r)
   ```
3. We create a `for` loop with multiple lists. In this example, the `x` iterable gets the items from the `data` list, and the `ip` iterable receives the items from the `ip_list` list. So, we get the data for Router-1 as the first set

of data, that for Router-2 as the second set of data, and the data for Router-3 as the third set of data. We render the data in each loop and execute the commands in a specific router with a **netmiko** module connection.

```
for x,ip in zip(data,ip_list):
  ip = {
   "host": f"{ip}",
   "username": "admin",
   "password": "cisco",
   "device_type": "cisco_ios",
   "global_delay_factor": 0.1}
  command = template.render(x)
  command = re.split("\n", command)
  print(f"\n---Try to Login: {ip['host']} ---\n")
  net_connect = Netmiko(**ip)
  output = net_connect.send_config_set(command)
  print(output)
```

If we use nested loops for the previous examples, the code runs all three device configurations on each device. So, all three devices would be configured as third device configuration. It's the wrong script to run, but in this example, in the first loop, we get the data for the first router and log in to the first router simultaneously; then, it continues the same way on other routers.

Example 5.9: Configure a single interface on multiple devices with jinja

```
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
from netmiko import Netmiko
import re
ip_list = [ "10.10.10.1", "10.10.10.2", "10.10.10.3" ]
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("commands.txt")
with open("info.yml") as r:
  data = safe_load(r)
for x,ip in zip(data,ip_list):
   ip = {
    "host": f"{ip}",
```

```
        "username": "admin",
        "password": "cisco",
        "device_type": "cisco_ios",
        "global_delay_factor": 0.1}
    command = template.render(x)
    command = re.split("\n", command)
    print(f"\n---Try to Login: {ip['host']} ---\n")
    net_connect = Netmiko(**ip)
    output = net_connect.send_config_set(command)
    print(output)
```

In the following command, we create two files again: one for the jinja template and another for the data in the YAML file. We already used this jinja template in the previous examples, which has different usage in the YAML file in *Example 5.9*. It has three items on a list, and each list includes a dictionary with keys and values. In each `for` loop, it gets the data from this file. So, Router-1 gets the first item. Each key has a specific value for Router-1, and the loop gets other device information in the same way.

commands.txt:

```
hostname {{hostname}}
interface {{int_name}}
  description {{description}}
  ip address {{ip_address}} {{subnet_mask}}
  no shutdown
```

info.yml:

```
- hostname: R1
  int_name: GigabitEthernet0/3
  description: Test-1
  ip_address: 10.1.1.1
  subnet_mask: 255.255.255.0
- hostname: R2
  int_name: GigabitEthernet0/3
  description: Test-2
  ip_address: 10.1.1.2
  subnet_mask: 255.255.255.0
- hostname: R3
  int_name: GigabitEthernet0/3
  description: Test-3
```

```
  ip_address: 10.1.1.3
  subnet_mask: 255.255.255.0
```

In *Example 5.10*, we write a script in more advanced usage. We created a **for** loop to configure three interfaces and did this in three routers; it's more complicated according to previous examples. We use the same Python script in *Example 5.9*, and we only change the jinja template file and the YAML file.

We used the same template in the jinja file to create multiple interface configurations in the previous example, but the YAML file is different in this example. We make a dictionary with its items, and we create items of lists again in the **interfaces**. So, when we run this code, it configures three routers with hostnames, interface IP addresses with their subnet masks, and interface descriptions and opens the ports with **no shutdown**.

Example 5.10: Configure multiple interfaces on multiple routers with jinja

```
commands.txt:
hostname {{hostname}}
{% for int in interfaces %}
interface {{ int["int_name"] }}
  description {{ int["description"] }}
  ip address {{ int["ip_address"] }} {{ int["subnet_mask"] }}
  no shutdown
{% endfor %}
info.yml:
- hostname: Router-1
  interfaces:
  - int_name: GigabitEthernet0/1
    description: Service_Interface_1
    ip_address: 172.16.10.10
    subnet_mask: 255.255.255.0
  - int_name: GigabitEthernet0/2
    description: MGMT_Interface_1
    ip_address: 10.0.0.10
    subnet_mask: 255.255.255.0
  - int_name: GigabitEthernet0/3
    description: Dowlink_Interface_1
    ip_address: 1.1.1.1
```

```
   subnet_mask: 255.255.255.0
- hostname: Router-2
  interfaces:
  - int_name: GigabitEthernet0/1
   description: Service_Interface_2
   ip_address: 172.16.10.20
   subnet_mask: 255.255.255.0
  - int_name: GigabitEthernet0/2
   description: MGMT_Interface_2
   ip_address: 10.0.0.20
   subnet_mask: 255.255.255.0
  - int_name: GigabitEthernet0/3
   description: Dowlink_Interface_2
   ip_address: 1.1.1.2
   subnet_mask: 255.255.255.0
- hostname: Router-3
  interfaces:
  - int_name: GigabitEthernet0/1
   description: Service_Interface_3
   ip_address: 172.16.10.30
   subnet_mask: 255.255.255.0
  - int_name: GigabitEthernet0/2
   description: MGMT_Interface_3
   ip_address: 10.0.0.30
   subnet_mask: 255.255.255.0
  - int_name: GigabitEthernet0/3
   description: Dowlink_Interface_3
   ip_address: 1.1.1.3
   subnet_mask: 255.255.255.0
```

In the following output, when we execute the script and check the interface information of three routers, we can see that the interface description and IP address with subnet mask are configured, and ports are configured with a `no shutdown` command.

```
ROUTER-1:
Router-1#show interfaces description
Interface Status Protocol Description
Gi0/0 up up
```

```
Gi0/1 down down Service_Interface_1
Gi0/2 down down MGMT_Interface_1
Gi0/3 down down Dowlink_Interface_1
Router-1#show ip interface brief
Interface IP-Address OK? Method Status Protocol
GigabitEthernet0/0 10.10.10.1 YES NVRAM up up
GigabitEthernet0/1 172.16.10.10 YES manual down down
GigabitEthernet0/2 10.0.0.10 YES manual down down
GigabitEthernet0/3 1.1.1.1 YES manual down down
ROUTER-2:
Router-2#show interfaces description
Interface Status Protocol Description
Gi0/0 up up
Gi0/1 down down Service_Interface_2
Gi0/2 down down MGMT_Interface_2
Gi0/3 down down Dowlink_Interface_2
Router-2#show ip interface brief
Interface IP-Address OK? Method Status Protocol
GigabitEthernet0/0 10.10.10.2 YES NVRAM up up
GigabitEthernet0/1 172.16.10.20 YES manual down down
GigabitEthernet0/2 10.0.0.20 YES manual down down
GigabitEthernet0/3 1.1.1.2 YES manual down down
ROUTER-3:
Router-3#show interfaces description
Interface Status Protocol Description
Gi0/0 up up
Gi0/1 up up Service_Interface_3
Gi0/2 down down MGMT_Interface_3
Gi0/3 down down Dowlink_Interface_3
Router-3#show ip interface brief
Interface IP-Address OK? Method Status Protocol
GigabitEthernet0/0 10.10.10.3 YES NVRAM up up
GigabitEthernet0/1 172.16.10.30 YES manual up up
GigabitEthernet0/2 10.0.0.30 YES manual down down
GigabitEthernet0/3 1.1.1.3 YES manual down down
```

If statement in Jinja

In *Example 5.11*, we use the **`if`** statement inside the jinja template. We execute the same Python script as in *Example 5.7*. We configure **Access List (ACL)** and interface information in the following code.

```
#
access-list 1 permit 10.10.10.0 0.0.0.255
access-list 1 permit 20.20.20.0 0.0.0.255
access-list 1 permit 30.30.30.0 0.0.0.255
#
interface GigabitEthernet0/1
description Service_Interface
ip address 172.16.10.10 255.255.255.0
no shutdown
ip access-group 1 in
#
interface GigabitEthernet0/2
description NOT_USED
ip address
shutdown
#
```

We send the **`shutdown`** or the **`no shutdown`** command according to the interface information, and we add the **`ACL`** command in the **`GigabitEthernet0/1`** interface.

In this example, we have two interfaces: **`GigabitEthernet0/1`** has the IP address, and **`GigabitEthernet0/2`** has no IP address, and the description is **`NOT_USED`**. Empty ports should be **`shutdown`** in network devices, and others should be configured as **`no shutdown`**. So, in the jinja **`for`** loop, we can decide which interface is active or closed.

There is an **`if`** statement line in the jinja file before we close the **`for`** loop. Usage of the **`if`** statement is also similar to that of the **`for`** statement. We write **`{% if int['active'] %} no {% endif %}`** to check if the active variable is true or false in the related interface on the YAML file. If it's true, the **`no`** string will be written, and that line will be **`no shutdown`**. If it's false, the **`no`** string will not be written, and that line will be **`shutdown`**. So, if the active key's value is true, the port is configured as **`no shutdown`**; otherwise, it's configured as **`shutdown`**.

When we check the YAML file, there is an **active** key in the items, and the value is true or false. **GigabitEthernet0/2** is false, so the port is configured as **shutdown**. **GigabitEthernet0/1** is true, so the port is configured as **no shutdown**.

Example 5.11: Access-list configuration in routers with the if statement

```
commands.txt:
{% for acl in access_list %}
access-list {{ acl_no }} permit {{ acl["ip_address"] }} {{
acl["wild_card"] }}
{% endfor %}
{% for int in interfaces %}
interface {{ int["name"] }}
  description {{ int["description"] }}
  ip address {{ int["ip_address"] }} {{ int["subnet_mask"] }}
  {% if int['active'] %}no {% endif %}shutdown
  {% if int['active'] %}ip access-group {{ acl_no }} in {%
  endif %}
{% endfor %}
info.yml:
acl_no: 1
access_list:
  - ip_address: 10.10.10.0
  wild_card: 0.0.0.255
  active: true
  - ip_address: 20.20.20.0
  wild_card: 0.0.0.255
  - ip_address: 30.30.30.0
  wild_card: 0.0.0.255
interfaces:
  - name: GigabitEthernet0/1
  description: Service_Interface
  ip_address: 172.16.10.10
  subnet_mask: 255.255.255.0
  active: true
  - name: GigabitEthernet0/2
  description: NOT_USED
  active: false
```

When we execute the code, the **GigabitEthernet0/2** interface has no IP address, and the description is **NOT_USED**, as in the following output. And it's also **administratively down**, which is **shutdown**.

```
Router-1#show interface description
Interface Status Protocol Description
Gi0/0 up up
Gi0/1 down down Service_Interface
Gi0/2 admin down down NOT_USED
Gi0/3 down down
Router-1#show ip interface brief
Interface IP-Address OK? Method Status Protocol
GigabitEthernet0/0 10.10.10.1 YES NVRAM up up
GigabitEthernet0/1 172.16.10.10 YES manual down down
GigabitEthernet0/2 unassigned YES unset administratively down
down
GigabitEthernet0/3 unassigned YES unset administratively down
down
```

In *Example 5.11*, the **GigabitEthernet0/2** interface has no IP address. But, when we check the output of our script, the **ip address** command is executed in the Cisco device. Because there is no IP and subnet mask on the command, the router returns a **% Incomplete command.** as a warning.

```
Router-1(config-if)# ip address
% Incomplete command.
```

We can create another **if** statement for the **ip address** command. So, if there is no **ip_address** key in the list item in the YAML file, **ip address {{ int['ip_address'] }} {{ int['subnet_mask'] }}** line is not run in the device. As we see in the YAML file, the **GigabitEthernet0/2** has no IP address key. So, our code doesn't send the **ip address** command.

```
{% if int['ip_address'] -%}
ip address {{ int['ip_address'] }} {{ int['subnet_mask'] }}
{% endif -%}
```

Configure devices with Napalm module

Network Automation and Programmability Abstraction Layer with Multivendor (**NAPALM**) is a Python library that connects network devices by a unified API.

- It's built on top of the netmiko module.
- It currently supports Cisco, Juniper, and Arista devices. It does not have as wide a range of vendor support as the netmiko module.
- It has a feature to manipulate configurations and commit or roll back the configuration on the network devices.
- We can combine NAPALM with the network automation frameworks such as Ansible and Salt.

We can use NAPALM with multiple platforms simpler than netmiko because it uses the same syntax for different vendors, and it's one of the powerful parts of NAPALM.

There are plenty of `getters` or the `get` functions in NAPALM to collect the logs in a smarter way that can be converted to a JSON format or more readable for us. We need to enable the **Secure Copy Protocol** (**SCP**) server in Cisco devices with the `ip scp server enable` command to log in with NAPALM. Otherwise, the code gives an error and asks to enable the SCP protocol.

We must run the `pip install napalm` command in the terminal to install the NAPALM module. For more details about NAPALM, like supported network devices, getters, and configurations, you can check the official website at the following link:

https://napalm.readthedocs.io/en/latest/support/index.html

Collect logs from devices with NAPALM

We will collect logs with the NAPALM module in this section. In *Example 5.12*, we collect interface information or route information detail. We can also convert it to the **JavaScript Object Notation** (**JSON**) format to make it more readable for us.

1. We import the napalm and JSON modules. After that, we write the host variable and add the hostname as IP address, username, and password information to log in to the device.

```
import napalm
import json
host = {"hostname": "10.10.10.1", "username": "admin",
"password": "cisco"}
```

2. We call the **get_network_driver** function from the NAPALM module and assign it to the **driver** variable. We choose the specific vendor to get data: for cisco - ios, for arista - eos, and for juniper - junos. We write **(**host)** to login a device like in **netmiko** module.We assign this value to a variable called **connect**.

   ```
   driver = napalm.get_network_driver("ios")
   connect = driver(**host)
   ```

3. We call the **open** function with connecting variable to open a connection session on the device. Now, we can call the getters or the commands to get the data or log from the device. We use the **get_interfaces** function to get the details of the interfaces in the device and write all the output logs to the **output** variable. We can also call other functions for other purposes.

   ```
   connect.open()
   output = connect.get_interfaces()
   ```

4. We can display the output with a **print** function. If we print only the **output** function, it displays a very long line of a dictionary, which is not a good way to read the data. In this situation, we use the JSON format to read the **output** variable. We use the **dumps** function from the JSON module and add an indentation between the items as one. Finally, we close the connection session with the **close** function.

   ```
   print(json.dumps(output, indent=1))
   connect.close()
   ```

Example 5.12: Get interface information with NAPALM

```
import napalm
import json
host = {"hostname": "10.10.10.1", "username": "admin",
"password": "cisco"}
driver = napalm.get_network_driver("ios")
connect = driver(**host)
connect.open()
output = connect.get_interfaces()
print(json.dumps(output, indent=1))
connect.close()
```

When we execute the script in *Example 5.12*, it collects detailed interface data from the routers, as in the following output. We can see the description, MAC address, MTU size, and more.


```
{
"GigabitEthernet0/0": {
  "is_enabled": true,
  "is_up": true,
  "description": "aaa",
  "mac_address": "0C:70:B7:89:00:00",
  "last_flapped": -1.0,
  "mtu": 1500,
  "speed": 1000.0
},
"GigabitEthernet0/1": {
  "is_enabled": false,
  "is_up": false,
  "description": "11",
  "mac_address": "0C:70:B7:89:00:01",
  "last_flapped": -1.0,
  "mtu": 1500,
  "speed": 1000.0
},
"GigabitEthernet0/2": {
  "is_enabled": false,
  "is_up": false,
  "description": "22",
  "mac_address": "0C:70:B7:89:00:02",
  "last_flapped": -1.0,
  "mtu": 1500,
  "speed": 1000.0
},
"GigabitEthernet0/3": {
  "is_enabled": false,
  "is_up": false,
  "description": "33",
  "mac_address": "0C:70:B7:89:00:03",
  "last_flapped": -1.0,
  "mtu": 1500,
  "speed": 1000.0
```

```
}
}
```

We can also get route information from the device. We call the **`get_route_to`** function to get the data, and we write the destination route IP address as a string in parentheses.

```
output = connect.get_route_to("192.168.10.30")
```

When we execute **`show ip route`** with the destination IP address, we can see the following output *From CLI* part. If we execute the Python script, we can see the output as the *From Python Code* part. From Router-1, we have an **Open Shortest Path First** (**OSPF**) neighbor as **`192.168.10.30`** in the router. So, we use this IP address to check the route details.

From CLI:

```
Router-1#show ip route 192.168.10.30
Routing entry for 192.168.10.30/32
  Known via "ospf 1", distance 110, metric 2, type intra area
  Last update from 10.10.10.3 on GigabitEthernet0/0, 00:37:50
  ago
  Routing Descriptor Blocks:
  * 10.10.10.3, from 10.10.10.3, 00:37:50 ago, via
  GigabitEthernet0/0
    Route metric is 2, traffic share count is 1
```

From Python code:


```
{ "192.168.10.30/32": [
  {
    "protocol": "ospf",
    "outgoing_interface": "GigabitEthernet0/0",
    "age": 2223,
    "current_active": true,
    "routing_table": "default",
    "last_active": true,
    "protocol_attributes": {},
    "next_hop": "10.10.10.3",
    "selected_next_hop": true,
    "inactive_reason": "",
    "preference": 2 } ] }
```

In *Example 5.13*, we add an **`ip_list`** variable as three IP addresses, and we create a **`for`** loop to log in all three devices and collect the detailed ARP table information with the **`get_arp_table`** function.

Example 5.13: Collect logs from multiple devices with NAPALM

```
import napalm
import json
ip_list = ["10.10.10.1","10.10.10.2","10.10.10.3"]
for ip in ip_list:
  print(f"*** Connecting to {ip} ***")
  host = {"hostname": ip, "username": "admin", "password":
  "cisco"}
  driver = napalm.get_network_driver("ios")
  connect = driver(**host)
  connect.open()
  output = connect.get_arp_table()
  print(json.dumps(output, indent=1))
  connect.close()
```

To check the full list of the **`get`** functions, you can check the following link to the NAPALM official website:

https://napalm.readthedocs.io/en/latest/support/index.html

Configure devices with NAPALM

In *Example 5.14*, we try to configure the **Border Gateway Protocol** (**BGP**) configuration in the router.

Instead of the **`get`** functions, we use the **`load_merge_candidate`** function to call the configuration file. We define the filename inside the parentheses.

```
output =
connect.load_merge_candidate(filename=”command_list.txt”)
```

We can print the output of the **`compare_config`** function to see the difference when we add configurations on the device as an option. In the following output, we can see that all configurations are added in order, and all of them get the **+** plus character at the beginning of the line. It means that this command is added to the device. If it’s **-** minus, it’s deleted from the device.

```
print(connect.compare_config())
```

```
+router bgp 100
+ bgp log-neighbor-changes
+ neighbor 10.10.10.2 remote-as 100
+ neighbor 10.10.10.2 description to_Router-2
+ neighbor 10.10.10.2 next-hop-self
```

Finally, we use the **commit_config** function to execute all these commands to the device.

```
connect.commit_config()
```

Example 5.14: Configure dynamic routes in devices with NAPALM

```
import napalm
host = {"hostname": "10.10.10.1", "username": "admin",
"password": "cisco"}
driver = napalm.get_network_driver("ios")
connect = driver(**host)
connect.open()
output =
connect.load_merge_candidate(filename="command_list.txt")
print(connect.compare_config())
connect.commit_config()
connect.close()
```

Instead of using a file to send the configuration to the device, we can use the string to load the candidate configuration. We use the **config** parameter and write a string to it or a **string** variable as the **command**.

```
output = connect.load_merge_candidate(config=command)
```

Configure devices with Nornir module

Nornir is one of the most popular and open-source network automation frameworks that is written in the Python language. The power of nornir is to use pure Python code when writing automation scripts. So, it has no limits like other automation tools and can also handle tasks in advanced usage. We can import and use any Python module and features with **nornir** module.

- Nornir is a framework formed by **plugins**. We can extend its capability by using advanced features of plugins.
- Nornir has a multithreaded feature that can connect many devices simultaneously, about which we learned in the previous chapters as

parallelism. It saves time when we make automation in a large-scale network.

- We use the NAPALM or **`netmiko`** modules to connect devices in the nornir framework. It also supports the use of paramiko or scrapli, which is also the SSH connection module.
- Inventories are created by the YAML files like we did in NAPALM module.

Nornir has similarities with the Ansible automation framework. It has tasks to execute commands and an inventory system to keep the device connection information. Ansible has its domain-specific language, but nornir uses Python in scripts.

- **Inventory**: Inventory stores the device information to connect. It can be IP address, username, password, platform as vendor type, or more. Inventory files are written in the YAML language. The inventory system has three structures in the more advanced usage:
    - **Hosts**: It stores unique host information like IP address or platform data.
    - **Groups**: We can group the data about the devices, like platform information.
    - **Defaults**: It stores similar data, which is identical in devices, like username or password.

  We can combine all three files in a single configuration file with the **`SimpleInventory`** plugin.
- **Tasks**: The task is a plugin that is a reusable Python code to execute functions in a single device. It returns an output at the end of the task.
- **Functions**: The function is a plugin from the **`nornir_utils`** module that executes an action or task. The **`print_result`** function is the most common function to display the output of the tasks.

We need to install two modules in the terminal, i.e., **`pip install nornir`** and **`pip install nornir-utils`**, to use the **`nornir`** module.

We need to install the additional **`nornir`** modules in the terminal for the connection type. We are using **`netmiko`** and **`napalm`** with nornir in this book, so we need to install **pip install nornir-napalm** and **pip install nornir_netmiko** in the terminal.

Configure inventory in Nornir

We can configure inventory with one file, i.e., **hosts.yaml**, in a simple solution. It's the default file for host information in the nornir framework.

In the YAML file, we start with three hyphen characters, **---**, at the top of the file. After that, we write the host information as dictionaries. In the following output, we have three device information. We write the device name at the beginning, like **Router-1**, which is not a hostname on the device; we can enter any string. Inside the device, we write **hostname** for the management IP address, **platform** as a vendor type, and **username** and **password** as the login information to the device.

hosts.yaml

```
---
Router-1:
   hostname: 10.10.10.1
   platform: ios
   username: admin
   password: cisco
Router-2:
   hostname: 10.10.10.2
   platform: ios
   username: admin
   password: cisco
Router-3:
   hostname: 10.10.10.3
   platform: ios
   username: admin
   password: cisco
```

We can use an advanced feature of inventory systems, and we can divide the data in **hosts.yaml** file into different YAML files.

In the following code, we have a **groups.yaml** file. We can add platforms as vendor information, like Cisco, Juniper, or Arista. For example, we create the **ios** and **junos** variables, and the value of the **platform** is **ios** or **junos**. So, in the **hosts.yaml** file, we can create a key as **groups** and write an item as **ios** for a Cisco device. We can use the **groups** feature for multi-vendor automation scripts.

We can also create a **defaults.yaml** file to enter the same data that can run in devices, such as **username** and **password**. In our example, all three devices have the same username and password information.

We only write the hostname and groups keys inside the **hosts.yaml** file with the router name, like **Router-1**.

```
groups.yaml
---
ios:
  platform: "ios"
junos:
  platform: "junos"
defaults.yaml
---
username: admin
password: cisco
hosts.yaml
---
Router-1:
   hostname: 10.10.10.1
   groups:
   - ios
Router-2:
   hostname: 10.10.10.2
   groups:
   - ios
Router-3:
   hostname: 10.10.10.3
   groups:
   - ios
```

In the following output, after configuring all three files, we create a **config.yaml** file to combine these files in the **nornir** framework. We added an **inventory** dictionary. We use the **SimpleInventory** plugin to combine these YAML files. Inside the **options**, we write **host_file**,

group_file, **defaults_file** keys and the YAML files that we create in a string.

In the second part, we write the **runner** plugin to use the multithreading feature of nornir. So, we can run the code on 10 or more devices concurrently. We use the **threaded** plugin, and in the **options**, we use the **num_workers** key as three. We can change this parameter to tell our code how many devices to connect concurrently. In this scenario, it tries to log in to three devices simultaneously and faster. If we enter 1 as the value of **num_workers**, the code logs in to devices one by one, which is slower.

config.yaml

```
---
inventory:
   plugin: SimpleInventory
   options:
    host_file: "hosts.yaml"
    group_file: "groups.yaml"
    defaults_file: "defaults.yaml"
runner:
   plugin: threaded
   options:
    num_workers: 3
```

Connection to devices with Nornir-Netmiko

In *Example 5.15*, we collect the **show arp** command output from Cisco devices using netmiko over the **nornir** framework. We use the **hosts.yaml** file above.

1. We import **nornir** for the nornir framework, **nornir_utils** for functions, and **nornir_netmiko** to log in via the **netmiko** module.
   ```
   from nornir import InitNornir
   from nornir_utils.plugins.functions import print_result
   from nornir_netmiko import netmiko_send_command
   ```
2. We initialize nornir with the **InitNornir()** function in the **nornir** module and assign it to a **connect** variable.
   ```
   connect = InitNornir()
   ```

3. We call the **run** function and assign it to a **result** variable. Inside this function, we write the **task** parameter to call the **netmiko_send_command** function from the **nornir_netmiko** module and the **command_string** parameter to execute the command in the device.

   ```
   result = connect.run(task=netmiko_send_command,
   command_string="show arp")
   ```

4. Finally, we have a special **print** function in the **nornir_utils** module. We call this function the **result** variable.

   ```
   print_result(result)
   ```

Example 5.15: Collect logs with Nornir-Netmiko

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_command
connect = InitNornir()
result = connect.run(task=netmiko_send_command,
command_string="show arp")
print_result(result)
```

When we execute the code in *Example 5.15*, it connects to three devices very fast. This is because the value of the **num_workers** parameter, which is used for multithreading, is 20 by default. So, if we don't set the **num_workers** value in the code, the code connects to max 20 device concurrency.

We can change the **num_workers** parameter to one in the following code. We must write it inside the **options** from the **threaded** plugin, which we set in the **runner** parameter.

```
connect = InitNornir(runner={"plugin": "threaded", "options":
{"num_workers": 1}},)
```

In *Example 5.16*, we can also add the YAML configuration files **groups**, **defaults**, **hosts.yaml** and **config.yaml**. To do that, we write the **config_file** parameter inside the **InitNornir** function and the value **config** file as a string.

Example 5.16: Collect logs by "config.yaml" with Nornir-Netmiko

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
```

```
from nornir_netmiko import netmiko_send_command
connect = InitNornir(config_file="config.yaml")
result = connect.run(task=netmiko_send_command,
command_string="show arp")
print_result(result)
```

In *Example 5.17*, We can execute multiple commands. The **`command_string`** variable must be a string, so we cannot add a list by various commands; we need to use a loop. That's why we create a commands variable as a list with the **`show`** commands and create a **`for`** loop after executing the **`InitNornir()`** function. We still use multithreading.

Example 5.17: Collect multiple logs with Nornir-Netmiko

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
import nornir_netmiko
commands = ["show arp", "show ip interface brief", "show
interface description"]
connect = InitNornir()
for comm in commands:
  result =
  connect.run(task=nornir_netmiko.netmiko_send_command,
  command_string=comm)
  print_result(result)
```

In *Example 5.18*, we use the **`netmiko_send_config`** function from the **`nornir_netmiko`** module to execute commands in the devices. We execute basic SNMP v2 configuration in the Cisco devices. We have the options of sending commands from a variable or from a file.

To send commands from a variable, we use the **`config_commands`** parameter inside the **`run`** function, and to send them from a file, we use the **`config_file`** parameter with a file named **`string`**.

```
result = connect.run(task=netmiko_send_config,
config_commands=commands)
result = connect.run(task=netmiko_send_config,
config_file="command_list.txt")
```

Example 5.18: SNMP configuration in devices with Nornir-Netmiko

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
```

```
from nornir_netmiko import netmiko_send_config
commands = ["snmp-server community public RO", "snmp-server community private RW",
     "snmp-server enable traps cpu threshold",
     "snmp-server host 10.10.10.150 version 2c snmp_user",
     "snmp-server source-interface informs GigabitEthernet0/0"]
connect = InitNornir()
result = connect.run(task=netmiko_send_config,
config_commands=commands)
print_result(result)
```

In *Example 5.19*, we can also filter devices. So, we can choose nornir to show only some commands or devices by the **`filter`** function. In the following code, we write the **`filter`** function with the hostname parameter as **`10.10.10.2`**, which is **`Router-2`** in our example. So, the code only connects this device and executes the **`show arp`** command.

Example 5.19: Using the filter function in the Nornir Framework

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_command
connect = InitNornir()
connect = connect.filter(hostname="10.10.10.2")
result = connect.run(task=netmiko_send_command,
command_string="show arp")
print_result(result)
```

Connection to devices with Nornir-NAPALM

In *Example 5.20*, we use the NAPALM connection mode with the **`nornir_napalm`** module. Everything is the same with the netmiko connection; we only change the task and commands parameter. We use the **`napalm_cli`** function for tasks and the **`commands`** parameter for sending commands. We save configuration commands with the **`write`** command in the Cisco devices.

Example 5.20: Save configuration with Nornir-NAPALM

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_napalm.plugins.tasks import napalm_cli
```

```
connect = InitNornir()
result = connect.run(task=napalm_cli, commands=["write"])
print_result(result)
```

In the `print_result` function, if we write the result variable as `[Router-1]`, the code collects data from all devices but only shows `Router-1` logs.

```
print_result(result["Router-1"])
```

If we use only the `print` function, the output will be different from that of the `print_result` function. We write device information with the specific command, which only displays the a particular log from that device as an output.

```
print(result["Router-1"].result["write"])
```

Instead of running commands, we can use the NAPALM feature of getters. In the following code, we must import the `napalm_get` function and add it to tasks. After that, we need to add a `getters` parameter and write the **get** function.

```
result = connect.run(tasks=napalm_get, getters=
"get_interfaces_ip")
```

We can also configure devices with the `nornir-napalm` module. To do that, we must import the `napalm_configure` function and add it to tasks, as shown in the following code. After that, we write a filename in the `filename` parameter.

```
result = connect.run(task=napalm_configure,
filename="command_list.txt")
```

Configure devices by Nornir and Jinja template

In *Example 5.21*, we combine nornir with the jinja2 template. So, the code has a more advanced usage. In the following code, we combine the jinja2 example and the `nornir-netmiko` example.

We get the jinja template from the `commands.txt` file and the data from the `info.yml` file, and we create a template as shown in the following code. We execute the following OSPF configuration commands in routers using the `nornir` module and the Jinja template.

```
router ospf 1
network 10.10.10.0 0.0.0.255 area 0
network 20.20.20.0 0.0.0.255 area 1
```

```
network 30.30.30.0 0.0.0.255 area 0
#
interface Loopback0
ip ospf network point-to-point
ip ospf cost 100
commands.txt
router ospf {{ ospf_process }}
{% for net in networks %}
  network {{ net["ip_address"] }} {{ net["subnet_mask"] }} area
  {{ net["area_id"] }}
{% endfor %}
int loopback {{ loopback_int }}
ip ospf cost {{ lo_cost }}
ip ospf network {{ net_type }}
info.yml
ospf_process: 1
networks:
  - ip_address: 10.10.10.0
  subnet_mask: 255.255.255.0
  area_id: 0
  - ip_address: 20.20.20.0
  subnet_mask: 255.255.255.0
  area_id: 1
  - ip_address: 30.30.30.0
  subnet_mask: 255.255.255.0
  area_id: 0
loopback_int: 0
lo_cost: 100
net_type: point-to-point
```

After that, we save this configuration template to the **conf.txt** file in the current directory. Finally, we call this file with the **config_file** parameter using the **netmiko_send_config** function.

Example 5.21: Configure OSPF with Jinja template in devices by Nornir

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_config
from jinja2 import Environment, FileSystemLoader
```

```
from yaml import safe_load
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("commands.txt")
with open("info.yml") as r:
  data = safe_load(r)
with open("conf.txt","w") as w:
  w.write(template.render(data))
connect = InitNornir()
result = connect.run(task=netmiko_send_config,
config_file="conf.txt")
print_result(result)
```

Conclusion

In this chapter, we learned about advanced network automation features to make scripts more flexible with high quality. We use templates, new connection methods, and an automation framework, which are essential in high-level automation engineering.

The next chapter will focus on file transfers with SCP, SSH, or SFTP protocols. We have different modules to complete these tasks in Python, and we also create plots of data from the devices like, CPU usage or interface traffic graphics, by plotting modules.

Multiple choice questions

1. How can you use the **for** loop in the jinja template?

    a. `{% for X in Y %}`
       `CONTENT`

    b. `{% for X in Y %}`
       `CONTENT`
       `{% end %}`

    c. `{% for X in Y %}`
       `CONTENT`
       `{% endfor %}`

    d. `{% for X in Y %}`
       `CONTENT`

```
{%}
```

2. Which of the following is not one of the get functions in NAPALM?
    a. `get_eigrp_neighbors`
    b. `get_facts`
    c. `get_bgp_neighbors`
    d. `get_vlans`
3. What is the maximum number of devices that can be connected simultaneously in the nornir framework?”
    a. 1
    b. 5
    c. 10
    d. 20

Answers

1. c
2. a
3. d

Questions

1. Using jinja, write a script to configure three network devices with the nornir-napalm module.
2. Write a nornir script to get VLAN data and save it to an Excel file in columns and rows.

CHAPTER 6
File Transfer and Plotting

This chapter will focus on file transfer and plotting data, including example scripts. We will use network connection modules to log in to devices and transfer files in upload and download directions. We will use file transfer protocols like FTP, SFTP, and SCP, and we will back up the device configuration file to the local PC with the SSH or SCP protocols. We also use netmiko to collect data and draw a plot in a new window.

Structure

In this chapter, we will cover the following topics:

- File transfers
    - Backup configuration file with SSH
    - File transfer with FTP connection
    - File transfer with SFTP connection
    - File transfer with SCP connection
    - Netmiko SCP connection with concurrent module
    - File transfer with Nornir SCP connection
    - Backup configuration file with SCP
- Plotting data
    - Plotting CPU levels
    - Plotting interface bandwidth

Objectives

We will use FTP, SFTP, and SCP to log in and transfer files in the network and system devices. Even if we can do this task in the CLI, we will create automation scripts and transfer files to many devices concurrently by using parallelism in Python language. We will use the **`ftplib`**, **`ftpretty`**, **`paramiko`**, **`netmiko`**, and **`nornir`** modules to transfer files in different protocols. We will also collect data periodically from devices with netmiko and draw a plot to check the graphics. We will use the matplotlib module to plot any data from the device and customize the drawing window.

File transfers

File transfer is one of the critical topics in the daily work of network engineering. We transfer a lot of data both ways: upload or download data. The data can be software or patch file, configuration data, packet captures, logs, or any other information to transfer.

There are various protocols in file transfer methods, such as **File Transfer Protocol** (**FTP**), **Secure File Transfer Protocol** (**SFTP**), and **Secure Copy Protocol** (**SCP**). Many file transfer tools exist, such as *FileZilla*, *WinSCP*, and more. We can transfer files or folders with these tools, but in this chapter, we will create custom-designed scripts and share files with these scripts. These are more flexible than the FTP tools because we can automate the network with our scripts and transfer files to many devices concurrently. We can see that the transfer success or file size matches the local file. We can touch on all the transferring processes in the advanced usage of these scripts.

- **FTP**: It's a simple file transfer protocol developed as one of the oldest protocols on the internet and used for over 40 years. It creates a connection session between two machines to transfer a file from one to the other. Connections occur over IP addresses, like with the other file transfer protocols. However, this protocol has no encryption, so it's insecure. It uses data channels, which are at risk of being manipulated by hackers.
- **SFTP**: It's created as an alternative to **File Transfer Protocol** (**FTP**) to transfer files over the SSH protocol. Instead of FTP, SFTP uses the SSH protocol more securely. It creates a single connection instead of FTP and encrypts the data for transfer. So, it's more secure than FTP.
- **SCP**: It transfers files over an encrypted tunnel based on the SSH protocol. We can use SCP only to transfer files both ways. Unlike FTP and SFTP, we cannot delete files, create directories, or list all content

in a directory in the remote host. It uses SSH for authentication, so it's also a secure file transfer protocol.

We have various modules in Python to use the file transfer protocols, such as **`ftplib`**, **`paramiko`**, **`netmiko`**, **`napalm`**, and **`nornir`**.

Backup configuration file with SSH

Before using the file transfer protocols on network devices, we can start with *Example 1.1* to log in devices with the netmiko SSH protocol and collect data. In this example, we collect complete device configurations, such as running configuration in Cisco or configuration in Juniper and Huawei devices. We can modify this script for other vendors like Nokia or Arista.

1. We import the **`netmiko`** function from the **`netmiko`** module. We create two lists, **`ip_list`** and **`device_list`**, for netmiko device type and the IP information for the management of devices. After that, we create a **`for`** loop with the **`zip`** feature, allowing us to iterate two lists together with a **`for`** loop. We write the netmiko connection parameters inside the loop as **`host`**, **`username`**, **`password`**, **`device_type`**, and **`global_delay_factor`**. So, in each iteration, the code gets the item in the **`ip_list`** and **`device_list`** lists.

   In the following code, we have three Cisco devices, a Juniper, and a Huawei device. So, the device types are different, and it's **`juniper_junos`** for Juniper and **`huawei`** for Huawei. If the loop gets the IP address as **`10.10.20.1`**, it also brings the **`juniper_junos`** item. So, we have an **`ip`** variable with a parameter to log in to a Juniper device.

```
from netmiko import Netmiko
ip_list = [ "10.10.10.1", "10.10.10.2", "10.10.10.3",
"10.10.20.1", "10.10.30.1" ]
device_list =
["cisco_ios","cisco_ios","cisco_ios","juniper_junos","hua
wei"]
for ip,device in zip(ip_list,device_list):
 ip = {
  "host": f"{ip}",
```

```
 "username":"admin",
 "password":"cisco",
 "device_type": f"{device}",
 "global_delay_factor": 0.1
}
```

2. We define the commands to collect the configuration data from each device. Collecting the configuration commands is different for each vendor. For Cisco, it's **`show running-config`**; for Juniper, it's **`show configuration`** or **`show configuration | display set`** according to show in different formats; and for Huawei, it's **`display current-configuration`**.

   We use the **`if`** condition to change the value of the **`command`** variable. For each vendor, the value will change. At the end of the **`if`** condition, we use the **`else`** statement so that in case the vendor is not Cisco, Juniper, or Huawei, it displays as a warning that this device is a different vendor's product.

```
if ip["device_type"] == "cisco_ios":
 command = "show running-config"
elif ip["device_type"] == "juniper_junos":
 command = "show configuration | display set"
elif ip["device_type"] == "huawei":
 command = "display current-configuration"
else:
print("This is different vendor (Not Cisco,Huawei or
Juniper)")
```

3. We create a **`try`**…**`except`** statement to check whether IP is reachable. Inside the **`try`** statement, we connect to devices and send the commands with the **`command`** variable in the previous code. So, for each IP or host, we send the specific vendor command to the device.

```
try:
 print(f"\n----Try to login: {ip['host']}---\n")
 net_connect = Netmiko(**ip)
 output = net_connect.send_command(command)
except:
 print(f"***Cannot login to {ip['host']}")
```

4. After we collect the configuration output from each device, we save it to different files, and the file name is the device's IP address. In this example, if we can log in and run commands on all five devices, we will get five text files in the same directory as our code.

```
with open (f"{ip['host']}.txt","w") as w:
 w.write(output)
```

Example 6.1: Backup configuration in a text file by SSH

```
from netmiko import Netmiko
ip_list = [ "10.10.10.1", "10.10.10.2", "10.10.10.3",
"10.10.20.1", "10.10.30.1" ]
device_list =
["cisco_ios","cisco_ios","cisco_ios","juniper_junos", "huawei"]
for ip,device in zip(ip_list,device_list):
  ip = {
   "host": f"{ip}",
   "username":"admin",
   "password":"cisco",
   "device_type": f"{device}",
   "global_delay_factor": 0.1
  }
  if ip["device_type"] == "cisco_ios":
   command = "show run"
  elif ip["device_type"] == "juniper_junos":
   command = "show configuration | display set"
  elif ip["device_type"] == "huawei":
   command = "display current-configuration"
  else:
   print("This is different vendor (Not Cisco,Huawei or
   Juniper)")
  try:
   print(f"\n----Try to login: {ip['host']}---\n")
   net_connect = Netmiko(**ip)
   output = net_connect.send_command(command)
  except:
   print(f"***Cannot login to {ip['host']}")
  with open (f"{ip['host']}.txt","w") as w:
   w.write(output)
```

File transfer with FTP connection

FTPlib **module**: We can use the **ftplib** module to log in to network and system devices by FTP, and we can transfer files both ways: upload and download. In Cisco and Juniper, the transfer system is different, and there is a direct connection between the peers to make only copy processes. The **ftplib** module is used to connect and run commands on the remote device. So, we cannot use this module in Cisco and Juniper routers. We can use it in Huawei, other vendors, or sytem devices.

Table 6.1 contains some functions to use the **ftplib** module for transferring files. Some of these functions are similar to Linux terminal commands:

Function	Description
storbinary()	Upload file from local host to remote host
retrbinary()	Download file from remote host to local host
mkd()	Create a new directory
cwd()	Change the directory of a folder or a file
dir()	List all content in the current directory
nlst()	Create a list with filenames in the current directory in the router
delete()	Delete a file
rmd()	Delete a folder
quit()	Close the session and exit

Table 6.1: *Ftpblib functions*

In *Example 6.2*, we have two codes: downloading files to a local device and uploading files to a remote device. In the first part, we upload a file from the local PC to the Huawei router with the **ftplib** module.

1. We import the ftplib module.
   ```
   import ftplib
   ```
2. We call the **FTP** class inside the **ftplib** module and enter three parameters in order: host information as the management IP address, username, and password. We create all these variables at the top of the code and call them inside the **FTP** class. We assign the **FTP** class to an

ftp variable. We also define the **filename** variable and value as the target file name in a string.

```
host = "10.10.30.1"
username = "admin"
password = "huawei"
filename = "test.txt"
ftp = ftplib.FTP(host, username, password)
```

3. We open the source file in binary mode to read, so we write the **rb** parameters together for the **open** function. After that, we call the **storbinary** function, which is used to upload files to the remote host. Inside the function, we write the **STOR** word, the **filename** as a string, and **upload** as the **open** function name. Finally, we terminate the FTP session with the **quit** function.

```
with open(filename, "rb") as upload:
  ftp.storbinary(f"STOR {filename}", upload)
ftp.quit()
```

In *Example 6.2*, with the second part, we download a file from the router to our PC. We use the **open** function in the following code to write the **retrbinary**. We use **wb** for the **write** and **binary** modes. Inside the **retrbinary** function used to download files, we write **RETR** and the **filename** inside a string. This time, the other parameter is **download.write**. **download** is a variable that we set in the **open** function to assign the output.

```
with open(filename, "wb") as download:
  ftp.retrbinary(f"RETR {filename}", download.write)
```

Example 6.2: File transfer with FTP via Ftplib

```
Upload a File:
import ftplib
host = "10.10.30.1"
username = "admin"
password = "huawei"
filename = "test.txt"  #Local PC Filename
ftp = ftplib.FTP(host,username,password)
with open(filename, "rb") as upload:
  ftp.storbinary(f"STOR {filename}", upload)
ftp.quit()
```

```
Download a File:
import ftplib
host = "10.10.30.1"
username = "admin"
password = "huawei"
filename = "test.txt"  #Local PC Filename
ftp = ftplib.FTP(host,username,password)
with open(filename, "wb") as download:
ftp.retrbinary(f"RETR {filename}", download.write)
ftp.quit()
```

In *Example 6.3*, we get the file size information from the router and compare the size in the local host. If both sizes are identical, we give output saying the size is the same.

1. We import the **`ftplib`**, **`re`**, and **`os`** modules. We use the **`os`** module to check the file size in the local host. OS module execute the operating system commands such as **dir** or **cd** commands.

    ```
    import ftplib
    import re
    import os
    ```

2. We enter the **`host`**, **`username`**, **`password`**, and **`filename`** variables, like in the previous example. We create an empty list named **`files`** that we use in the following part. After that, we log in to the device with the **`FTP`** function.

    ```
    host = "10.10.30.1"
    username = "admin"
    password = "huawei"
    filename = "test.txt"
    files = []
    ftp = ftplib.FTP(host, username, password)
    ```

3. We use the **`dir`** function. We append all the outputs to the files list. If we directly write **`dir()`**, the code will give an outcome of the list of the files in the router's current directory. That's why we append all outputs to a variable named **`files`** and then convert it to a string with the **`join`** function.

    ```
    output = ftp.dir(files.append)
    files = " ".join(files)
    ```

4. The output of the **dir** command in CLI is given as follows. In the last line, the target file is **test.txt**, and the size is **31,570** bytes. So, we need to get this value with the **re** module. We write the special sequences to find the data that we need in the following code:

```
<HW_Router-1>dir
Directory of cfcard:/
Idx  Attr     Size(Byte)  Date          Time       FileName
  0  drw-              -  Jun 1 2022 02:00:11  aaa
  1  drw-              -  Jun 1 2022 02:00:11  bios
  2  -rw-          5,406  Jun 1 2022 02:00:11  vrpcfg.zip
  3  -rw-         31,570  Jun 1 2022 02:00:11  test.txt
file_size = re.findall(f"(\d+)\s+\w+\s+\d+\s+\d+:\d+\s+
{filename}", files)
```

5. Now, we need to find the file size in the local host, so we call the **getsize** function from the **os** module.

```
local = os.path.getsize(filename)
```

6. We need to compare two variables. The **file_size** variable we got from the device is a list, and the first item is our value. So, we call the first value of this list. We compare both variables as an integer value. If both are identical, we display an output that both file sizes are the same. Otherwise, we display that file size has a problem.

```
if int(local) == int(file_size[0]):
print(f"'{filename}': '{local}' Bytes. It's same on local
and remote host.")
else:
 print("ERROR: File size has a problem.")
```

Example 6.3: Compare file sizes in the remote and local devices

```
import ftplib
import re
import os
host = "10.10.30.1"
username = "admin"
password = "huawei"
filename = "test.txt"
files = []
ftp = ftplib.FTP(host, username, password)
```

```
output = ftp.dir(files.append)
files = " ".join(files)
file_size = re.findall(f"(\d+)\s+\w+\s+\d+\s+\d+:\d+\s+
{filename}", files)
local = os.path.getsize(filename)
if int(local) == int(file_size[0]):
  print(f"'{filename}': '{local}' Bytes. It's same on local and
  remote host.")
else:
  print("ERROR: File size has problem.")
```

Ftpretty module: Instead of the **`ftplib`** module, we can use the **`ftpretty`** module to transfer files from or to remote devices. We need to install this module by using the **`pip install ftpretty`** command in the terminal. We can transfer any file format with this module.

- The **`get`** function is used to download a file from the remote device to our local device.
- The **`put`** function is used to upload a file from our local device to the remote device.

Table 6.2 contains some functions to use the **`ftpretty`** module for transferring files and file handling:

Function	Description
`put()`	Upload file from local host to remote host
`get()`	Download file from remote host to local host
`mkdir()`	Create a new directory
`cd()`	Change the directory of a folder or file
`delete()`	Delete a file in remote host
`list()`	List all content in the current directory
`close()`	Terminate the session and exit

Table 6.2: *Ftpretty functions*

In *Example 6.4*, we download and upload files both ways. We create functions to do that.

1. We import the **`ftpretty`** function from the **`ftpretty`** module.

    ```
    from ftpretty import ftpretty
    ```

2. We write the hostname as the management IP address of the **router**, **username**, and **password** variables.

```
host = "10.10.30.1"
username = "admin"
password = "huawei"
```

3. We create two functions: **upload** and **download**. We will call them according to our task in the following code. We have two parameters in the **upload** function: **local_file** and **remote_file**, so we define files when we call the upload function. We call the **ftpretty** function and assign it to the **ftp** variable. Inside the function, we write the device information, like **host**, **username**, and **password**. Then, we call the **put** function to upload the file from our local PC to a remote device. We write the local filename with its extension and the remote filename with its extension. At the end, we terminate the FTP connection session with the **close** function.

```
def  upload(local_file, remote_file):
 ftp = ftpretty(host, username, password)
 ftp.put(local_file, remote_file)
 ftp.close()
```

4. In this function, we write code similar to the last part. We only change the function name and use the **get** function instead of the **put** function. We also change the parameter order in the **get** function. First, we write the remote host filename with its extension, and then the local PC filename with its extension.

```
def download(local_file, remote_file):
 ftp = ftpretty(host, username, password)
 ftp.get(remote_file, local_file)
 ftp.close()
```

5. Finally, we must call **upload** the function to execute the function. In this example, we call the **upload** function that we create. So, we write upload as the function name and write two parameters. We write **test.txt** as the local filename and **test2.txt** as the remote filename. When we execute the script, it will upload the **test.txt** file to the remote device with the **test2.txt** filename.

```
upload("test.txt","test2.txt")
```

Example 6.4: Upload and download files with the ftpretty module

```
from ftpretty import ftpretty
host = "10.10.10.1"
username = "admin"
password = "cisco"
def  upload(local_file, remote_file):
  ftp = ftpretty(host, username, password)
  ftp.put(local_file, remote_file)
  ftp.close()
def  download(local_file, remote_file):
  ftp = ftpretty(host, username, password)
  ftp.get(remote_file, local_file)
  ftp.close()
upload("test.txt","test.txt")
```

We can list all files by calling the **`list()`** function; it creates a **`list`** variable.

```
ftp.list()
```

In *Example 6.5*, we create a script to get the file size of each file in the remote host and display it. To do that, we use an additional parameter, which is the **`extra`** parameter in the list function. With the **`extra`** parameter as a **`True`** value, we can list all files by details, such as filename, size, created time, and more. The value of the **`extra`** parameter is **`False`** by default. We add the **`extra`** parameter in the following code and assign the **list** function to the variable **`a`**. If we print **`a`**, it displays all items with details. We can also print the size value of items in this variable. We create a **`for`** loop, inside which we print list items in each iteration with the **`name`** and **`size`** keys. So, in each iteration, the code gets the value from the **`name`** and **`size`** keys to print.

```
a=ftp.list(extra=True)
for i in range(len(a)):
  print("File:",a[i]["name"], "- Size:", a[i]["size"])
```

Example 6.5: Get file size of each file with ftpretty

```
from ftpretty import ftpretty
host = "10.10.30.1"
username = "admin"
password = "huawei"
ftp = ftpretty(host, username, password)
```

```
a=ftp.list(extra=True)
for i in range(len(a)):
print("File:",a[i]["name"], "- Size:", a[i]["size"], "Bytes")
```

Output:

```
File: paf.txt - Size: 230 Bytes
File: vrpcfg.zip - Size: 12400 Bytes
File: license.bin - Size: 1023412 Bytes
```

We can also create a folder and put the source file from the local PC to the remote host using code similar to that in *Example 6.5*. After we log in to the device, we create a folder named **test_folder** in the remote host with the **mkdir** function. Afterward, we must go to this directory to upload the source file. So, we use the **cd** function to change the directory to the new file. After that, we upload the **test.txt** file with the **put** function.

```
ftp.mkdir("test_folder")
ftp.cd("test_folder")
ftp.put("test.txt","test.txt")
```

We can list the folder content from the CLI:

```
<HW_Router-1>dir test_folder
Directory of cfcard:/test_folder/
  Idx  Attr      Size(Byte)  Date         Time        FileName
0  -rw-         10,200  Jun 12 2022 10:00:00    test.txt
```

File transfer with SFTP connection

After FTP, we continue with SFTP, a more commonly used protocol than FTP. We can use the **paramiko** module to connect devices with SFTP.

Table 6.3 has some functions to use in **paramiko**, like in the **ftplib** module:

Function	Description
`put()`	Upload file from local host to remote host
`get()`	Download file from remote host to local host
`mkdir()`	Create a new directory
`rmdir()`	Delete a directory
`chdir()`	Change the directory of a folder or a file
`remove()`	Delete a file in remote host

`listdir()`	List all content in the current directory as a list
`rename(old_name, new_name)`	Change the name of the file or directory in remote host
`close()`	Terminate the session and exit

Table 6.3: *Ftpretty functions*

In *Example 6.6*, we connect to Huawei routers to transfer files with SFTP. We create three functions: SFTP connection, file upload, and file download. For Huawei or any other network and system device, you must configure the SFTP server and enable it to log in by script. After that, we can quickly log in to devices.

1. We import the **`paramiko`** module.

   ```
   import paramiko
   ```

2. We create the first function named **`sftp_connect`** and then call the **`SSHclient`** function. Over this function, we call the **`set_missing_host_key_policy`** and the **`connect`** functions we wrote several times in the previous chapters. We call the **`open_sftp`** function to create an SFTP session between our PC and the remote host, and we assign it to a variable named **`sftp`**. In the last line, we return the **`sftp`** variable. We learned how to call a variable outside the function in the previous.

   ```
   def sftp_connect():
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy
    (paramiko.AutoAddPolicy())
   ssh.connect(hostname="10.10.30.1", username="admin",
   password="huawei")
    sftp = ssh.open_sftp()
    return sftp
   ```

3. So, we finish our SFTP connection function. Now, we can write the upload function. We use the **`put`** function from paramiko to upload a file from the local PC to the remote host. We cannot directly write the **`put(local, remote)`** function. We must use the variable from the **`sftp_connect`** function, which is the **`sftp`** variable. So, we write **`sftp_connect().put()`**. To call a variable from a function, there are two things to do. First, we must return the variable end of the **`source`**

function. Second, we need to call the **source** function. We terminate the SFTP session with the **close** function.

```
def sftp_upload(local_file,remote_file):
  sftp_connect().put(local_file,remote_file)
  sftp_connect().close()
```

4. Then, we write the download function with the **get** function from the **paramiko** module. We call the **sftp_connect** function again but assign it to a variable named **sftp_d**. So, **sftp_d** equals the **sftp** variable in the **sftp_connect** function. So, we can write **sftp_d.get** in this function. There is no difference between the previous and the following code; only the usage is different. We terminate the SFTP session with the **close** function.

```
def sftp_download(remote_file,local_file):
  sftp_d = sftp_connect()
  sftp_d.get(remote_file,local_file)
  sftp_d.close()
```

5. All three functions finish. We can call the **sftp_download** function by writing the remote and local files in order as a string.

```
sftp_download("remote_test.txt", "local_test.txt")
```

We write the **sftp_upload** function to upload the files from the local PC to the remote host. We write local and remote files in order.

```
sftp_download("local_test.txt", "remote_test.txt")
```

Example 6.6: SFTP file transfer with Paramiko

```
import paramiko
def sftp_connect():
  ssh = paramiko.SSHClient()
  ssh.set_missing_host_key_policy (paramiko.AutoAddPolicy())
  ssh.connect(hostname="10.10.30.1", username="admin",
  password="huawei")
  sftp = ssh.open_sftp()
  return sftp
def sftp_upload(local_file,remote_file):
  sftp_connect().put(local_file,remote_file)
  sftp_connect().close()
def sftp_download(remote_file,local_file):
  sftp_d = sftp_connect()
```

```
  sftp_d.get(remote_file,local_file)
  sftp_d.close()
sftp_download("remote_test.txt","local_test.txt")
```

If we want to print all of items in the default directory on the remote host, we can use **`sftp_connect().listdir()`** in *Example 6.6*.

```
print(sftp_connect().listdir())
```

We can also change the destination filename with its extension in the **`paramiko`** module. We use the same function to connect a device with SFTP and call the **`rename`** function by writing the old and new names in order inside the parentheses.

```
sftp_connect().rename("test.txt","test2.txt")
print(sftp_connect().listdir())
```

Output: `['aaa', 'bios', 'bootlogfile', 'statlogfile', 'fpga', 'diaginfo', 'test2.txt']`

File transfer with Netmiko SCP connection

SCP is one of the secure file transfer protocols in network and system devices. Paramiko module can support SFTP, so we cannot use the **`paramiko`** module for SCP transfer. However, we can use the netmiko module to transfer files by the SCP protocol.

We can easily use the SCP protocol with the **`netmiko`** module. There are two main functions to use SCP in the **`netmiko`** module: **`Netmiko`** and **`file_transfer`**. The **`Netmiko`** function is used to log in devices with the SSH protocol we used in the previous chapters. The **`file_transfer`** function is used to connect network devices to transfer files from a local to a remote host and vice versa. It uses the SCP protocol to make the file transfer.

In *Table 6.4*, the **`file_transfer`** function has different parameters to execute:

Function	Description
`source_file`	Specify the source file with its extension
`dest_file`	Specify the destination file with its extension
`direction`	`put`: Upload from local PC to remote host

	`get`: Download from remote host to local PC
file_system	Filesystem information, for example `flash:`
overwrite_file	True/False: If file exists in destionation, whether to overwrite
disable_md5	True/False: Whether to perform Md5 encryption check after the transfer

Table 6.4: *Netmiko "file_transfer" function parameters*

In *Example 6.7*, we log in to a single Cisco router and transfer files from the local PC to the remote host.

1. We import the **netmiko** and **file_transfer** functions from the **netmiko** module.
   ```
   from netmiko import Netmiko, file_transfer
   ```
2. We write the device connection information for the **netmiko** module, as we did in the previous examples. Then, we call the **netmiko** function to log in the device with SSH protocol and assign it to the **net_connect** variable.
   ```
   device = {"host": "10.10.10.1", "username": "admin",
   "password": "cisco", "device_type": "cisco_ios",
   "global_delay_factor": 0.1 }
   net_connect = Netmiko(**device)
   ```
3. After we log in to the device, we can transfer any file from both sides with the **file_transfer** function. The mandatory parameters are **source_file**, **dest_file** and **direction**. Other parameters in Table 6.4 are optional. Inside the parentheses, we write the **connection** variable **net_connect**. After that, we write three parameters with their values as strings. The direction parameter has two options: **put** and **get**. We write **put** to upload a file from a local PC to a remote host.
   ```
   file_transfer(net_connect, source_file="test.txt",
   dest_file="test.txt", direction="put")
   ```
4. Finally, we terminate the SSH session with the **disconnect** function.
   ```
   net_connect.disconnect()
   ```

Example 6.7: SCP file transfer with Netmiko

```
from netmiko import Netmiko, file_transfer
device = {"host": "10.10.10.1", "username": "admin",
"password": "cisco", "device_type": "cisco_ios",
```

```
"global_delay_factor": 0.1 }
net_connect = Netmiko(**device)
file_transfer(net_connect,
       source_file="test.txt",
       dest_file="test10.txt",
       direction="put")
net_connect.disconnect()
```

We can also download a file from a remote host to a local PC by writing the same code as in *Example 6.7* while replacing **put** with **get** in the **direction** parameter.

In *Example 6.8*, we connect three routers and upload a file using the **file_transfer** protocol. In this example, we create a JSON file and get each device's information from that file inside a new function.

In the following output, we create a JSON file called **device_list.json**. There's a dictionary on the top, and it has keys and their values; the values are lists. So, we try to convert this JSON file to **netmiko** device login format as a dictionary with its items.

device_list.json:

```
{
  "Router-1" :
  [
   {   "host": "10.10.10.1",
    "username": "admin",
    "password": "cisco",
    "device_type": "cisco_ios",
    "global_delay_factor": 0.1
   }
  ],
  "Router-2" :
  [
   {   "host": "10.10.10.2",
    "username": "admin",
    "password": "cisco",
    "device_type": "cisco_ios",
    "global_delay_factor": 0.1
   }
```

```
  ],
  "Router-3" :
  [
   {   "host": "10.10.10.3",
    "username": "admin",
    "password": "cisco",
    "device_type": "cisco_ios",
    "global_delay_factor": 0.1
   }
  ]
}
```

We write the code to connect device with the SCP protocol with the following steps.

1. We import the **netmiko** and **json** modules.

    ```
    from netmiko import Netmiko, file_transfer
    import json
    ```

2. We create a function to convert the **device_list.json** JSON file to a list. Each item in the list has information about each device for a netmiko connection. We create an empty list to append each dictionary at the end. After that, we call the **open** function to open the **device_list.json** file and parse or convert the JSON file format to the Python file format and assign it to the **data** variable.

    ```
    def json_device():
     host_list = []
     with open('device_list.json') as json_file:
      data = json.load(json_file)
    ```

3. JSON file is converted to Python file as the **data** variable. We need to get each item in the dictionary, so we create a **for** loop. In each loop, we got the item as a dictionary of device information by writing **item[1][0]**. When we print this value in the loop, we can see that in each iteration, the code displays each router's information that can be used by the **netmiko** module. We append or add each dictionary in a **host_list** list and return **host_list** to use it outside the function.

    ```
     for item in data.items():
      host = item[1][0]
      print (host)
    ```

```
    host_list.append(host)
  return host_list
```

The value of the **host_list** is in the following output. It's a list that contains dictionaries as items.

```
[{'host': '10.10.10.1', 'username': 'admin', 'password':
'cisco', 'device_type': 'cisco_ios',
'global_delay_factor': 0.1}, {'host': '10.10.10.2',
'username': 'admin', 'password': 'cisco', 'device_type':
'cisco_ios', 'global_delay_factor': 0.1}, {'host':
'10.10.10.3', 'username': 'admin', 'password': 'cisco',
'device_type': 'cisco_ios', 'global_delay_factor': 0.1}]
```

4. Now, we can call the **json_device()** function and assign this function to the **host** variable. The **host** variable's value equals the **host_list** variable.

```
host = json_device()
```

5. We need to create a **for** loop to connect each device because in the **Netmiko(**host_info)** function, we must write the host information as a string; we cannot use a list. That's why we create a **for** loop to achieve our goal. We log in to each device and upload the **test.txt** file by the **file_transfer** function, as we did in the previous example. After all, we terminate the SSH session with the remote device by the **disconnect** function.

```
for ip in host:
  net_connect = Netmiko(**ip)
  file_transfer(net_connect,
      source_file="test.txt",
      dest_file="test111.txt",
      direction="put",
      )
net_connect.disconnect()
```

Example 6.8: SCP file transfer by getting device information from a JSON file

```
from netmiko import Netmiko, file_transfer
import json
def json_device():
  host_list = []
```

```
  with open('device_list.json') as json_file:
   data = json.load(json_file)
  for item in data.items():
   host = item[1][0]
   host_list.append(host)
   print(host_list)
  return host_list
host = json_device()
for ip in host:
  net_connect = Netmiko(**ip)
  file_transfer(net_connect,
       source_file="test.txt",
       dest_file="test.txt",
       direction="put",
       )
  net_connect.disconnect()
```

We can add the **`disable_md5`** optional parameter inside the **`file_transfer`** function. By default, there is no **md5** validation. The **md5** value assigns to **False**.. So, the code checks the source and destination files' md5 encryption. If we change the value to **`True`**, the code will not validate or check md5 encryption.

```
file_transfer(net_connect,
      source_file="test.txt",
      dest_file="test.txt",
      direction="put"
      disable_md5 = True
      )
```

We can also specify whether to overwrite if the file exists on the peer side, which can be a local PC or remote host, according to direction. The **`overwrite_file`** parameter is **`False`** by default. So, we can overwrite a file if the file exists in the destination host, and we can change its value to **`True`** and overwrite a file even if it exists in the destination host.

```
file_transfer(net_connect,
      source_file="test.txt",
      dest_file="test.txt",
      direction="put"
overwrite_file = True
```

```
        )
```

We can change the default directory in the remote host. When we upload a file to the Cisco router, it automatically uploads the file to the default directory, **flash:**. But we can also have the option to change the file directory with the **file_transfer** function. We use the **file_system** parameter to change the **upload** directory. We write **flash2:** in the following code as the destination file system. When we run the **dir flash2:** command in Cisco CLI after we upload the file, we can see that file is successfully uploaded to **flash2:** instead of **flash:**.

```
file_transfer(net_connect,
        source_file="test.txt",
        dest_file="test123.txt",
        direction="put",
        file_system="flash2:"
        )
```

Output:

```
Router-1#dir flash2:
Directory of flash2:/
  4  -rw-       31900  Aug 20 2022 17:56:16 +00:00  test123.txt
966656 bytes total (897024 bytes free)
```

One of the best features of file transfer in netmiko is that we can see the progress in the output of the code. We need to import the **progress_bar** function from the **netmiko** module and add **progress4=progress_bar** inside the **file_transfer** function.

```
From netmiko import progress_bar
…
…
file_transfer(net_connect,
        source_file="test.txt",
        dest_file="test.txt",
        direction="put",
        file_system="flash:",
        overwrite_file = True,
        progress4=progress_bar
        )
```

When we execute the code, code logs in three devices one by one to transfer the `test.txt` file in order. So, when we need to upload a large file, such as a software file, we can see the progress of the transfer quickly.

Output:

```
Transferring file to ('10.10.10.1', 22):
test.txt                  | (0.00%)
Transferring file to ('10.10.10.1', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>                         | (51.36%)
Transferring file to ('10.10.10.1', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| (100.00%)
Transferring file to ('10.10.10.2', 22):
test.txt                    | (0.00%)
Transferring file to ('10.10.10.2', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>                         | (51.36%)
Transferring file to ('10.10.10.2', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| (100.00%)
Transferring file to ('10.10.10.3', 22):
test.txt                    | (0.00%)
Transferring file to ('10.10.10.3', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>                         | (51.36%)
Transferring file to ('10.10.10.3', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| (100.00%)
```

Netmiko SCP connection with concurrent module

We can upload files to devices one by one in the `for` loop. In *Example 6.8*, we have three devices to transfer files, and we send a text file of 32KB. So, uploading that file to all three devices takes approximately 20 seconds, which is not too much time. But if we try to upload a file whose size is 200MB, maybe it will take 30 minutes to upload three devices. If we have 30 devices to upload, the total time to upload is 5 hours, which is not acceptable in real life.

We use the parallelism feature of Python to reduce the time. We connect devices concurrently. Even if we have 30 devices to send 200 MB worth of files, it will take only 10 minutes, which is the period to upload the software file to a single device, because we use the multithreading feature.

We use the **`concurrent.futures`** function. We already used the **`threading`** module with paramiko in the previous chapters.

In *Example 6.9*, we transfer the **`test.txt`** file concurrently to three devices. In the previous example, it took 20 seconds. In this example, it takes 7 seconds. So, the code acts like it is connecting to a single device. Even if we add 50 more devices in the example lab, the time will stay the same: 7 seconds. It's the significant power of the parallelism feature in Python.

1. We import the **`ThreadPoolExecutor`** function from the **`concurrent.futures`** module and the **`netmiko`** with its necessary functions.

   ```
   from concurrent.futures import ThreadPoolExecutor
   from netmiko import Netmiko, file_transfer, progress_bar
   ```

2. We create a **`get_ip_address`** function to get the IP addresses from a file. We collect IP addresses with the **`open`** function and write them to a list with the **`splitlines`** function line-by-line. After that, we return the variable **`host_list`**. When we call the **`get_ip_address`** function, it returns the **`host_list`** variable.

   ```
   def get_ip_address():
    with open("device_list.txt") as r:
     host_list = r.read().splitlines()
    return host_list
   ```

3. We create another function named **`netmiko_scp`**. Inside this function, we create a variable as a host to add the device information for the netmiko connection. We have an **`ip`** variable that is the parameter in the **`netmiko_scp`** function. We connect the device with the **`Netmiko`** function and transfer the file with the **`file_transfer`** function. After it finishes, we disconnect from the device and return the function. We already did this in the previous examples:/p>

   ```
   def netmiko_scp(ip):
   host = {"ip": ip, "username": "admin", "password":
   "cisco", "device_type": "cisco_ios"}
    print(f"---Try to Login:{ip}---")
    net_connect = Netmiko(**host)
    file_transfer(net_connect,
        source_file="test.txt",
        dest_file="eee.txt",
   ```

```
            direction="put",
            file_system="flash:",
            overwrite_file=True,
            progress4=progress_bar)
        net_connect.disconnect()
        return
```

4. We call the **ThreadPoolExecutor** function as executor. We use the **map** function with the **executor**. The **map** function is used with a function and an iterable, which can be anything, like a list. In this example, we write the **netmiko_scp** function and the IP list we collect in the **get_ip_address** function inside the parentheses. We assign the **get_ip_address** function to the **host_ip** variable and add it to the parentheses of the **map** function.

   The code with **ThreadPoolExecutor** can only log in to 12 devices concurrently by default, but we can change the default value with the **max_workers** parameter. In this example, we write 25, but it can be more, based on our PC resources. This is because when we run the code, our local PC CPU/memory resources can be increased. We can also change the value to 1, so we can see the time difference when we change the **max_workers** value.

```
with ThreadPoolExecutor(max_workers=25) as executor:
 host_ip = get_ip_address()
 result = executor.map(netmiko_scp, host_ip)
```

Example 6.9: SCP file transfer with netmiko simultaneously with parallelism

```
from concurrent.futures import ThreadPoolExecutor
from netmiko import Netmiko, file_transfer, progress_bar
def get_ip_address():
  with open("device_list.txt") as r:
   host_list = r.read().splitlines()
  return host_list
def netmiko_scp(ip):
  host = {"ip": ip, "username": "admin", "password": "cisco",
  "device_type": "cisco_ios"}
  print(f"---Try to Login:{ip}---")
  net_connect = Netmiko(**host)
```

```
  file_transfer(net_connect,
       source_file="test.txt",
       dest_file="test.txt",
       direction="put",
       file_system="flash:",
       overwrite_file=True,
       progress4=progress_bar)
  net_connect.disconnect()
  return
with ThreadPoolExecutor(max_workers=25) as executor:
  host_ip = get_ip_address()
  result = executor.map(netmiko_scp, host_ip)
```

When we execute the script, in the output, the code is connected to all three devices concurrently, as in the following output. In the previous example, the code connected to **10.10.10.1** the first device in the list. After the file transfer was finished, it continued with **10.10.10.2**, like working in a loop. But in this example, the code connected all devices simultaneously and didn't wait until the current connection was finished, so there is no order in the following output.

Output:

```
---Try to Login:10.10.10.1---
---Try to Login:10.10.10.2---
---Try to Login:10.10.10.3---
Transferring file to ('10.10.10.3', 22):
test.txt                | (0.00%)
Transferring file to ('10.10.10.1', 22): test.txt
| (0.00%)
Transferring file to ('10.10.10.3', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>                         | (51.36%)
Transferring file to ('10.10.10.3', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| (100.00%)
Transferring file to ('10.10.10.2', 22):
test.txt                | (0.00%)
Transferring file to ('10.10.10.1', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>                         | (51.36%)
Transferring file to ('10.10.10.1', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| (100.00%)
```

```
Transferring file to ('10.10.10.2', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>                         | (51.36%)
Transferring file to ('10.10.10.2', 22): test.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| (100.00%)
```

File transfer with Nornir SCP connection

We can also use the **nornir** framework to transfer files by the SCP protocol. We use the **nornir_netmiko** module to handle the transfer process in nornir, which is a powerful module that we can connect many devices simultaneously. In the pure **netmiko** module, to connect multiple devices simultaneously, we need to use the parallelism feature of Python code, such as the **threading** or **concurrent** modules. But with nornir, we don't need to use this module in the script. The parallelism structure is already written in the back end of the nornir framework, and we don't need to handle this complicated process.

In *Example 6.10*, we connect all three devices simultaneously and transfer files. We use the YAML file as **hosts.yaml** in the following output:

```
hosts.yaml
---
Router-1:
   hostname: 10.10.10.1
   platform: ios
   username: admin
   password: cisco
Router-2:
   hostname: 10.10.10.2
   platform: ios
   username: admin
   password: cisco
Router-3:
   hostname: 10.10.10.3
   platform: ios
   username: admin
   password: cisco
```

1. We import the **InitNornir** function from the **nornir** module to initiate the **nornir** framework to connect devices. Then, we import the

print_result function from the **nornir.utils** module to print the detailed output of the process. And finally, we import the **netmiko_file_transfer** function from **nornir_netmiko** to transfer files by the **netmiko** module in the **nornir** framework.

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_file_transfer
```

2. We initialize the **nornir** framework with connecting devices in the **hosts.yaml** file and assign its value to the **connect** variable.

```
connect = InitNornir()
```

3. We call the **run** function to call a task. In this example, we call the task value **netmiko_file_tranfer**. The direction is from the local PC to the remote device. We only write the **source_file** and **dest_file** parameters with their **string** values.

```
result = connect.run(task=netmiko_file_transfer,
source_file="test.txt", dest_file="kkk.txt")
```

4. When we call the **print_result** function to see the output.

```
print_result(result)
```

In the output, the code returns a value of the task as **True**, which means that the file transfer process is successfully done.

Example 6.10: SCP file transfer with Nornir

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_file_transfer
connect = InitNornir()
result = connect.run(task=netmiko_file_transfer,
       source_file="test.txt",
       dest_file="test.txt"
       )
print_result(result)
Output:
netmiko_file_transfer******************************************
********** *******
* Router-1 ** changed : False
**************************************************
```

```
vvvv netmiko_file_transfer ** changed : False
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv INFO
True
^^^^ END netmiko_file_transfer
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* Router-2 ** changed : False
**************************************************
vvvv netmiko_file_transfer ** changed : False
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv INFO
True
^^^^ END netmiko_file_transfer
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* Router-3 ** changed : False
**************************************************
vvvv netmiko_file_transfer ** changed : False
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv INFO
True
^^^^ END netmiko_file_transfer
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We also have various parameters in the **nornir-netmiko** module, like in the **netmiko** module. We can choose the direction from a remote device to the local PC by writing the **direction** parameter as **get**. By default, the value of this parameter is **put**.

```
result = connect.run(task=netmiko_file_transfer,
       source_file="test2.txt",
       dest_file="test2.txt",
       direction="get")
```

We can also check the file's existence on the destination device by adding the **disable_md5** parameter. The default value is **False**, so we can overwrite the file when we transfer the same file. But if we change the value of this parameter to **True**, we cannot transfer the file if it exists in the destination device. And nornir gives an error message **ValueError: File already exists, and overwrite_file is disabled** as an output.

```
result = connect.run(task=netmiko_file_transfer,
       source_file="test2.txt",
       dest_file="test2.txt",
       direction="put",
```

```
disable_md5=True
)
```

Backup configuration file with SCP

In *Example 6.11*, we save **running-configuration** in the Cisco routers and download the config file as a backup to the local PC with the filename, including the time stamp. We use similar code in *Example 6.8*, using the **device_list.json** JSON file. We add more code pieces to it.

1. We import the **netmiko**, **json**, modules, and **datetime** function from the **datetime** module. We save the configuration backup file with the time stamp.

```
from netmiko import Netmiko, file_transfer
import json
from datetime import datetime
```

2. We open JSON files and convert them into a list. Then, we can get the data from this list to connect to the devices. Then, we call this function and assign it to a **host** variable.

```
def json_device():
 host_list = []
 with open('device_list.json') as json_file:
  data = json.load(json_file)
 for item in data.items():
  host = item[1][0]
  host_list.append(host)
 return host_list
host = json_device()
```

3. We create a loop to log in devices in order. We execute the **wr** command to save the Cisco device configuration.

```
for ip in host:
 net_connect = Netmiko(**ip)
 print(f"\n----Try to login: {ip['host']}---\n")
 save = net_connect.send_command("wr")
 print(save)
```

4. We call the **datetime** function and the current PC time by calling the **now** function inside. After that, with the **strftime** function, we get the

year, month, day, hour, minute, and second data by writing the following code. We assign the current time value to the **time** variable.

```
time = datetime.now().strftime("%Y_%m_%d_%H_%M_%S")
```

5. We get the device hostname with the **find_prompt** function in netmiko. It's used to get the hostname information of the device quickly. We must write **[:-1]** after that function because it also gets the **#** character after the hostname in Cisco, like **Router-1#**. For other vendors, that prompt sign character is different. So, we get the hostname information and assign it to the **hostname** variable.

```
hostname = net_connect.find_prompt()[:-1]
```

6. We call the **file_transfer** function to download the **startup-config** file from the Cisco device. We write the **dest_file** parameter with the hostname and the **time** variables. We also add the **file_system** parameter to change the default directory. The **startup-config** file is in **nvram:**, so we write the value of this parameter as **nvram:**. We finally terminate the session with the **disconnect** function.

```
file_transfer(net_connect,
    source_file="startup-config",
    dest_file=f"{hostname} backup_config {time}.cfg",
    direction="get",
    file_system="nvram:",
    overwrite_file=True
    )
net_connect.disconnect()
```

Example 6.11: Backup configuration file with Netmiko SCP

```
from netmiko import Netmiko, file_transfer
import json
from datetime import datetime
def json_device():
  host_list = []
  with open('device_list.json') as json_file:
   data = json.load(json_file)
  for item in data.items():
   host = item[1][0]
   host_list.append(host)
  return host_list
```

```
host = json_device()
for ip in host:
  net_connect = Netmiko(**ip)
  print(f"\n----Try to login: {ip['host']}---\n")
  save = net_connect.send_command("wr")
  print(save)
  time = datetime.now().strftime("%Y_%m_%d_%H_%M_%S")
  hostname = net_connect.find_prompt()[:-1]
  file_transfer(net_connect,
       source_file="startup-config",
       dest_file=f"{hostname} backup_config {time}.cfg",
       direction="get",
       file_system="nvram:",
       overwrite_file=True
       )
  net_connect.disconnect()
```

When we execute the code in *Example 6.11*, the code downloads three files from remote devices, which start with their hostname and end with the time that we download to our PC as the following filenames. If we back up a configuration file from a device multiple times, we can quickly find the most updated one or its old versions with time stamps in the configuration file.

```
Filenames in the local PC:
Router-1 backup_config 2022_08_21_18_29_16
Router-2 backup_config 2022_08_21_18_29_24
Router-3 backup_config 2022_08_21_18_29_33
```

Plotting data

Graphics are used in network automation. We can create plots, scatters, or bars. **Network Monitoring System** (**NMS**) tools take the data from network devices and plot it as graphics. It can be interface bandwidth in both ways, inbound and outbound, or CPU and memory level to show if it's close to the threshold or increasing over an extended period. We can take action according to these graphics daily or for a very long period.

We can also get the alarms from all devices in the network and categorize them according to their severity level, such as minor, major, and critical alarms. After that, we can create a graphic with bars monthly to see the alarm process. So, we can see what happens in our network over a long period.

By adding the time stamp, we can collect traffic bandwidth data in a period and see the traffic changes in the related interface. We focus on collecting data from a device in a short period and plot it as a graphic.

We can create a scheduler, like the **crontab** feature in Linux, which automates the scripts to execute in specified periods or time intervals repeatedly.

We can use the **matplotlib** module in Python to plot the data in a graphic, and it's a powerful data visualization and third-party Python plotting module. So, we must install it first with the **pip install matplotlib** command in the terminal.

The usage area of **matplotlib** is large; we can draw data of any type. It's also designed to work with NumPy arrays in the NumPy module. Numpy extends multi-dimensional lists, arrays, and matrices in more complex usage.

We can change the color and thickness of the lines in the plot. We can add **x** and **y** axis values and grids in the graphic. We can also 3D surface graphics with this module.

We must import the **pyplot** function from the **matplotlib** module in the following code to use the **pyplot** function in our scripts.

```
from matplotlib import pyplot
```

So, each time we write **pyplot** in the code, it calls the **pyplot** function from the **matplotlib** module. We have another option to use the function: by changing its name in the code. After importing a module or function, we write **as** and write any value. In the following code, we write **as plt**. So, in the same Python file, if we write **plt**, it calls the **pyplot** function from **matplotlib**.

```
from matplotlib import pyplot as plt
```

In the official documentation of **matplotlib**, we can use the module functions by the following code. We can also import a function by writing **import MODULE_NAME.FUNCTION_NAME**. In the example, it's **import matplotlib.pyplot**.

```
import matplotlib.pyplot as plt
```

There are various functions and parameters in the **`pyplot`** function. We can manipulate the data and draw different styles of graphs with these functions efficiently.

- **`plot(X, Y)`**: This is the primary function to draw a graphic in the **`pyplot`** function. There are two mandatory parameters inside the **`plot`** function: X and Y. These are X-axis and Y-axis variables in the drawings. We create these variables with the same quantity of items. If the total item count of the **`X`** and **`Y`** variables does not match, the code gives an error: **`ValueError: x and y must have same first dimension`**. The list values can be of different data types, like strings, integers, or others.
- **`xlabel()`**: It's used to add an information header on the X-axis.
- **`ylabel()`**: It's used to add an information header on the Y-axis.

  For example, if we draw a CPU usage with pyplot, we can write the graphic header on the X-axis as **`Time in Seconds`** and on the Y-axis as **`CPU Levels`**. So, the drawing is more understandable for us.
- **`title()`**: It adds a header to the graphic. For example, we can write **`CPU Level Measure Drawing`** to define the title of the plot figure. The **`xlabel`**, **`ylabel`**, and title functions are informational and optional.
- **`show()`**: We plot the data with the **`plot`** function, but it will not show the output of the drawing; we should call the **`show`** function to see the drawing as an output.
- **`figure()`**: We can modify the drawing window specifications with the **`figure`** function. We need to write parameters, such as **`figsize`** to change the size of the plot window and **`facecolor`** to change the color of the window. There are also other parameters to customize the graphic window; you can check the source code of the **`figure`** function or the official documentation of the **`matplotlib`** module.

In the following code, we import the **`pyplot`** function from the **`matplotlib`** module and assign it as **`plt`**. After that, we create two variables: **`a`** and **`b`** lists. Each have five items. The item count must be matched to draw the graphic, and the data type of items can differ. After that, we call the **`plot`** function with the X-axis as the **`a`** variable and the Y-axis as the **`b`** variable. So, the code draws the data, but we want to see the output. We call the **`show`** function to open a new window and its plot.

```
import matplotlib.pyplot as plt
a = [1,2,3,4,5]
b = [10,20,30,40,50]
plt.plot(a, b)
plt.show()
```

In *Figure 6.1*, a new window is opened, and the simple graphics are drawn according to the data we provided in the code. We can zoom in on some part of the drawing, save it as a file, and move on both the X-axis and the Y-axis.

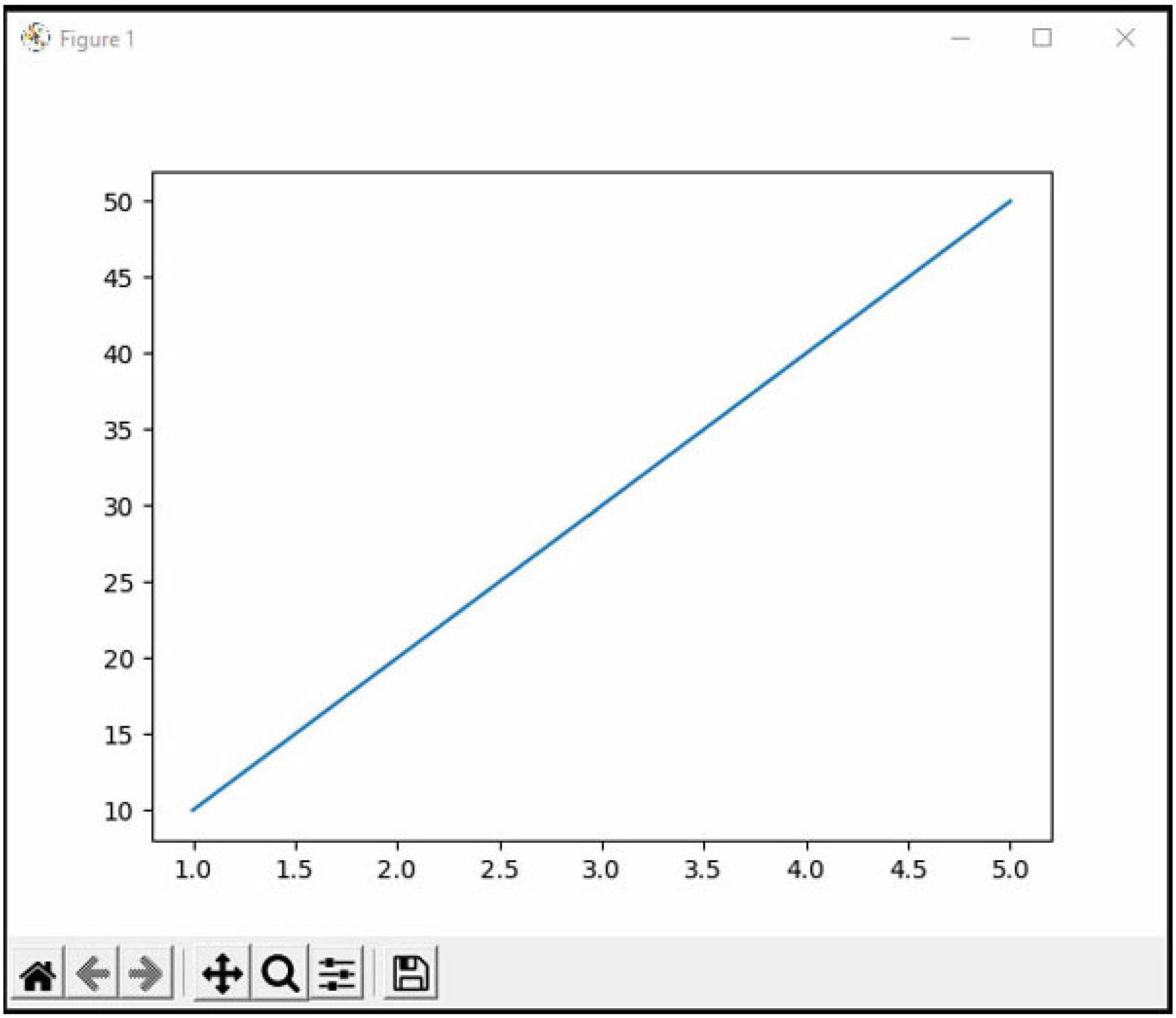


Figure 6.1: *Drawing Sample of Matplotlib Module*

In *Example 6.12*, we can add additional features to make the drawing better. We call the **`figure`** function and write the **`figsize`** as 8 to 8 dimensions. So, the output will be in a larger window. We also add the **`facecolor`** parameter with the color code in a string, so we change the color of the window, and it's white by default.

We add color to the plot as **Red** and assign it to the **color** parameter, and it draws the plot in red instead of blue, the default color.

We also add a description with **xlabel**, **ylabel**, and **title** to the X-axis, Y-axis, and the head of the graphic, and we call the **grid** function by writing **True** as its value. So, it adds a grid to the drawing. Finally, the code displays the plot's output with the **show** function.

Example 6.12: Draw a graphic with Matplotlib

```
import matplotlib.pyplot as plt
a = [1,2,3,4,5]
b = [10,20,30,40,50]
plt.figure(figsize=(8,8), facecolor="#FFCEB4")
plt.plot(a, b, color="Red")
plt.xlabel("Value of 'a'")
plt.ylabel("Value of 'b'")
plt.title("Chart of 'a' and 'b' Values")
plt.grid(True)
plt.show()
```

When we execute the script, we can see the changes in *Figure 6.2*. We change the color of the window and the **plot**, add X-axis and Y-axis headers and drawing headers, add a grid and change the drawing window size.

There are plenty of options to customize the drawing with the **matplotlib** module. The graphic is created with dots if we replace the **plot** function with the **scatter** function. There will be no line in the drawing.

```
plt.scatter(a, b, color="Red")
```

There are other options like **bar**, **stem**, **step**, **fill_between**, and more. You can see the difference when you call these functions and execute the code to draw the plot.

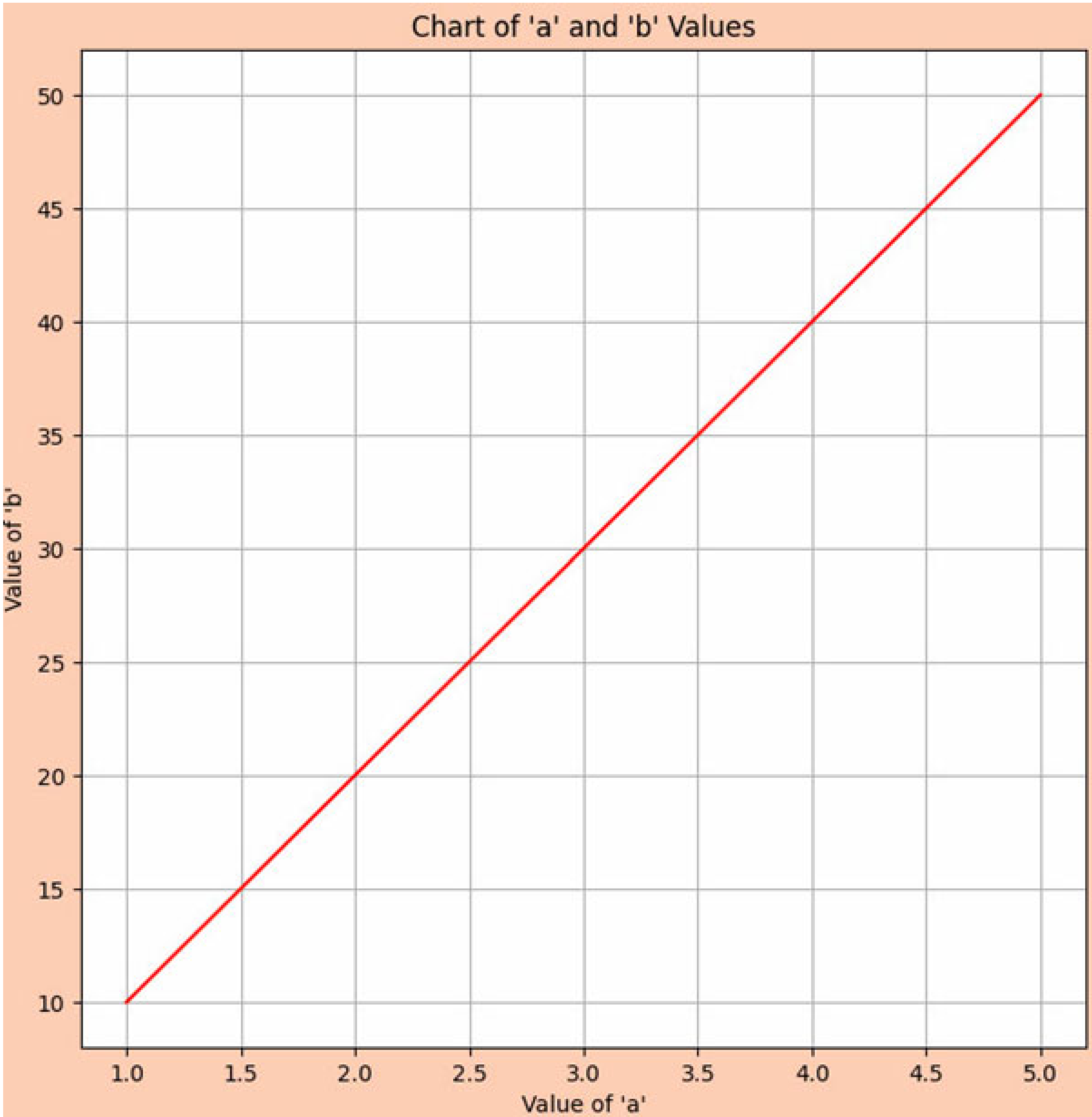


Figure 6.2: *Drawing Advanced Usage of Matplotlib Module*

We can also add two drawings in a single window. We write plot functions twice and call the **`show`** command at the end, and we can increase the plot count by more than two. We need this feature when we get the inbound and outbound data and draw it in the same graphics.

In the following code, we create the **`c`** and **`d`** variables and plot both. We also change the color of both plots so that we divide both drawings smoothly. Refer to *Figure 6.3*:

```
import matplotlib.pyplot as plt
```

```
a = [1,2,3,4,5]
b = [10,20,30,40,50]
c = [6,8,10,12,14]
d = [30,50,23,64,72]
plt.plot(a, b, color="Red")
plt.plot(c, d, color="Blue")
plt.show()
```

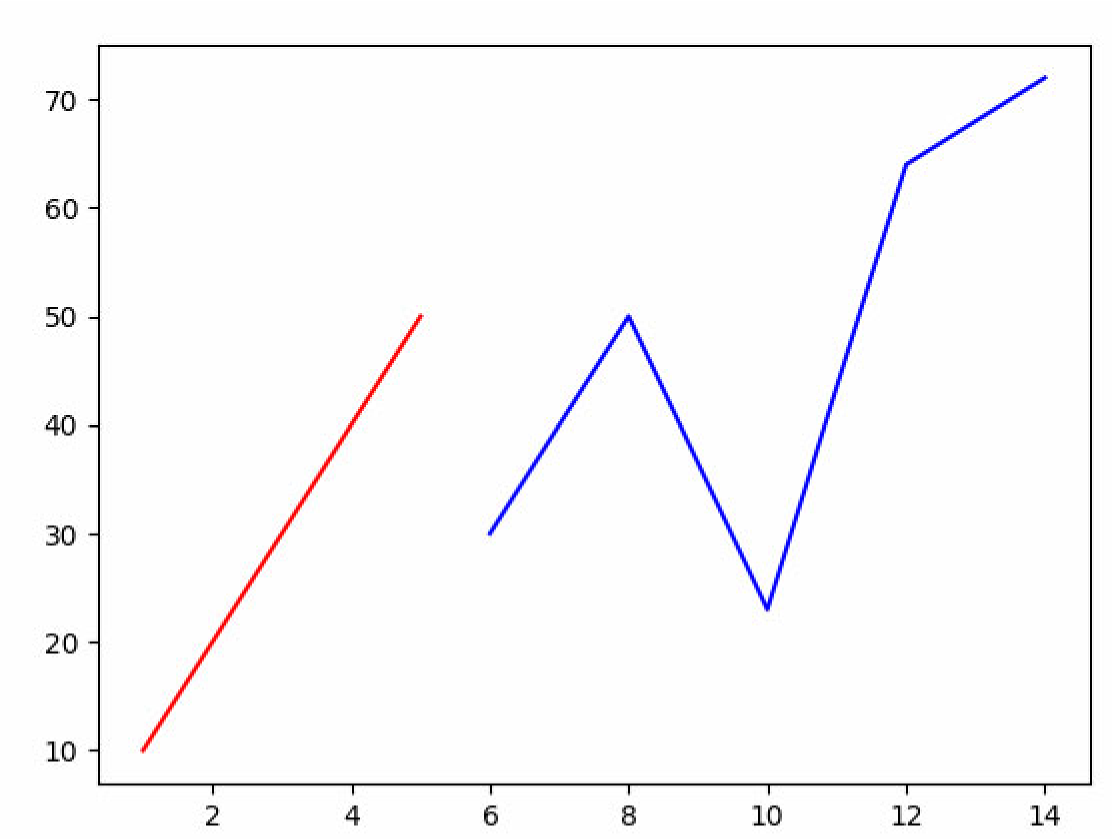


Figure 6.3: *Drawing of Two Plots in a Single Window*

Plotting CPU levels

In *Example 6.13*, we periodically collect CPU levels from a Cisco device and draw a graphic. The X-axis is the local PC's current time in the `hh:mm:ss` format, and the Y-axis is the CPU levels.

1. We import all necessary modules to execute the script.

```
from matplotlib import pyplot as plt
import re
```

```
from netmiko import Netmiko
from time import sleep
from datetime import datetime
```

2. We create a **host** variable with information to log in to a device with the **netmiko** module. We create three variables: **count** is used to denote the number of times we run the command to get the CPU levels. In the example, it's **7**. So, we execute the same command seven times. The value of the **delay** variable is **3**. After sending the command, we wait for 3 seconds to see the CPU changes efficiently. We also create a command variable called **show processes cpu** to get the CPU logs from the device each time.

   We also create two empty lists that we use in the loop and append all CPU and time data in them.

```
host = {"host": "10.10.10.1", "username":"admin",
"password":"cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
count = 7
delay = 3
command = "show processes cpu"
cpu_levels = []
time_list = []
```

3. After that, we log in to the device with the **netmiko** function and create a loop. We use range, showing how often we collect the same data from the device.

```
net_connect = Netmiko(**host)
for i in range(1,count):
 print(f"Get CPU levels count: {i}")
```

4. Inside the loop, we send show command. After that, we got the current time in the local PC and appended it to the **time_list** variable.

```
  output = net_connect.send_command(command)
  time = datetime.now().strftime("%H:%M:%S")
  time_list.append(time)
```

5. We add delay in the code to wait for 3 seconds with the **delay** variable. After that, we collect the CPU level value with the **findall** function from the **RE** module. Loop finishes after we print the item in the **cpu_data**.

```
 sleep(delay)
cpu_data = re.findall("CPU utilization for five seconds:
(\d+)%/", output)
 cpu_levels.append(int(cpu_data[0]))
 print("CPU Level: ",cpu_data[0])
```

6. Outside the loop, we plot two lists named **time_list** and **cpu_levels** with the labeling and adding grid.

```
plt.plot(time_list, cpu_levels)
plt.xlabel("Time")
plt.ylabel("CPU Levels in %")
plt.grid(True)
plt.show()
```

Example 6.13: Collect and draw CPU levels of a router

```
from matplotlib import pyplot as plt
import re
from netmiko import Netmiko
from time import sleep
from datetime import datetime
host = {"host": "10.10.10.1", "username":"admin",
"password":"cisco", "device_type": "cisco_ios",
"global_delay_factor": 0.1}
count = 7
delay = 3
command = "show processes cpu"
cpu_levels = []
time_list = []
net_connect = Netmiko(**host)
for i in range(1,count):
  print(f"Get CPU levels count: {i}")
  output = net_connect.send_command(command)
  time = datetime.now().strftime("%H:%M:%S")
  time_list.append(time)
  sleep(delay)
  cpu_data = re.findall("CPU utilization for five seconds:
  (\d+)%/", output)
  cpu_levels.append(int(cpu_data[0]))
```

```
  print("CPU Level: ",cpu_data[0])
plt.plot(time_list, cpu_levels)
plt.xlabel("Time")
plt.ylabel("CPU Levels in %")
plt.grid(True)
plt.show()
```

When we execute the code, the data's drawing output is as shown in *Figure 6.4*. On the X-axis, there are seven timestamps for each CPU level, and on the Y-axis, there are CPU levels in percentage. We collect the CPU data from a device in around 17-second periods with a 3-second interval.

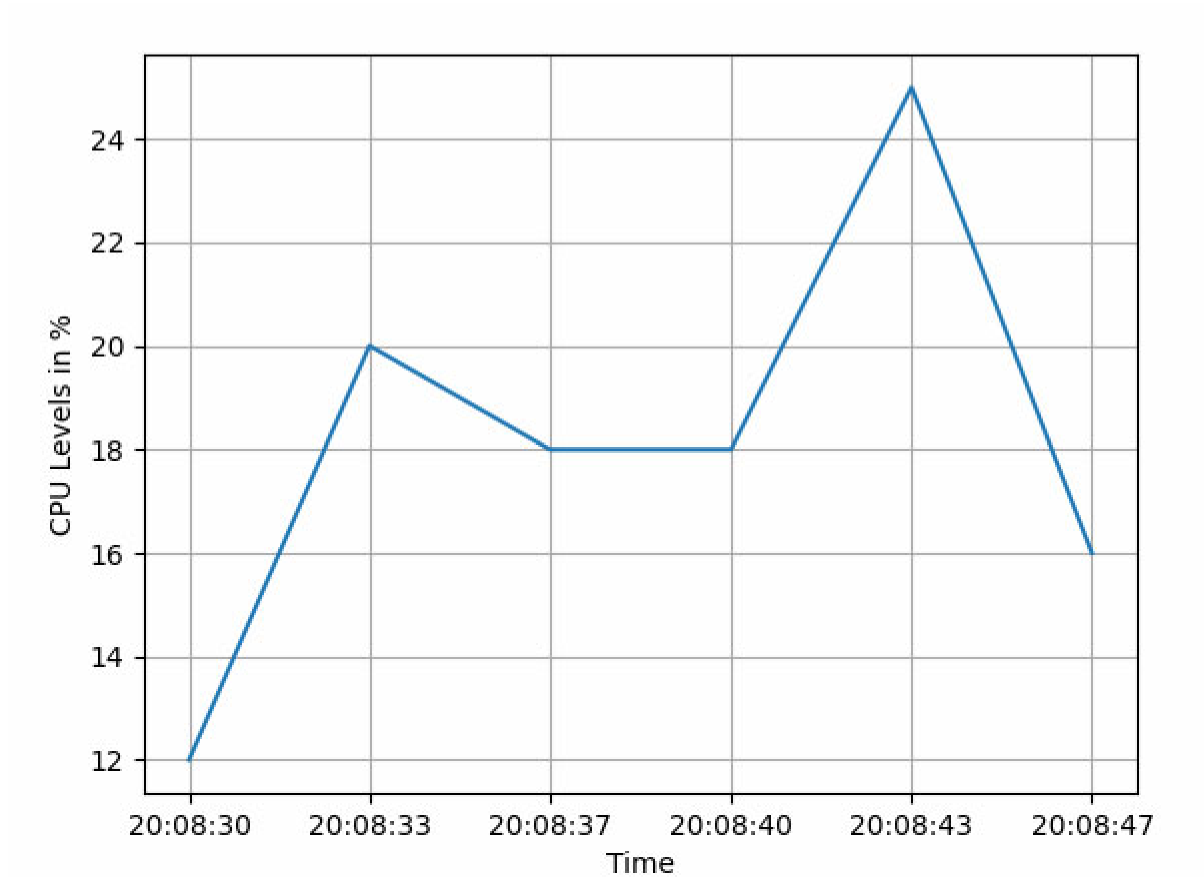


Figure 6.4: *CPU Level Drawing of a Device*

Plotting interface bandwidth

In *Example 6.14*, we collect the interface traffic data in inbound and outbound directions. Then, we plot the data for both traffic usage. If the connected device is a test machine, you only see zero traffic in the inbound and outbound directions. You can use a traffic generator to create traffic in the device.

We change some parts in the script according to *Example 6.13*. We create two empty lists to get the inbound and outbound traffic usage. We execute the **`show interfaces INTERFACE`** command in a Cisco device. In this example, interface information is **`GigabitEthernet0/1`**.

After we log in and run the command, we catch the interface inbound and outbound traffic value with the **`findall`** function and append it to the empty lists. We also get the timestamp from the current time of the local PC. We collected data five times with 3-second intervals.

In the end, we have two different sets of data, so we use the **`plot`** function to draw data at inbound and outbound.

Example 6.14: Collect and draw interface bandwidth values of a router

```
from matplotlib import pyplot as plt
import re
from netmiko import Netmiko
from time import sleep
from datetime import datetime
host = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor":
0.1}
count = 5
delay = 3
interface = "GigabitEthernet0/1"
command = f"show interfaces {interface}"
inbound_rate = []
outbound_rate = []
time_list = []
net_connect = Netmiko(**host)
for i in range(1, count):
  output = net_connect.send_command(command)
  time = datetime.now().strftime("%H:%M:%S")
time_list.append(time)
  input_level = re.findall("5 minute input rate (\d+)", output)
  output_level = re.findall("5 minute output rate (\d+)",
  output)
```

```
  inbound_rate.append(int(input_level[0]))
  outbound_rate.append(int(output_level[0]))
  sleep(delay)
  print("Input Level: ", input_level[0])
  print("Output Level: ", output_level[0])
plt.plot(time_list, inbound_rate, color="blue",
label="Inbound")
plt.plot(time_list, outbound_rate, color="red",
label="Outbound")
plt.xlabel("Time")
plt.ylabel("Interface Levels in MBs")
plt.title(f"Interface Rate of {host['host']} - {interface}")
plt.show()
```

When we execute the code, if there is traffic on the interface, there should be two different plots, like in *Figure 6.5*. We can plot any interface graphic with the data we collect from the device.

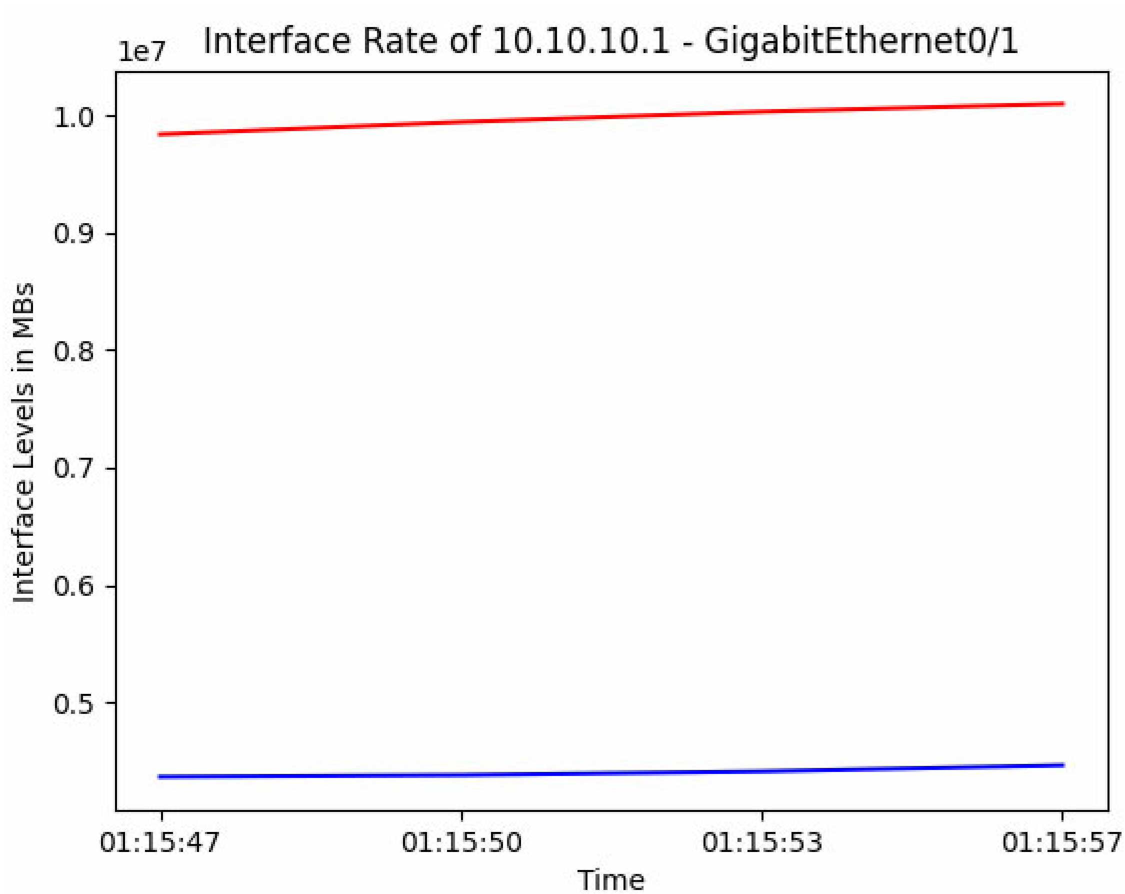


Figure 6.5: *Plot of an Interface Bandwidth Usage*

Conclusion

In this chapter, we learned about file transfers and plotting features in network automation. We used the **`ftplib`**, **`ftpretty`**, **`paramiko`**, **`netmiko`**, and **`nornir`** modules to transfer files. We connected devices and opened file transfer sessions such as FTP, SFP, and SCP to transfer files from a local PC to a remote device and vice versa. We also used the **`matplotlib`** module to draw any data we collected from the device, including the CPU or memory level of the device or interface bandwidth usage of an interface.

The next chapter will focus on maintaining and troubleshooting network issues by creating custom scripts. We will collect the necessary logs, alarms, and other data, and display or send them in various methods.

Multiple choice questions

1. Which protocol cannot support connecting with the paramiko module?
   a. SFTP
   b. SCP
   c. Telnet
   d. SSH
2. Which function does not belong to the matplotlib module?
   a. `grid ( )`
   b. `scatter ( )`
   c. `figure ( )`
   d. `draw ( )`
3. Which code must we write to change the plot color to red and window color to blue?
   a. `plot(x, y, color="Red")`
      `figure(facecolor="Blue")`
   b. `plot(x, y, color="Red", facecolor="Blue")`
   c. `plot(x, y, color="Blue", facecolor="Red")`

d. `plot(x, y, color="Blue")`
   `figure(facecolor="Red")`

Answers

1. b
2. d
3. a

Questions

1. Download the config file from three devices with the **`paramiko`** module by using the **`concurrent`** module.
2. Collect all alarms from a device, get the counts of the **`Minor`**, **`Major`**, **`Critical`** alarms, and draw a bar chart with **`matplotlib`** according to alarm severities.

CHAPTER 7
Maintain and Troubleshoot Network Issues

This chapter will focus on network device upgrades, collecting alarms, SNMP communication, email notifications, and reachability test for network and system devices. We will create scripts to troubleshoot the network in basic methods and secure it by identifying the alarms by their severity. We will also collect logs and share them with others by mailing them to Python's built-in modules.

Structure

In this chapter, we will cover the following topics:

- Upgrade network devices
- Alert alarms in devices
- Collect logs with SNMP
- Send logs via email
- Reachability test to network devices
    - Ping test script
    - Traceroute test script

Objectives

We will upgrade devices by uploading software files, setting the boot file after rebooting the device, and then rebooting the device, which are the three steps to upgrading software in various vendors. We will use the **`netmiko`** module to connect and collect the logs from the network devices, like in the previous chapters. We will send these logs or device configurations using the email and the SMTP modules. We will use the **`subprocess`** module to execute the **`ping`** and **`tracert`** commands in Windows devices to make reachability tests in the network.

Upgrade network devices

When we upgrade a Cisco device, there are three steps to take. In the first step, we need to upload the new software file to the device. Then, we need to set the boot file, and finally, we need to reboot the device. These steps are similar to those that need to be followed for other vendors like Juniper, Huawei, or Nokia; only the commands differ.

We already have many scripts about transferring files in *Chapter 6, File Transfer and Plotting*. We must write a script to set the new software file and the reboot process.

In *Example 7.1*, we write a script to transfer a new software file to the Cisco device, set the latest software file, and reboot. In more advanced usage, we need to verify whether the new software file size is identical to the local PC file size. We need to save the configuration file and back it up to the local PC. After that, we are ready to upgrade our device.

1. We import the **`netmiko`**, **`re`**, and **`os`** modules to use in this script.
   ```
   from netmiko import Netmiko, file_transfer
   from re import findall
   import os
   ```
2. We set device information to upgrade. In this example, we only upgrade one Cisco device. We create three variables: **`filename`** is the latest software file name with its extension on the local PC, **`set_software`** is the command to set the new software file to boot in Cisco Router and change the **`config-register`** value to reload, and **`local_filesize`** is the file size of the latest software on the local PC.
   ```
   device = {"host": "10.10.10.1", "username": "admin",
   "password": "cisco", "device_type": "cisco_ios",
   "global_delay_factor": 0.1 }
   filename = "universalk9.17.08.01.bin"
   set_software = [f"boot system {file_sys}
   {filename}","config-register 0x2102"]
   local_filesize = os.path.getsize(filename)
   ```
3. We connect to the device with netmiko and transfer the software file from our local PC to the remote device with the **`file_transfer`** function, as we did in the previous chapters.
   ```
   net_connect = Netmiko(**device)
   file_transfer(net_connect,
   ```

```
source_file=filename,
dest_file= filename,
direction="put",
)
```

4. We execute the **dir** command to find the new software after the upload finishes. We get the file size with the **findall** function.

```
output = net_connect.send_command(f"dir | include
{filename}")
remote_filesize = findall("\d+",output)
```

When we run the same command in CLI, the second digit item is the file size, for example, **31900**. So, if we write **remote_filesize[1]**, we catch the file size of our new software file.

```
Router-2#dir | include test.txt
280   -rw-       31900  Aug 21 2022 06:45:46
+00:00  test.txt
```

5. We send the **boot** command. This command sets the boot file in the device. Then, we send the **show run** command to find whether the boot command is configured on the device. We write the **findall** function to find whether the command is set on configuration. Finally, we save the device with the **wr** command.

```
net_connect.send_config_set(set_software)
output = net_connect.send_command(f"show run | include
{filename}")
boot_set = findall(set_software[0],output)
net_connect.send_command(f"wr")
```

6. In the final step, we compare local and remote file sizes and **boot** commands in the configuration. If one or two of the conditions do not match, the code passes the **if** statement and disconnects from the device. Otherwise, we continue to **reload** the device, which is the **reboot** command in Cisco devices. After that, the device asks to continue to reload. In that step, we need to push the *Enter* button. In the code, we must write **\n** to press enter or go to the following line.

   But here, we write the **expect_string** parameter. So, the code waits until the **Proceed with reload** line is shown, and then it continues. We can also add timing with **send_command_timing**, and we add delay as 1 second with the **delay_factor** parameter.

```
    if str(local_filesize) == remote_filesize[1] and
    set_software[0] == boot_set[0]:
     print("File is uploaded and set to boot successfully")
    net_connect.send_command("reload", expect_string="Proceed
    with reload")
     net_connect.send_command_timing("\n", delay_factor=1)
    else:
     print("File upload or setting software as boot is failed")
    net_connect.disconnect()
```

Example 7.1: Upgrade a network device with netmiko

```
from netmiko import Netmiko, file_transfer
from re import findall
import os
device = {"host": "10.10.10.1", "username": "admin", "password":
"cisco", "device_type": "cisco_ios", "global_delay_factor": 0.1 }
filename = "universalk9.17.08.01.bin"
set_software = [f"boot system {file_sys}{filename}","config-
register 0x2102"]
local_filesize = os.path.getsize(filename)
net_connect = Netmiko(**device)
file_transfer(net_connect,
      source_file=filename,
      dest_file= filename,
      direction="put",
      )
output = net_connect.send_command(f"dir | include {filename}")
remote_filesize = findall("\d+",output)
net_connect.send_config_set(set_software)
output = net_connect.send_command(f"show run | include
{filename}")
boot_set = findall(set_software[0],output)
net_connect.send_command(f"wr")
if str(local_filesize) == remote_filesize[1] and set_software[0]
== boot_set[0]:
  print("File is uploaded and set to boot successfully")
  net_connect.send_command("reload", expect_string="Proceed with
  reload")
  net_connect.send_command_timing("\n", delay_factor=1)
```

```
else:
  print("File upload or setting software as boot is failed")
net_connect.disconnect()
```

Alert alarms in devices

In *Example 7.2*, we collect alarm information from the Juniper devices with the **`show system alarms`** command. After that, we collect specific data on those alarms, such as alarm time, alarm severity, and alarm description, and save it to an Excel file. We also collect total alarms in the network and divide them according to severity, such as **`Minor`**, **`Major`**, and **`Critical`**, in another sheet or tab in the same Excel file.

1. We import the **`netmiko`**, **`re`**, and **`pandas`** modules.
   ```
   from netmiko import Netmiko
   from re import findall
   from pandas import DataFrame
   ```
2. We have three devices in this example. This time, we log in to the Juniper devices and create empty lists and integers to use in the following code:
   ```
   host = ["10.10.20.1", "10.10.20.2", "10.10.20.3"]
   time_list, severity_list, description_list, ip_list = ([]
   for x in range(4))
   total_minor = total_major = total_critical = 0
   ```
3. We create a **`for`** loop to log in devices in each iteration. After the connection, we execute the show command and collect the output.
   ```
   for ip in host:
   device = {f"host": {ip}, "username": "admin", "password":
   "juniper", "device_type": "juniper", "global_delay_factor":
   0.1 }
    net_connect = Netmiko(**device)
    output = net_connect.send_command("show system alarms")
   ```
4. We collect all devices' total alarm count, **`Minor`**, **`Major`**, and **`Critical`** alarm counts in the network. So, we use **`findall`** to get these data and save it to various variables in the following code. For the **`alarms`** variable, we delete the first four items with the **`del`** function because they are unnecessary lines in the output earlier:
   ```
    alarm_count = findall("(\d+) alarms currently
    active",output)
   ```

```
alarms = split("\n",output)
del alarms[0:4]
total_alarms = total_alarms + len(alarms)
minor_alarms = findall("Minor",output)
major_alarms = findall("Major",output)
critical_alarms = findall("Critical",output)
total_minor = total_minor + len(minor_alarms)
total_major = total_major + len(major_alarms)
total_critical = total_critical + len(critical_alarms)
```

5. We create an inner or second **for** loop in the following code. In the previous code, we collect the total count of the alarms. This time, we collect the specific data in each device, like alarm occurs time, severity, and the description with the **findall** function. Afterward, we append this data in each device to the lists to use them with the **dataframe** function because we add all of them to an Excel file at the end.

```
for alarm_item in alarms:
 time = findall("\d+-\d+\d+ \d+:\d+:\d+ UTC", alarm_item)
 severity = findall("Minor|Major|Critical", alarm_item)
description = findall("\d+-\d+\d+ \d+:\d+:\d+ UTC\s+\w+\s+
(.*)", alarm_item)
 ip_list.append(f"{ip}")
 time_list.append(time[0])
 severity_list.append(severity[0])
 description_list.append(description[0])
```

6. We collect the alarms with description, time, and severity value. Finally, we need to create an Excel file and write the items with the **dataframe** function from the **pandas** module. We need to create two tabs in Excel, so we need to use the **ExcelWriter** function. We open the Excel file to fill it. And we create two variables: **df1** and **df2**. We call the **dataframe** function and fill this function like in the previous chapters. After that, we write these values to the same Excel with the **to_excel** function. We call the **writer** function inside the **to_excel** function for both the **df1** and **df2** variables. We also write the **sheet_name** or Excel tab, such as **Summary** and **Alarms**.

```
with pandas.ExcelWriter('Alarm List.xlsx') as writer:
 df1 = pandas.DataFrame({"Alarm Count":
 [total_alarms],"Minor": [total_minor],"Major":
 [total_major],"Critical": [total_critical]})
```

```
        df2 = pandas.DataFrame({"Device IP": ip_list, "Time":
        time_list, "Severity": severity_list, "Description":
        description_list})
        df1.to_excel(writer, sheet_name="Summary", index=False)
        df2.to_excel(writer, sheet_name="Alarms", index=False)
```

Example 7.2: Collect alarm information from devices and summarize

```
from netmiko import Netmiko
from re import findall
from pandas import DataFrame
host = ["10.10.20.1", "10.10.20.2", "10.10.20.3"]
time_list, severity_list, description_list, ip_list = ([] for x in
range(4))
total_minor = total_major = total_critical = 0
for ip in host:
  device = {f"host": {ip}, "username": "admin", "password":
  "juniper", "device_type": "juniper", "global_delay_factor": 0.1
  }
  net_connect = Netmiko(**device)
  output = net_connect.send_command("show system alarms")
  alarm_count = findall("(\d+) alarms currently active",output)
  alarms = split("\n",output)
  del alarms[0:4]
  total_alarms = total_alarms + len(alarms)
  minor_alarms = findall("Minor",output)
  major_alarms = findall("Major",output)
  critical_alarms = findall("Critical",output)
  total_minor = total_minor + len(minor_alarms)
  total_major = total_major + len(major_alarms)
  total_critical = total_critical + len(critical_alarms)
  for alarm_item in alarms:
   time = findall("\d+-\d+\d+ \d+:\d+:\d+ UTC", alarm_item)
   severity = findall("Minor|Major|Critical", alarm_item)
   description = findall("\d+-\d+\d+ \d+:\d+:\d+ UTC\s+\w+\s+
   (.*)", alarm_item)
   ip_list.append(f"{ip}")
   time_list.append(time[0])
   severity_list.append(severity[0])
   description_list.append(description[0])
```

```
with pandas.ExcelWriter('Alarm List.xlsx') as writer:
  df1 = pandas.DataFrame({"Alarm Count":
  [total_alarms],"Minor": [total_minor],"Major":
  [total_major],"Critical": [total_critical]})
  df2 = pandas.DataFrame({"Device IP": ip_list, "Time":
  time_list, "Severity": severity_list, "Description":
  description_list})
  df1.to_excel(writer, sheet_name="Summary", index=False)
  df2.to_excel(writer, sheet_name="Alarms", index=False)
```

When we run **show system alarms** in the Juniper devices, the output is similar to the following. There is an empty line; after that, there are total active alarm counts and the alarm titles. Finally, alarms are listed with their details. We divide each part in *Example 7.2*. If there is no Juniper device in your lab, you can try with other vendors to change the command on the device. It would be best if you also modified the **findall** function according to that output. On the other hand, you can save each Router's output in a text file in the following output and open it in the same script by modifying some parts:

```
Junos_Router-1:
Junos_Router-1> show system alarms
4 alarms currently active
Alarm time               Class     Description
2022-08-02 15:00:00 UTC  Minor     IPsec VPN tunneling usage
requires a license
2022-08-24 15:00:00 UTC  Major     Rescue configuration is not
sent
2022-08-25 15:00:00 UTC  Major     /root partition usage crossed
critical threshold
2022-08-12 15:00:00 UTC  Critical  PCI Corrected error on dev
0000:00:01
Junos_Router-2:
Junos_Router-2> show system alarms
4 alarms currently active
Alarm time               Class     Description
2022-07-24 16:00:00 UTC  Minor     IPsec VPN tunneling usage
requires a license
2022-07-24 16:00:00 UTC  Major     Rescue configuration is not
sent
```

```
2022-07-05 16:00:00 UTC  Critical  FPC 8 internal link errors
detected
2022-07-16 16:00:00 UTC  Minor    NSD 12 channel error on physical
interfaces
Junos_Router-3:
Junos_Router-3> show system alarms
5 alarms currently active
Alarm time               Class     Description
2022-07-23 17:00:00 UTC  Minor     IPsec VPN tunneling usage
requires a license
2022-08-02 17:00:00 UTC  Major     Rescue configuration is not
sent
2022-07-11 17:00:00 UTC  Critical  Side Fan Tray 7 Failure
2022-05-11 17:00:00 UTC  Minor     Side Fan Tray 7 Overspeed
2022-07-16 16:00:00 UTC  Minor    NSD 12 channel error on physical
interfaces
```

When we execute the code, it creates an Excel file in the same directory as our code. In the first sheet or tab in this Excel file, which is **`Summary`**, we can see the description of **`Alarm Count`**, **`Minor`**, **`Major`**, and **`Critical`** with their values. Refer to *Figure 7.1*:

Alarm Count	Minor	Major	Critical
13	6	4	3

Figure 7.1: *Output of the "Summary" Section in Excel*

In the next tab, **`Alarms`**, we divide each item in the output according to time, severity, and description. We also write device management IP addresses to define alarms belonging. So, we can easily filter any information in this Excel file. Refer to *Figure 7.2*:

Device IP	Time	Severity	Description
10.10.20.1	08-02 15:00:00 UTC	Minor	IPsec VPN tunneling usage requires a license
10.10.20.1	08-24 15:00:00 UTC	Major	Rescue configuration is not sent
10.10.20.1	08-25 15:00:00 UTC	Major	/root partition usage crossed critical threshold
10.10.20.1	08-12 15:00:00 UTC	Critical	PCI Corrected error on dev 0000:00:01
10.10.20.2	07-24 16:00:00 UTC	Minor	IPsec VPN tunneling usage requires a license
10.10.20.2	07-24 16:00:00 UTC	Major	Rescue configuration is not sent
10.10.20.2	07-05 16:00:00 UTC	Critical	FPC 8 internal link errors detected
10.10.20.2	07-16 16:00:00 UTC	Minor	NSD l2 channel error on physical interfaces
10.10.20.3	07-23 17:00:00 UTC	Minor	IPsec VPN tunneling usage requires a license
10.10.20.3	08-02 17:00:00 UTC	Major	Rescue configuration is not sent
10.10.20.3	07-11 17:00:00 UTC	Critical	Side Fan Tray 7 Failure
10.10.20.3	05-11 17:00:00 UTC	Minor	Side Fan Tray 7 Overspeed
10.10.20.3	07-16 16:00:00 UTC	Minor	NSD l2 channel error on physical interfaces

Figure 7.2: *Output of the "Alarms" Section in Excel*

Collect logs with SNMP

Simple Network Management Protocol (**SNMP**) is one of the essential protocols in networking. It's a communication protocol to share device information. NMS tools use SNMP to get data from the devices and display it in the tool. There are three versions of SNMP: versions 1, 2, and 3. SNMPv1 has fragile security protection, and SNMPv2 has more security than SNMPv1; however, the most secure version is SNMPv3, which has data encryption. It has an authentication process to to prevent unauthorized connections.

We can collect data like CPU and memory usage, device uptime, **Open Shortest Path First** (**OSPF**) neighbors, and interface status, which can be UP/DOWN.

Devices have **Management Information Base** (**MIB**), which is an object that keeps the data from the local device. MIB is a file that stores the information collected from the device. So, the SNMP manager uses MIB files to get data from any device.

Various objects are inside the MIB, identified by **Object Identifier** (**OID**). NMS requests the object's value from the agent with these OIDs. OID is a numerical address to identify the objects in the MIB hierarchy. The `1.3.6.1.2.1.25.1.1.0` OID number is used to get the device uptime. When the NMS or the monitoring tool sends this OID to a device, the device sends the device uptime information back to the agency. So, the tool gets all the data with different OIDs from the network device and creates a database.

Many MIB information or OIDs are the same and generic for different vendors, but vendor specific MIBs can also be downloaded from the vendor's official websites.

We have a third-party SNMP module in Python, `pysnmp`, which is a mature library to communicate with network and system devices by the SNMP protocol. We must install it with the `pip install pysnmp` command in the terminal.

We have a third-party SNMP module in Python, `pysnmp`, which is a mature library to communicate with network and system devices by the SNMP protocol. We must install it with the `pip install pysnmp` command in the terminal.

We must enable the SNMP feature to collect the data from the network device by the Python script. We need to configure community value with options like the `read-only` or `read-write` parameters. We configure the Cisco device with the community `public`. This command in Cisco devices enables SNMPv1 and SNMPv2. To use SNMPv3, we must add other commands: the `authentication` and `encryption` commands.

```
Router-1#configure terminal
Router-1(config)#snmp-server community public ro
```

In *Example 7.3*, we collect the free memory data from `Router-1` and display it in the output in bytes. We use the `pysnmp` module to collect memory data from the Cisco device.

1. We import the `pysnmp` module. We use many functions inside the `hlapi` function in the `pysnmp`, so we import all the functions inside it with the `*` character.

   ```
   from pysnmp.hlapi import *
   ```

2. We define the `host` variable for the management IP address of the device, the `snmp_community` variable for the community configuration on the device, which is `public`, and finally, the `snmp_oid` variable to get the free memory data from the device. The `1.3.6.1.4.1.9.2.1.8.0` oid number collects the device's free memory. You can search for generic or vendor-specific mib files and oids on the internet to get the complete list.

   ```
   host = "10.10.10.1"
   snmp_community = "public"
   snmp_oid = "1.3.6.1.4.1.9.2.1.8.0"
   ```

3. In the following code, we use the **pysnmp** functions. We have the **errorIndication**, **errorStatus**, **errorIndex**, and **varBinds** variables, which are equal to the **next** function in **pysnmp** with the **getCmd** function in it.

   ```
   errorIndication, errorStatus, errorIndex, varBinds =
   next(getCmd()
   ```

   Inside the **getCmd** function, we execute the **SnmpEngine** class instance to start the SNMP feature. All the SNMP operations are involved in this class. We have class instances like **CommunityData**, in which we write the community configured on the device. We can also opt to add the **mpModel** parameter. If the value is zero, it uses SNMPv1 to communicate with the device. If it's one, it uses SNMPv2, the default value. To communicate with the device in SNMPv3, we must write the **UsmUserData** class instance.

   After that, we write the **UdpTransportTarget** object to connect a device via SNMP. We write the host and port information as 161, which is the SNMP protocol's port number.

   We also need to write the **ContextData** object. The SNMP context is a message header in the SNMP protocol that finds the specific MIB. So, we must initialize this object to get the data from the device.

   Finally, we call the **ObjectType** class instance to get the oid number with the **ObjectIdentity** object.

   ```
   errorIndication, errorStatus, errorIndex, varBinds = next(
    getCmd(SnmpEngine(),
     CommunityData(snmp_community, mpModel=1),
     UdpTransportTarget((host, 161)),
     ContextData(),
     ObjectType(ObjectIdentity(snmp_oid)), ) )
   ```

4. After we get the data from the device, we need to display it in the output. We create a **for** loop to get the **oid** and **val** variables from **varBinds**. We print each value with the **prettyPrint()** function to display it in human-readable mode.

   ```
   for oid, val in varBinds:
    print(oid.prettyPrint()," - ", val.prettyPrint())
   ```

Example 7.3: Collect device information with SNMP

```
from pysnmp.hlapi import *
host = "10.10.10.1"
```

```
snmp_community = "public"
snmp_oid = "1.3.6.1.4.1.9.2.1.8.0"
errorIndication, errorStatus, errorIndex, varBinds = next(
  getCmd(SnmpEngine(),
     CommunityData(snmp_community, mpModel=1),
     UdpTransportTarget((host, 161)),
     ContextData(),
     ObjectType(ObjectIdentity(snmp_oid)),     )      )
for oid, val in varBinds:
  print(oid.prettyPrint()," - ", val.prettyPrint())
```

`1.3.6.1.4.1.9.2.2.1.1.20.1` gets the interface status, such as UP/DOWN state. If we want to get the first two interface statuses, we write `1.3.6.1.4.1.9.2.2.1.1.20.1` and `1.3.6.1.4.1.9.2.2.1.1.20.2`. We use these OID numbers in *Example 7.4*. We can also add another loop to collect data from multiple devices.

Example 7.4: Collect multiple OID data with SNMP

```
from pysnmp.hlapi import *
host = "10.10.10.1"
snmp_community = "public"
snmp_oid =
["1.3.6.1.4.1.9.2.2.1.1.20.1","1.3.6.1.4.1.9.2.2.1.1.20.2"]
for id in snmp_oid:
  errorIndication, errorStatus, errorIndex, varBinds = next(
   getCmd(SnmpEngine(),
      CommunityData(snmp_community, mpModel=1),
      UdpTransportTarget((host, 161)),
      ContextData(),
      ObjectType(ObjectIdentity(id)),
      )
  )
  for oid, val in varBinds:
   print(oid.prettyPrint()," - ", val.prettyPrint())
```

Output:

```
SNMPv2-SMI::enterprises.9.2.2.1.1.20.1  -  up
SNMPv2-SMI::enterprises.9.2.2.1.1.20.2  -  administratively down
```

From CLI:

```
Router-1#show ip int br
```

```
Interface              IP-Address   OK?   Method
Status                  Protocol
GigabitEthernet0/0  10.10.10.1  YES  NVRAM  up
up
GigabitEthernet0/1  unassigned  YES  unset  administratively
down  down
```

We can get a lot of data from devices with OID:

```
"1.3.6.1.4.1.9.2.1.3.0" - Hostname Information
"1.3.6.1.4.1.9.2.1.58.0" - CPU Usage
"1.3.6.1.4.1.9.2.1.4.0" - Domain name
```

We can also use the object name instead of OID. We change the parameters inside `ObjectIdentity`. We write the MIB name, object as **sysName** and zero. `sysName` gets the hostname of the device with its domain name.

```
ObjectType(ObjectIdentity("SNMPv2-MIB", "sysName", 0)),
```

Output: `SNMPv2-MIB::sysName.0  -  Router-1.networkautomation`

The Python files are inside the `venv` directory, located in the same folder as our project code:

`PROJECT_FILE\venv\Lib\site-packages\pysnmp\smi\mibs`

We can get a lot of data from devices with OID, such as the following:

- `sysDescr`: Gets system information, including version and device model.
- `snmpInPkts`: Gets inbound SNMP packet count. If you execute the code, it will increase each time because we send SNMP requests to the device.
- `sysUpTime`: Gets the system uptime in hundreds of seconds.

We cannot directly use the MIB file with the `pysnmp` module; we must convert it to a Python file. There are options to convert, like the `mibdump.py` Python code found on the internet or the local `pysnmp` module. Using Ubuntu, you can install the `libsmi2pysnmp` package to convert it.

On the following website, there are many converted MIB files to download. You can download the MIB files and paste them to the `PROJECT_FILE\venv\Lib\site-packages\pysnmp\smi\mibs` directory. Then, you can use any of them.

https://pypi.org/project/pysmi/#files

For example, we can use `OSPF-MIB` with the `ospfRouterId` object. It collects `OSPF router-id` from the device. If you set the OSPF `router-id` of the device, you can get output as `router-id` data.

```
ObjectType(ObjectIdentity("OSPF-MIB", "ospfRouterId", 0)).
```

Output: `OSPF-MIB::ospfRouterId.0 - 10.10.10.1`

Send logs via email

We can send emails with Python's built-in modules, such as **`email`** and **`smtplib`**. In this part, we use the Gmail account to send emails to any email address. We can send device alarms, logs, and configurations with emails and add attachments to the emails.

When we use a Gmail account to send emails via Python script, we need a 16-digit password that's different from our Google account password. We must follow the given steps to get this password:

1. Enter the following website and log in to the Gmail account.

   https://myaccount.google.com/

2. Click on **`Security`** on the opening page, as shown in *Figure 7.3*:

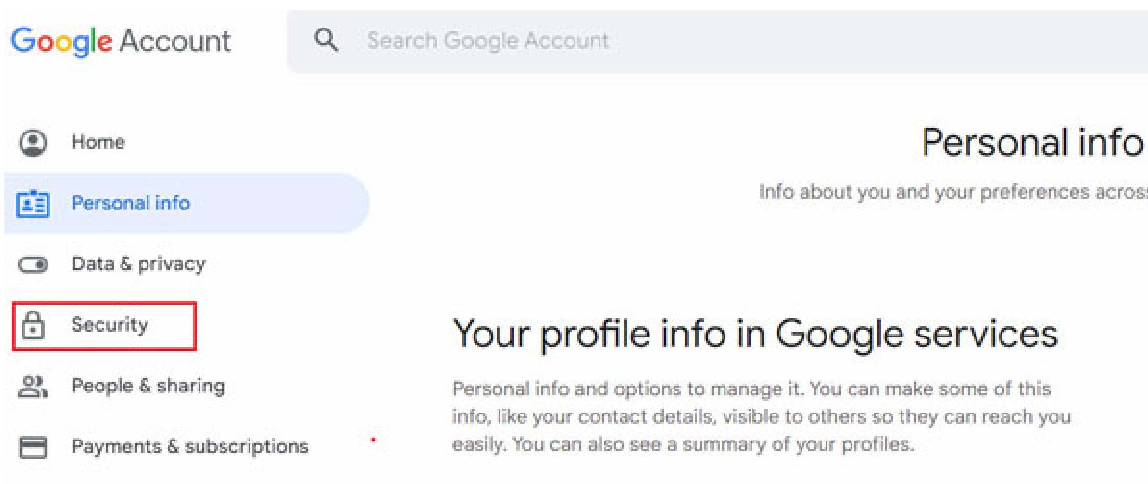


Figure 7.3: *Creating Password in Gmail Step-2*

3. As shown in *Figure 7.4*, **`2-Step Verification`** must be enabled. If it's disabled, you must enable it before entering the **`App passwords`** section.

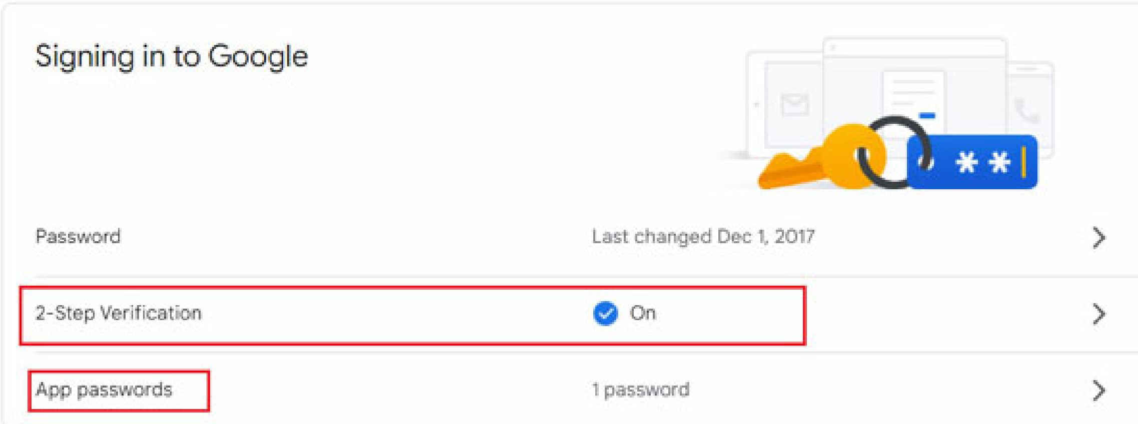


Figure 7.4: *Creating Password in Gmail Step-3*

4. As shown in [*Figure 7.5*](#), you need to click on the `Select app` button and write the app name. You can write anything to be understandable for you later. After that, the `GENERATE` button becomes blue and is activated; you can click on it to create a password.

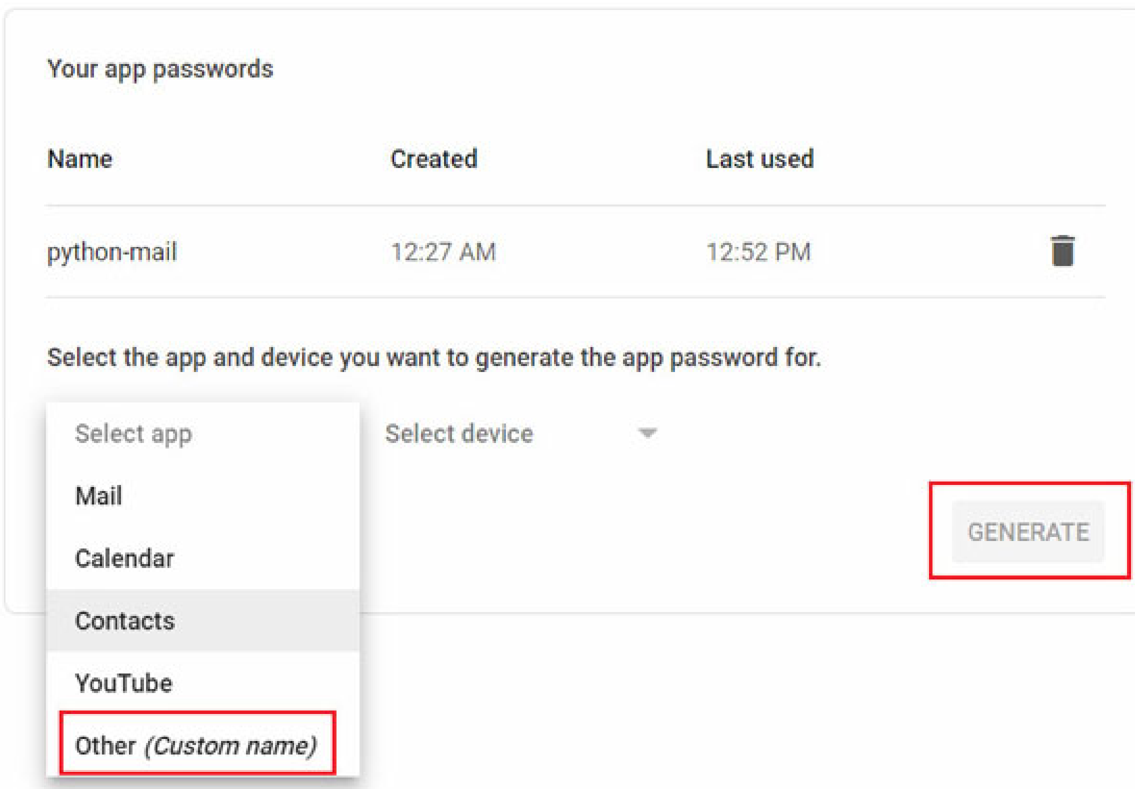


Figure 7.5: *Creating Password in Gmail Step-4*

5. [*Figure 7.6*](#) shows a new popup that opens in the web browser. You must copy the 16-digit password in a safe place and not share it with anyone.

It’s a unique password that is created for your account and app. The password creation finishes with this step, and we can continue to write the scripts in Python.

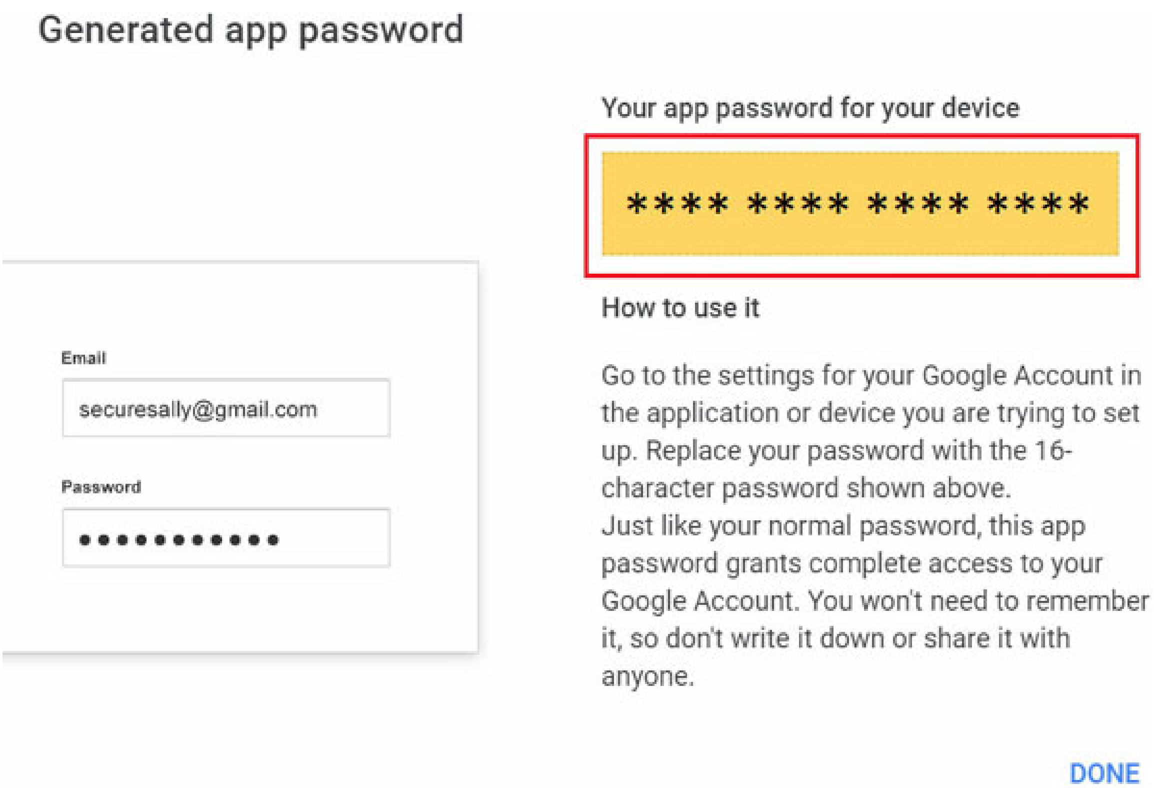


Figure 7.6: *Creating Password in Gmail Step-5*

6. We import the **`smtplib`** module and the **`message`** class from the **`email`** module.

```
import smtplib
from email import message
```

7. We create various variables. We add sender mail to **mail_from** and password to **mail_password** variable. We get the **mail_password** value from the previous steps in Gmail. It’s 16-digit password which is not a Gmail password.We also add the receiver as **`mail_to`**; optionally, we add cc and bcc. And in the final two variables, we add subject and content variables. All these variables’ values are in string values.

```
mail_from = "example@gmail.com"
mail_password = "16-DIGIT-PASSWORD"
mail_to = "example@gmail.com"
mail_to_cc = "example@gmail.com"
mail_to_bcc = "example@gmail.com"
```

```
mail_subject = "Test Email"
mail_content = "Hi,\nThis is a test email"
```

8. We call the **EmailMessage** function from the message class and assign the **send** variable; we add the email details to it. We use the **add_header** function to add email subject, sender, and receiver address information. We write **From** as the sender, **To** as a receiver, **Cc** as the cc-receiver, **Bcc** as the bcc-receiver, and **Subject** as the mail subject.

```
send = message.EmailMessage()
send.add_header("From", mail_from)
send.add_header("To", mail_to)
send.add_header("Cc", mail_to_cc)
send.add_header("Bcc", mail_to_bcc)
send.add_header("Subject", mail_subject)
send.set_content(mail_content)
```

9. At the end, we execute the **SMTP_SSL** function from the **smtplib** module. We use the **smtp.gmail.com** server with port **465**, which is the SMTP protocol port number. We log in to our account with the sender's email and password information with the **login** function. Finally, we send an email by the **sendmail** function by entering email details like sender, receiver, and content of the mail.

```
with smtplib.SMTP_SSL("smtp.gmail.com", 465) as smtp:
  smtp.login(mail_from, mail_password)
  smtp.sendmail(mail_from, mail_to, send.as_string())
```

Example 7.5: Sending email via Gmail

```
import smtplib
from email import message
mail_from = "example@gmail.com"       #The value must be Gmail 
address
mail_password = "16-DIGIT-PASSWORD"
mail_to = "example@gmail.com"
mail_to_cc = "example@gmail.com"
mail_to_bcc = "example@gmail.com"
mail_subject = "Test Email"
mail_content = "Hi,\nThis is a test email"
send = message.EmailMessage()
send.add_header("From", mail_from)
send.add_header("To", mail_to)
```

```
send.add_header("Cc", mail_to_cc)
send.add_header("Bcc", mail_to_bcc)
send.add_header("Subject", mail_subject)
send.set_content(mail_content)
with smtplib.SMTP_SSL("smtp.gmail.com", 465) as smtp:
  smtp.login(mail_from, mail_password)
  smtp.sendmail(mail_from, mail_to, send.as_string())
```

We can also add attachments to these emails. We can attach various file formats, like text-based files, pictures, and more. First, we need to open the source file with the `open` function in reading and binary mode as `rb`. Then, we need to read the file and assign it to a variable. In the following line, we need to use `mime_type` and `encoding` variables and assign them to the `mimetypes.guess_type` function with the filename. After that, we call the **add_attachment** function. We write **attached_file**, **maintype**, **subtype**, and **filename** variables. Finally, we add the `filename` parameter, and its value is the filename of the source file. We use the following code in *Example 7.6* with the `for` loop to add multiple files in the attachment.

```
import mimetypes
with open("test.txt", "rb") as r:
  attached_file = r.read()
mime_type, encoding = mimetypes.guess_type(filename)
send.add_attachment(      attached_file,
maintype=mime_type.split("/")[0], subtype=mime_type.split("/")[1],
filename=filename
)
```

In *Example 7.6*, we collect three Cisco router configurations with the `netmiko` module and save them to the local directory. After that, we send these three configuration files from our mail address to another email address.

We create a function to collect configurations of the devices and save them to the text files with different namings. After that, we call the function in the script. In the rest of the code, we only add the attachment part in the loop to add three files in the same mail content and send the mail to the receiver.

Example 7.6: Sending router configurations by mail

```
import smtplib
from email import message
import mimetypes
from netmiko import Netmiko
```

```
def collect_configuration():
  host = ["10.10.10.1", "10.10.10.2", "10.10.10.3"]
  for ip in host:
   device = { "host": ip, "username": "admin", "password":
   "cisco", "device_type": "cisco_ios"}
   net_connect = Netmiko(**device)
   output = net_connect.send_command("show run")
   with open (f"{ip} config.txt","w") as wr:
    wr.write(output)
   net_connect.disconnect()
  return host
host = collect_configuration()
mail_from = "example@gmail.com"
mail_password = "16-DIGIT-PASSWORD"
mail_to = "example@gmail.com"
mail_subject = "Router Configurations"
mail_content = "Hi,\nYou can find the all configuration files in
the attachment."
send = message.EmailMessage()
send.add_header("From", mail_from)
send.add_header("To", mail_to)
send.add_header("Subject", mail_subject)
send.set_content(mail_content)
for file in host:
  filename = f"{file} config.txt"
  with open(filename, "rb") as r:
   attached_file = r.read()
  mime_type, encoding = mimetypes.guess_type(filename)
  send.add_attachment(attached_file, maintype=mime_type.split("/")
  [0], subtype=mime_type.split("/")[1], filename= filename)
with smtplib.SMTP_SSL("smtp.gmail.com", 465) as smtp:
  smtp.login(mail_from, mail_password)
  smtp.sendmail(mail_from, mail_to, send.as_string())
```

Reachability test to network devices

The most basic and initial troubleshooting step in networking is to make the reachability tests. These are the **ping** and **traceroute** tests. The ping test checks whether the remote device is reachable from the source device, and we test it for connectivity problems. The traceroute test checks the hops or the devices in the network until we reach the destination device. For example, if we have a topology A-B-C-D, there are four different routers in the network. When we make a traceroute test from A to D, if A and D can ping each other, the traceroute output will be A, B, C, and D. So, all the hops will be shown in the output.

Ping test script

In *Example 7.7*, we create a script to make reachability tests for various IP addresses from our local PC. In this example, we use a Windows machine, and the code is also for Windows OS. The following output shows two ping tests: `10.10.10.1` which is a reachable IP address, and `10.10.10.10`, which is an unreachable IP address. We make the ping test from the local PC cmd (command prompt).

If the ping test is successful, the reply message has the destination IP address, packet size in bytes, period of packet travels from source to destination and **Time to Live** (**TTL**), which is the packet life cycle. At the end, there are values like `Sent`, `Received`, and `Lost`, with the lost rate inside the parentheses.

If the ping test fails, it gives a message saying `Request timed out`.

```
C:\> ping 10.10.10.1
Pinging 10.10.10.1 with 32 bytes of data:
Reply from 10.10.10.1: bytes=32 time=4ms TTL=255
Reply from 10.10.10.1: bytes=32 time=6ms TTL=255
Reply from 10.10.10.1: bytes=32 time=6ms TTL=255
Reply from 10.10.10.1: bytes=32 time=6ms TTL=255
Ping statistics for 10.10.10.1:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss)
C:\> ping 10.10.10.10
Pinging 10.10.10.10 with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.
Ping statistics for 10.10.10.10:
Packets: Sent = 4, Received = 0, Lost = 4 (100% loss)
```

We create a variable in that each item is the destination IP address or website can ping. We try to send three ping packets. To start the ping, we execute the

Popen function from the **subprocess** module in Windows. We write the ping count parameter and the IP information inside this function. We collect the output and get all data we need from a list with an **append** function. Finally, we create the Excel file with this data and save it.

We call the **Popen** function from the subprocess module in the following code. Inside the parentheses, we write the **ping** command with **cmd /c ping** and then write the IP address with the **-n** option, which sets the ping packet count. In this example, it's three. We also add **stdout** and **encoding** parameters. We create a loop to get the ping process line-by-line and use the **rstrip** string method to remove whitespaces at the end of the string.

```
output = Popen(f"cmd /c ping {ip} -n {ping_count}", stdout=PIPE,
encoding ="utf-8")
  for line in output.stdout:
    data = data +"\n" + line.rstrip('\n')
```

Example 7.7: Ping test from command prompt in Windows

```
from re import findall
from pandas import DataFrame
from subprocess import Popen, PIPE
host = ["10.10.10.1","123.214.2.3","www.google.com",
"192.168.123.24", "8.8.8.8"]
ping_count = "3"
packet_loss, ip_list, status_list, sent_list, received_list,
lost_list = ([] for i in range(6))
for ip in host:
  data = ""
  print(f"\n---Try to Ping: {ip} ---")
  output= Popen(f"cmd /c ping {ip} -n {ping_count}",stdout=PIPE,
  encoding="utf-8")
  for line in output.stdout:
    data = data +"\n" + line.rstrip('\n')
  print(data)
  ping_test = findall("TTL", data) #Check TTL word if the ping is
  successful or not
  if ping_test:
    status = "Successful"  #Ping Successful or Failed
    sent = findall("Sent = (\d+)", data)  #Find Sent packet number
    received = findall("Received = (\d+)", data)  #Find received
    packet number
```

```
    lost = findall("Lost = (\d+)", data)  #Find lost packets
    number
    loss = findall("\((.*) loss", data)  #Get loss packet
    percentage
  else:
    status = "Failed"
    sent = findall("Sent = (\d+)", data)
    received = ["0"]
    lost = sent
    loss = ["100%"]
  sent_list.append(sent[0])
  received_list.append(received[0])
  lost_list.append(lost[0])
  packet_loss.append(loss[0])
  ip_list.append(ip)
  status_list.append(status)
  df = DataFrame({"IP Address": ip_list, "Status": status_list,
  "Sent": sent_list, "Received": received_list, "Lost": lost_list,
  "Packet Loss Rate": packet_loss})
  df.to_excel("Ping Result.xlsx", sheet_name="Ping", index=False)
```

Figure 7.7 shows six different items and five IP addresses as tested. We can see that each item and values is in a separate column:

IP Address	Status	Sent	Received	Lost	Packet Loss Rate
10.10.10.1	Successful	3	3	0	0%
123.214.2.3	Failed	3	0	3	100%
www.google.com	Successful	3	3	0	0%
192.168.123.24	Failed	3	0	3	100%
8.8.8.8	Successful	3	3	0	0%

***Figure** 7.7: Output of the "Ping Result" Excel File*

Traceroute test script

In *Example 7.8*, we make a traceroute test to the destination IP address. So, we try to get each hop or router IP address until the destination IP address. In Windows, the maximum traceroute hop is `30`. We can change this with the `-h` parameter in the `tracert` command.

As we did in *Example 7.7*, we use the `Popen` function. We only change the `ping` command with tracert and add the `-h` parameter with its value as `hops`.

After that, we collect the logs in the **data** variable. We save the output of the traceroute command in a file in the same directory as our script.

We also check whether we reach the destination IP in the specified max hops value. If the final line of the loop has the destination IP address, we reach the target device; otherwise, we cannot reach it. So, we search for the IP address in the output. The IP address is also at the beginning of the traceroute, like in the following output. So, to identify it, we use the **findall** function by writing **ms\s+IP_ADDRESS**. We can locate the IP address on the last line if it exists.

```
C:\> tracert -h 1 10.10.10.1
Tracing route to 10.10.10.1 over a maximum of 1 hops
  1     7 ms     6 ms    10 ms  10.10.10.1
```

Example 7.8: Tracert test from command prompt in Windows

```
from subprocess import Popen, PIPE
from re import findall
hostname = "10.10.10.1"
hops = 1
output = Popen(f"cmd /c tracert -h {hops} {hostname}",stdout=PIPE,
encoding="utf-8")
data = ""
for line in output.stdout:
  data = data + "\n" + line.rstrip('\n')
  print(line.rstrip('\n'))
with open (f"Traceroute to {hostname}","w") as wr:
  wr.write(data)
result = findall(f"ms\s+{hostname}",data)
if result:
  print (f"***Traceroute to {hostname} is successfully finished")
else:
  print(f"***Cannot reach {hostname}")
```

Conclusion

In this chapter, we learned the operational steps for software upgrades in network devices, such as uploading files, setting the boot software file, and reloading the device. We also modified dummy data to make it meaningful for engineers, like collecting all alarms from the network and creating statistics to check the risks by severity. We also collected the device data with SNMP, such as system information, hostname, or interface status. We created backups of the configuration files and sent them with emails in attachments. Towards the end, we made a reachability test by executing the **`ping`** and **`tracert`** commands in the Windows machines to troubleshoot the network.

The next chapter will focus on monitoring and managing Linux servers and storage. We will create scripts to maintain multiple servers concurrently, such as collecting logs, installing new packages, and upgrading operating systems.

Multiple choice questions

1. Which command is used to restart Cisco devices?
   a. `reboot`
   b. `restart`
   c. `reload`
   d. `shutdown`
2. How can you change the maximum hop count in the tracert in Windows OS?
   a. `-n`
   b. `-t`
   c. `-m`
   d. `-h`
3. What data is collected from the devices when we write the **`sysUpTime`** parameter in SNMP?
   a. System uptime in minutes
   b. System uptime in hundreds of seconds
   c. System uptime in seconds
   d. System uptime in hours

Answers

1. c
2. d
3. 3
4. b

Questions

1. Write a script to collect the interface information, such as interface name, interface number, interface status, interface IP address, and description.

CHAPTER 8
Monitor and Manage Servers

This chapter will focus on server management, including collecting logs and configuring servers. We will use paramiko and netmiko modules to log in to servers. All examples in this chapter are based on Ubuntu OS, which is a Linux distro. We will implement the server environment, collect logs, and modify them and change configurations on the Linux servers, which are daily tasks for a system engineer.

Structure

In this chapter, we will cover the following topics:

- Implement server environment
  - Download VMware player and Ubuntu
  - Install Ubuntu on VMware
  - Activate SSH connection
- Maintain Linux servers
  - Collect logs via syslog
  - Login servers with secure password
  - Collect CPU and memory levels
  - Collect interface information
  - Collect type and permission of files
- Server configurations
  - Create users in servers
  - Install packages
  - Transfer files with Paramiko
  - Reboot servers concurrently
  - Stop running processes by script

Objectives

With the help of the **Virtual Machine** (**VM**) tool, we will prepare a lab setup with three Linux servers. We will set up Ubuntu as the OS and VMware Player as the virtual machine tool. We will also log in to servers using the **`netmiko`** and **`paramiko`** modules. We will be gathering the syslog information and transmitting it to you as an attachment. Additionally, we will discuss how to configure servers by adding a new user, moving files, rebooting the servers, and quitting any open processes.

Implement server environment

In system engineering, servers are the essential devices to work with, like routers and switches in network engineering. We use Linux servers to execute automation scripts and still use the Pycharm tool in Windows, but these scripts can also be run on Linux devices.

We will install three **Ubuntu** OS as Linux Distributors in this chapter. In the previous chapters, we always worked with network devices, mainly Cisco devices. In this chapter, we will connect and automate system devices, and Linux OS are the essential systems for automation as a system engineer. You can also use Fedora, Suse, or other Linux distros instead of Ubuntu. In this chapter, we will write our scripts for Ubuntu OS.

We need to create the environment for the scripts to execute. The lab has three Ubuntu servers in VMs, and the following steps belong to the Windows OS. For Linux or MAC, you need to check from the internet. There are tiny differences between them in creating the Ubuntu server environment.

Download VMware player and Ubuntu

To use Ubuntu on a Windows PC, we have the option to use VM tools. So, we download the VM tool as VMware player and Ubuntu's latest version from their respective official websites:

1. We need to download **VMware Workstation Player** from VMware's official source. At the end of the page are Windows and Linux versions of VMware Player. We are using Windows, so we need to install the Windows version.

   https://www.vmware.com/tr/products/workstation-player/workstation-player-evaluation.html

2. After the download process, we must install the VMware Player tool on our PC. We can also use other VM tools like VirtualBox or Hyper-V as virtualization tools. In all examples, we use VMware Player.
3. After that, we need to download the latest version of the **Ubuntu OS** from the official Ubuntu website. It's recommended to download the **Long Time Support** (**LTS**) version of Ubuntu as it has official support for upgrades and any bugs and vulnerabilities. That said, you also have the option to download older versions.

   https://ubuntu.com/download/desktop

Install Ubuntu on VMware

We need to import the Linux distro as Ubuntu into the VMware tool, so we install the OS by following the steps mentioned here. We install three Linux Servers in the VM, so we can automate three of them with a single Python script.

1. After downloading the Ubuntu file, which has a `.iso` file extension, we must import it to the VMware tool. We open VMware and click on the `Create a New Virtual Machine` button to import the ISO file, as shown in *Figure 8.1*:

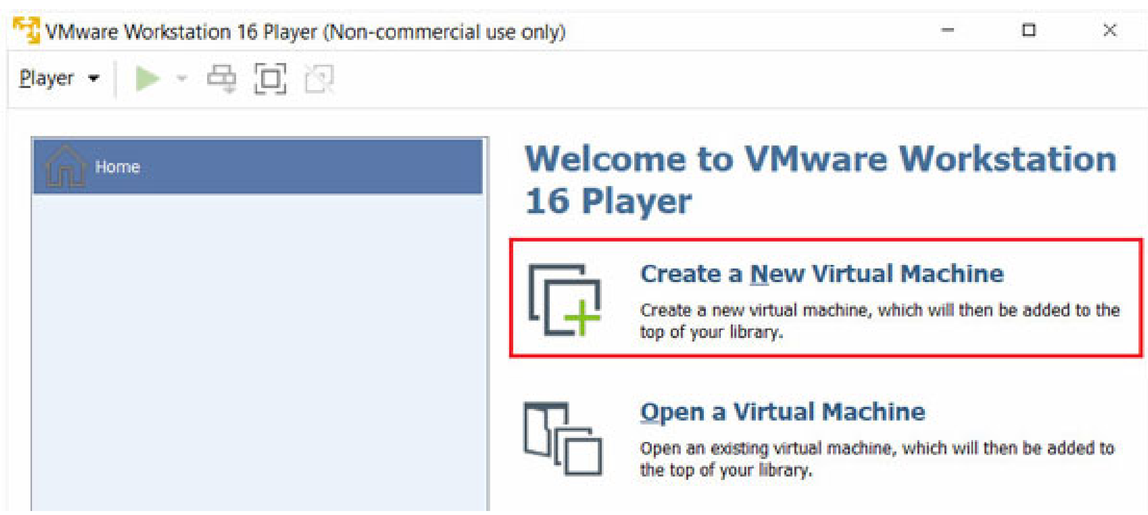


Figure 8.1: *Importing the ISO file step-1*

2. On the opening page, we choose `Installer disc image file (iso)`: and find the downloaded ISO file by clicking on the `Browse` button, as shown in *Figure 8.2*:

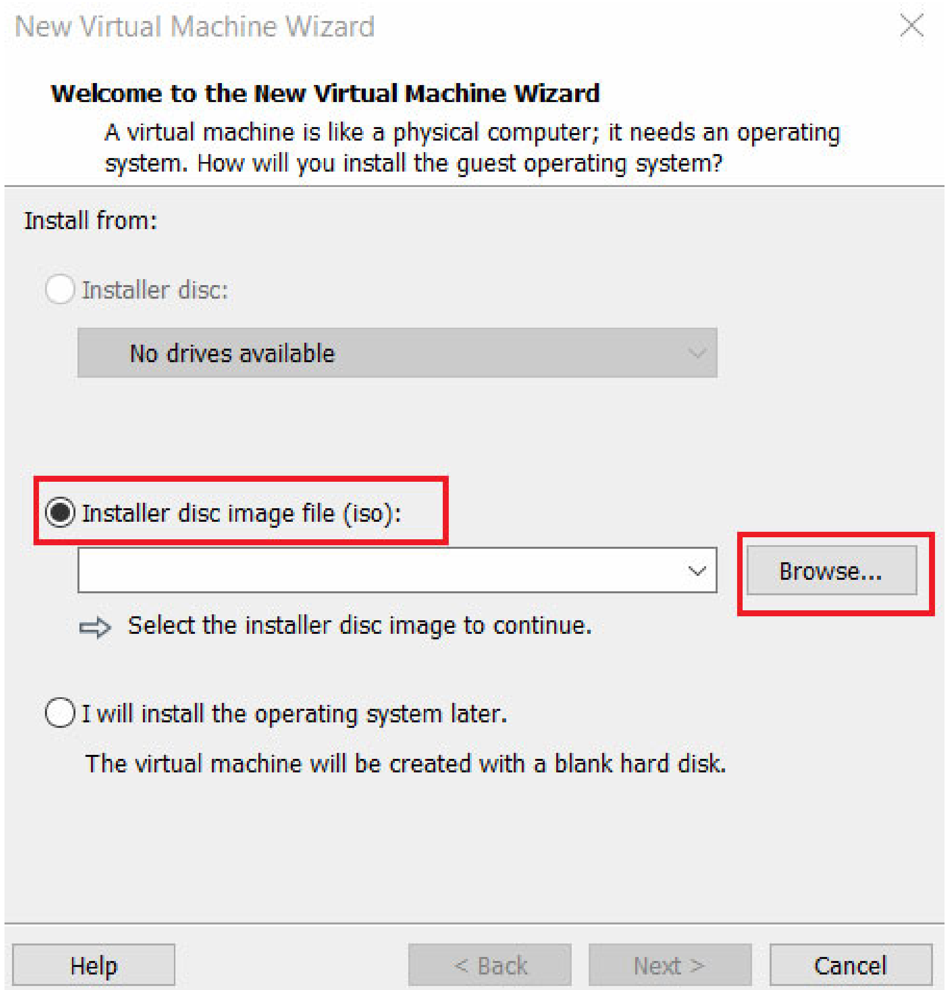


Figure 8.2: *Importing the ISO file step 2*

3. The next page asks for **`Full name`** as hostname, username, and password. We need to fill in these values. You can add the following data in this part:

   `Full name: Server-1`

   `Username: ubuntu`

   `Password: ubuntu`

4. After that, it asks to set the **`Maximum Disk Size`**. By default, it's 20 GB, which is enough for simple usage.
5. We finalize the steps by finishing other steps by the default values.
6. When it finishes successfully, we can see the New Ubuntu VM in the VMware tool with its hostname or full name that we configured in step 3. When we open it, Ubuntu installation from the ISO file has started.

7. The installation of Ubuntu takes 10-20 minutes, and it asks some questions to continue, such as language options, timezone, username, password, and hostname. After choosing the configurations or going with the default options, it is installed.
8. After the installation of Ubuntu finishes, Ubuntu starts on the same VM page. The first server is ready to automate. Now, we copy this VM to create other VMs, or we can install two other VMs in the same procedure.

```
Hostname: Server-1 / Server-2 / Server-3
Username: ubuntu
Password: Ubuntu
```

Activate SSH connection

After finishing the Server OS installation, we need to configure the SSH connection and activate it. We connect Linux servers with the SSH protocol, so we install the net-tools and openssh-server packages. Then, we can configure the IP addresses in the same subnet and activate the SSH server in **`systemctl`**.

1. We can log in to **`terminal`** via the **`Show Applications`** section in the Ubuntu window in the bottom-left corner.
2. We need to download the net-tools package with the **`sudo apt install net-tools`** command in the terminal. After that, we can run the **`ifconfig`** command. There is an interface that has a **`192.168.163.135`** IP address, and it's automatically given to the server. You can change the IP range, but remember that you may need to change the network driver IP address if you want to reach the internet.

```
ubuntu@Server-1:~$ sudo apt install net-tools
ubuntu@Server-1:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
 inet 192.168.163.135  netmask 255.255.255.0  broadcast
 192.168.163.255
 inet6 fe80::87fe:a97d:5cf7:9625  prefixlen 64  scopeid
 0x20<link>
  ether 00:0c:29:ff:0e:b1  txqueuelen 1000  (Ethernet)
  RX packets 149785  bytes 218151216 (218.1 MB)
  RX errors 0  dropped 0  overruns 0  frame 0
  TX packets 19397  bytes 1316978 (1.3 MB)
```

```
TX errors 0  dropped 0 overruns 0  carrier 0  collisions
0
```

When we write the SSH connection scripts in Python, we use the following IP addresses to log in to servers. In your environment, IP addresses may be different.

```
Server-1 IP Address: 192.168.163.135
Server-2 IP Address: 192.168.163.136
Server-3 IP Address: 192.168.163.137
```

We need to test the server IP with a `ping` command from the local PC. Reachability is successful if we can ping all the server's IP addresses from our local PC.

3. We must enable SSH protocol in servers. Otherwise, we cannot log in to the devices with SSH by default. We must install the `openssh-server` package to activate SSH. After that, we must enable the SSH service in the system and start the SSH service.

```
$ sudo apt-get install openssh-server
$ sudo systemctl enable ssh
$ sudo systemctl start ssh
```

4. If all the SSH activation commands are successful, we can make an SSH connection test from our local PC to the servers. We need to write `ssh USERNAME@IP_ADDRESS` to enter a device with SSH. So, we write the following command in the Windows terminal. The server username and password are `ubuntu`, and we can log in via SSH if we write the server's IP address.

```
C:\>ssh ubuntu@192.168.163.135
ubuntu@192.168.163.135's password:
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic
x86_64)
 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
0 updates can be applied immediately.
Last login: Mon Aug 29 10:25:03 2022 from 192.168.163.1
ubuntu@Server-1:~$
```

Maintain Linux servers

This part will focus on collecting logs from the Linux servers. Maintaining the system devices like servers is essential for system engineers, and we can use Python to automate these servers. We will use Ubuntu as a Linux distro in the following examples.

We collect specific data, modify them, and convert them to more readable formats such as creating text or Excel files. We also send the logs in text files with emails. To study the topics in this section, you should be familiar with primary Ubuntu usage. Some basic commands to check are **`sudo`**, **`hostnamectl`**, **`uname`**, **`reboot`**, **`free -m`**, **`htop`**, **`top`**, **`ifconfig`**, **`ps`**, **`ls`**, **`nano`**, **`vim`**, **`rmdir`**, **`mkdir`**, and **`touch`**, with optional parameters and package installation commands like **`sudo apt-get install PACKAGE_NAME`**.

You can check the details of these commands on the internet. On the other hand, you can check any command's manual by writing **`man COMMAND_NAME`** in the Linux terminal.

```
ubuntu@Server-1:~$ man nano
NANO(1)                General Commands Manual                NANO(1)
NAME
   nano - Nano's ANOther editor, inspired by Pico
SYNOPSIS
   nano [options] [[+line[,column]] file]…
   nano [options] [[+[crCR](/|?)string] file]…
DESCRIPTION
     nano  is  a  small and friendly editor. It copies the look
  and feel of Pico, but is free software, and implements several
  features that
     Pico lacks, such as: opening multiple files, scrolling per
  line, undo/redo, syntax coloring, line numbering, and soft-
  wrapping  overlong
   lines.
……
```

In *Example 8.1*, we use the **`paramiko`** module to connect three servers we created. We collect the hostname information from all of them via the **`hostnamectl hostname`** command and display it as the output.

1. We import the necessary functions from the **`paramiko`** and **`time`** modules. After that, we create a variable named **`host`** with the management IP addresses of the servers.

```
from paramiko import SSHClient, AutoAddPolicy
```

```
from time import sleep
host = ["192.168.163.135", "192.168.163.136",
"192.168.163.137"]
```

2. We create a **for** loop to log in to each device in a sequence. We open the SSH session and add the functions to open an active shell session on the device.

```
for ip in host:
 client = SSHClient()
 client.set_missing_host_key_policy(AutoAddPolicy())
client.connect(hostname=ip, username="ubuntu", password=
"ubuntu")
 commands = client.invoke_shell()
```

3. After that, we execute the necessary commands with the paramiko **send** function and wait to get all the output. Finally, we collect the result in human-readable **utf-8** format and display it in the output.

```
 commands.send("hostnamectl hostname\n")
 sleep(1)
 output = commands.recv(1000000).decode("utf-8")
print(f"\n\n-------------------\nConnected to: {ip}\n------
-------------\n{output}")
```

Example 8.1: Connect Ubuntu servers with paramiko

```
from paramiko import SSHClient, AutoAddPolicy
from time import sleep
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
for ip in host:
  client = SSHClient()
  client.set_missing_host_key_policy(AutoAddPolicy())
  client.connect(hostname=ip, username="ubuntu",
  password="ubuntu")
  commands = client.invoke_shell()
  commands.send("hostnamectl hostname\n")
  sleep(1)
  output = commands.recv(1000000).decode("utf-8")
  print(f"\n\n-------------------\nConnected to: {ip}\n-----------
  --------\n{output}")
```

The following output is quite long for a device because when we make an SSH connection to the device, there is a banner that meets us with information about the device and support page links. But we only try to collect the hostname information, such as `Server-1`, `Server-2`, and `Server-3`. We can delete the other data with some functions, like in the `RE` module. On the other hand, we can use the `netmiko` module, which has clearer output according to the `paramiko` module. We will use netmiko to connect servers and collect and configure them in the following examples:

```
-------------------
Connected to: 192.168.163.135
-------------------
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic x86_64)
* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage
0 updates can be applied immediately.
hostnamectl hostname
ubuntu@Server-1:~$ hostnamectl hostname
Server-1
ubuntu@Server-1:~$
-------------------
Connected to: 192.168.163.136
-------------------
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic x86_64)
* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage
0 updates can be applied immediately.
hostnamectl hostname
ubuntu@Server-2:~$ hostnamectl hostname
Server-2
ubuntu@Server-2:~$
-------------------
Connected to: 192.168.163.137
-------------------
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic x86_64)
* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage
0 updates can be applied immediately.
hostnamectl hostname
```

```
ubuntu@Server-3:~$ hostnamectl hostname
Server-3
ubuntu@Server-3:~$
```

Another alternative to using the paramiko module to connect servers is to use the netmiko module. As it's already well-explained in the previous chapters, netmiko has shorter and simple code. Also, the output is much better than that of paramiko.

In *Example 8.2*, we import the Netmiko module and add the device data that netmiko needs to log in. We set the IP address, username, and password as usual, but this time, the value of the **`device_type`** key must be **`linux`** instead of **`cisco`**. For Linux devices, we must always use **`linux`** for this key.

After that, we execute the **`uname -a`** command in the Linux machines, which displays the output of device information, such as hostname, and OS and version release information. When we wrote the **`send_command`**, we added the **`command`** variable in the previous examples. In this example, we also add an optional parameter, **`strip_command`**, with value **`False`**. By default, its value is **`True`**.

At the end, we print the output variable to see the result of the command. The output is much more straightforward, and there is no banner of device information at the beginning of the output. As netmiko removes it for us, we only see the **`command`** output of the device. As we set **`strip_command`** as **`False`**, in the output, the code displays the **`uname -a`** command. If we don't add it, it will not show the command we execute on the device. We can also use this parameter as **`False`** in the previous examples in network devices.

Example 8.2: Connect Ubuntu servers with Netmiko

```
from netmiko import Netmiko
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux"}
  command = "uname -a"
  net_connect = Netmiko(**device)
  output = net_connect.send_command(command, strip_command=False)
  net_connect.disconnect()
  print(f"{ip}:{output}\n")
```

Output:

```
192.168.163.135:uname -a
```

```
Linux Server-1 5.15.0-47-generic #51-Ubuntu SMP Thu Aug 11
07:51:15 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
192.168.163.136:uname -a
Linux Server-2 5.15.0-47-generic #51-Ubuntu SMP Thu Aug 11
07:51:15 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
192.168.163.137:uname -a
Linux Server-3 5.15.0-47-generic #51-Ubuntu SMP Thu Aug 11
07:51:15 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
```

Collect logs via syslog

In *Example 8.3*, we use the netmiko module to log in to servers and collect the syslogs. Syslog is an essential file for Linux servers, and all the log data related to the server is stored inside it. By default, it's inside the **/var/log/** directory. To open a text file in the Linux terminal, we write the **cat** command, which displays the file's output. We only write **cat FILENAME** to show it. So, we execute the **cat** command with the full path of the Syslog file in this example and save the output in different files with the IP address information of three devices.

By default, there is no paging in the Linux terminal. In network devices, there is paging. For example, **terminal length 0** needs to be entered in Cisco. In paramiko examples, we enter it, but in netmiko, it automatically enters this command inside the functions.

Example 8.3: Collect syslog data and save it to the file

```
from netmiko import Netmiko
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux"}
  command = "cat /var/log/syslog"
  net_connect = Netmiko(**device)
  output = net_connect.send_command(command)
  net_connect.disconnect()
  with open (f"{ip} syslog.txt","a") as w:
   w.write(output)
```

In *Example 8.4*, we collect the lines that include **SSH** or **ssh** words in the Syslog. We use the pipeline and the grep word **| grep** to search for something in the file. It's similar to Cisco, such as **| include** . After that, we write

SSH\|ssh, which means find **SSH** or **ssh** words in all lines. **\|** is used as the **or** logical operator in Linux systems.

After we collect the SSH data, we save it to an individual text file with its IP address. We write the emailing script from a Gmail account that we wrote in the previous chapter.

Example 8.4: Collect Syslog data and send by email

```
from netmiko import Netmiko
import smtplib
from email import message
import mimetypes
def collect_configuration():
  host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
  for ip in host:
   device = {"host": ip, "username": "ubuntu", "password":
   "ubuntu", "device_type": "linux"}
   command = "cat /var/log/syslog | grep 'SSH\|ssh'"
   net_connect = Netmiko(**device)
   output = net_connect.send_command(command)
   net_connect.disconnect()
   with open (f"{ip} syslog.txt","a") as w:
    w.write(output)
  return host
host = collect_configuration()
mail_from = "example@gmail.com"
mail_password = "16-DIGIT-CODE"
mail_to = "example@gmail.com"
mail_subject = "Router Configurations"
mail_content = "Hi,\nYou can find the all configuration files in
the attachment."
send = message.EmailMessage()
send.add_header("From", mail_from)
send.add_header("To", mail_to)
send.add_header("Subject", mail_subject)
send.set_content(mail_content)
for file in host:
  filename = f"{file} syslog.txt"
  with open(filename, "rb") as r:
   attached_file = r.read()
```

```
  mime_type, encoding = mimetypes.guess_type(filename)
  send.add_attachment(attached_file, maintype=mime_type.split("/")
  [0], subtype=mime_type.split("/")[1], filename=filename)
with smtplib.SMTP_SSL("smtp.gmail.com", 465) as smtp:
  smtp.login(mail_from, mail_password)
  smtp.sendmail(mail_from, mail_to, send.as_string())
```

Login servers with secure password

We constantly add passwords in scripts, but generally, these users and passwords are unique to engineers. When we share the scripts with other team members, we need to remove the password value from the script. On the other hand, we have the option to enter the password when we execute the script. Python language has a pretty good built-in module as the **`getpass`**. We use the **`getpass`** function from this module, and it creates an input session when we execute the script. If we run the code in the terminal writing **`Python EXAMPLE.py`**, the code asks for the password.

However, if we use an IDE tool like Pycharm, the code gets stuck and does not ask for the password by default. We need to change the setting in the Pycharm tool.

In *Figure 8.3*, we enter the **`Run`** tab and click on the **`Edit Configurations`** section.

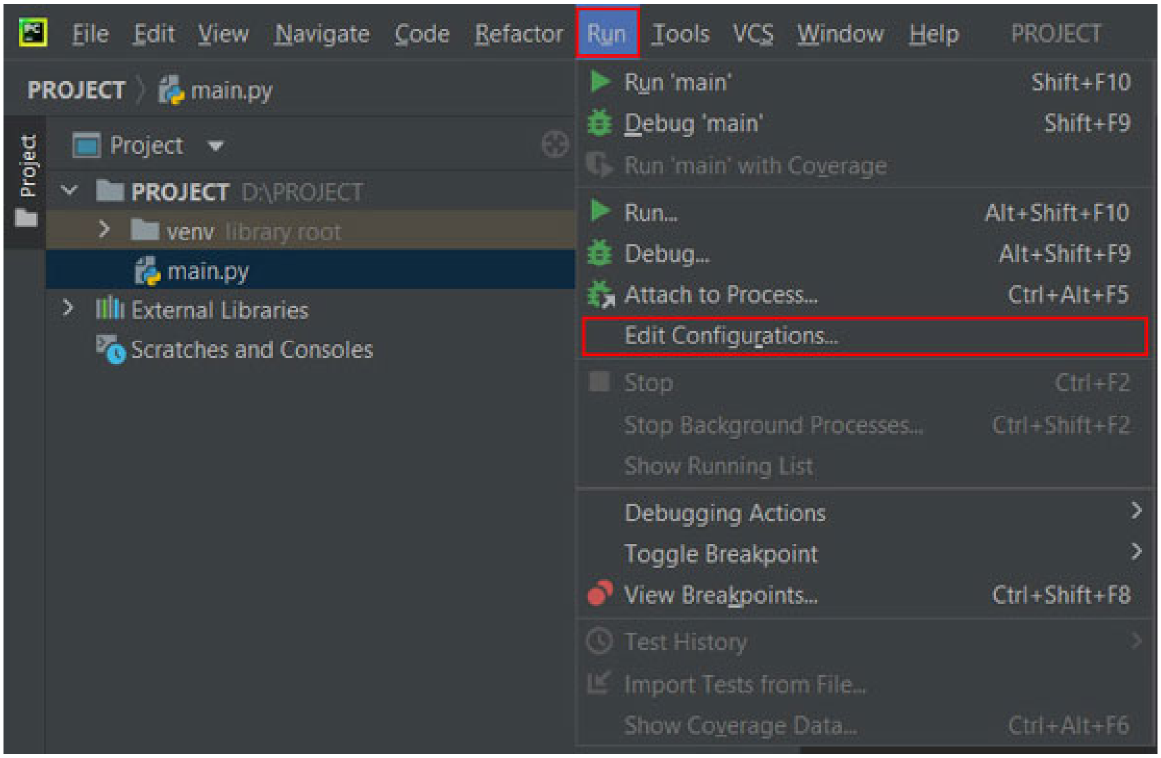


Figure 8.3: *Modifying Pycharm Configuration-1*

In the new opening window, we need to enable the **`Emulate terminal in output console`** feature, like in *Figure 8.4*, and close the window by clicking on the **`Apply`** button.

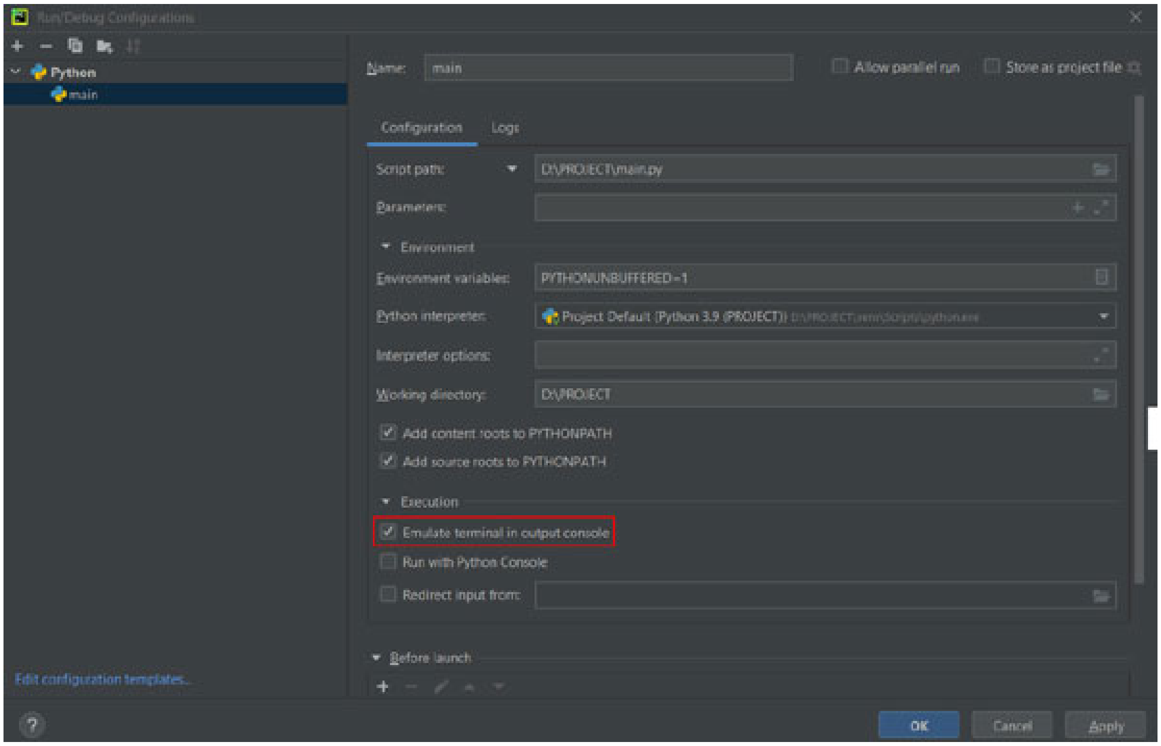


Figure 8.4: Modifying Pycharm Configuration-2

In *Example 8.5*, we import the **getpass** function from the **getpass** module. In the device variable, we write the **getpass()** value for the password key instead of the device password. The only difference in the following code is this.

Example 8.5: Login servers with the secure password with the getpass module

```
from netmiko import Netmiko
from getpass import getpass
host = "192.168.163.135"
device = {"host": host, "username": "ubuntu", "password":
getpass(), "device_type": "linux"}
command = "uname -a"
net_connect = Netmiko(**device)
show_output = net_connect.send_command(command)
net_connect.disconnect()
print(f"{host}:{show_output}\n")
```

When we execute the code, we can see the **Password:** output to enter. We manually enter the password in the output terminal. It's also a secret password, so we cannot see the output of the password value we enter from the keyboard. It's a secure way to enter a password.

Output:

```
Password:
192.168.163.135:
Linux Server-1 5.15.0-47-generic #51-Ubuntu SMP Thu Aug 11
07:51:15 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
```

When we try to log in to multiple devices, if we create a loop in the previous example and execute it, each device code asks for the password. So, if we have 100 devices to log in to, we must write the password 100 times.

In *Example 8.6*, we assign the **`getpass()`** function at the beginning of the code, which is outside of the **`for`** loop. Then, we call this variable as the **`password`** key. In the loop, the value of the password is always the password we enter when executing the code.

Example 8.6: Log in to multiple devices with a secure password

```
from netmiko import Netmiko
from getpass import getpass
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
command = "uname -a"
password = getpass()
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  password, "device_type": "linux"}
  net_connect = Netmiko(**device)
  show_output = net_connect.send_command(command)
  net_connect.disconnect()
  print(f"{ip}:{show_output}\n")
```

There is only one **`Password:`** output. When we enter the password on the terminal, it assigns the new value to the password variable. So, we manually enter the password to collect the following data from three devices.

Output:

```
Password:
192.168.163.135:
Linux Server-1 5.15.0-46-generic #49-Ubuntu SMP Thu Aug 4 18:03:25
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
192.168.163.136:
Linux Server-2 5.15.0-46-generic #49-Ubuntu SMP Thu Aug 4 18:03:25
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
192.168.163.137:
```

```
Linux Server-3 5.15.0-46-generic #49-Ubuntu SMP Thu Aug 4 18:03:25
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
```

Collect CPU and memory levels

In *Example 8.7*, we collect CPU and memory (total, used, and free) data and save it to an Excel file.The **free -m** command in Ubuntu is used to get memory data.

```
ubuntu@Server-1:~$ free -m
  total        used         free     shared    buff/cache    available
Mem:      3889         651          1872      13         1365          2983
Swap:     2139         0            2139
```

Ubuntu's **top** command is used to get memory data, but it's a live log. So, it's frequently updated by default. Suppose we execute this command with netmiko; the code gets stuck in this line because netmiko continues to the following line in the code. If it finishes collecting the log from the device, it's infinite in the **top** command. So, we use the **top -n 1** command. **-n** is used for a number of iterations as limits. If we enter the value of **-n** as **1**, it only gets the data once. The **top** output is quite long, including memory usage, and shows each application's use of resources. So, we execute the **top -n 1 | grep %Cpu** command to get the lines that include the word **%Cpu**.

```
ubuntu@Server-1:~$ top -n 1 | grep %Cpu
%Cpu(s):  5,4 us,  5,4 sy,  0,0 ni, 89,2 id,  0,0 wa,  0,0
hi,  0,0 si,  0,0 st
```

1. We import the **netmiko**, **RE**, and **pandas** modules with their necessary functions.
   ```
   from netmiko import Netmiko
   from re import findall
   from pandas import DataFrame
   ```
2. We create empty lists with the following code. We also write the **host** variable with the device management IP addresses.
   ```
   memory_total, memory_free, memory_used, cpu_used, host_list
   = ([] for i in range(5))
   host = ["192.168.163.135", "192.168.163.136",
   "192.168.163.137"]
   ```
3. We create a **for** loop to log in to devices. Inside the loop, we create a device dictionary with the keys and values that the **netmiko** module

needs to log in to the devices. We log in and execute two commands and assign them to variables like **mem_output** and **cpu_output**.

```
for ip in host:
device = {"host": ip, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux"}
 net_connect = Netmiko(**device)
mem_output = net_connect.send_command("free -m",
strip_command=False)
cpu_output = net_connect.send_command("top -n 1 | grep
%Cpu", strip_command=False)
```

4. We can collect the hostname information with the **find_prompt()** function in the **netmiko** module. So, we collect it and get only the hostname data with the **findall** function. As there are some words like the **USERNAME@HOSTNAME**: value in the prompt, we only try to get the **HOSTNAME** value; that's why we use the findall function.

```
hostname = findall("@(.*):", net_connect.find_prompt())
```

5. We collect the data, and we use the **findall** function to get the specific data for each value: CPU value and total, free and used memory. After that, we assign each match to a particular list we created at the beginning of the code.

```
total = findall("Mem:\s+(\d+)", mem_output)
free = findall("Mem:\s+\d+\s+(\d+)", mem_output)
used = findall("Mem:\s+\d+\s+\d+\s+(\d+)", mem_output)
cpu = findall("\d+,\d+", cpu_output)
memory_total.append(f"{total[0]} MB")
memory_free.append(f"{free[0]} MB")
memory_used.append(f"{used[0]} MB")
cpu_used.append(f"% {cpu[0]}")
host_list.append(hostname[0])
```

Instead of using the code in step 5, we can decrease the lines of code by collecting only digits in the **free -m** command and getting the **total**, **free**, and **used** data in the same list.

```
total = findall("\d+", mem_output)
cpu = findall("\d+,\d+", cpu_output)
memory_total.append(f"{total[0]} MB")
memory_free.append(f"{total[1]} MB")
memory_used.append(f"{total[2]} MB")
cpu_used.append(f"% {cpu[0]}")
```

```
host_list.append(hostname[0])
```

6. After the loop finishes, we use the `DataFrame` function to organize all the lists we filled with data and save them to an Excel file with the `to_excel` function.

```
df = DataFrame({"Hostname": host_list, "Total Memory":
memory_total, "Free Memory": memory_free, "Memory Usage":
memory_used, "CPU Usage": cpu_used})
df.to_excel("CPU-Memory Usage.xlsx", index=False)
```

Example 8.7: Collect CPU and memory levels of servers

```
from netmiko import Netmiko
from re import findall
from pandas import DataFrame
memory_total, memory_free, memory_used, cpu_used, host_list = ([]
for i in range(5))
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux"}
  net_connect = Netmiko(**device)
  mem_output = net_connect.send_command("free -m",
  strip_command=False)
  cpu_output = net_connect.send_command("top -n 1 | grep %Cpu",
  strip_command=False)
  hostname = findall("@(.*):", net_connect.find_prompt())
  total = findall("Mem:\s+(\d+)", mem_output)
  free = findall("Mem:\s+\d+\s+(\d+)", mem_output)
  used = findall("Mem:\s+\d+\s+\d+\s+(\d+)", mem_output)
  cpu = findall("\d+,\d+", cpu_output)
  memory_total.append(f"{total[0]} MB")
  memory_free.append(f"{free[0]} MB")
  memory_used.append(f"{used[0]} MB")
  cpu_used.append(f"% {cpu[0]}")
  host_list.append(hostname[0])
df = DataFrame({"Hostname": host_list, "Total Memory":
memory_total, "Free Memory": memory_free, "Memory Usage":
memory_used, "CPU Usage": cpu_used})
df.to_excel("CPU-Memory Usage.xlsx", index=False)
```

Figure 8.5 shows the Excel file content of the script in *Example 8.7*. There are three servers which has total, free, and used memory count with CPU usage data.

Hostname	Total Memory	Free Memory	Memory Usage	CPU Usage
Server-1	3889 MB	649 MB	1875 MB	% 10,5
Server-2	3889 MB	659 MB	2147 MB	% 5,9
Server-3	3889 MB	640 MB	2160 MB	% 2,9

Figure 8.5: *Excel File Output of Example 8.7*

Collect interface information

In Ubuntu, we use the **`ifconfig`** command to get all interface data from the server. The **`ifconfig`** command software package must be installed to execute this command, and we need to install its package by running **`sudo apt install net-tools`**. When we run **`ifconfig`** in the Ubuntu terminal, we get the following output. It has two interfaces named **`ens33`** and **`lo`** as loopback in Server-1. In the following line, we have **`inet`** and the IP address, **`netmask`** and the netmask address. We will collect this data for the following example.

```
ubuntu@Server-1:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 192.168.163.135  netmask 255.255.255.0  broadcast
    192.168.163.255
    inet6 fe80::87fe:a97d:5cf7:9625  prefixlen 64  scopeid
    0x20<link>
    ether 00:0c:29:ff:0e:b1  txqueuelen 1000  (Ethernet)
    RX packets 29835  bytes 39126512 (39.1 MB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 14325  bytes 2328952 (2.3 MB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
   inet 127.0.0.1  netmask 255.0.0.0
    inet6 ::1  prefixlen 128  scopeid 0x10<host>
    loop  txqueuelen 1000  (Local Loopback)
    RX packets 390  bytes 40395 (40.3 KB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 390  bytes 40395 (40.3 KB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

In *Example 8.8*, we collect the hostname, interface name, interface IP address, and netmask from all devices and write them to an excel file.

1. We import the necessary modules and create empty lists to fill them with data. And we log in to three devices, as we did in the earlier examples.

```
from netmiko import Netmiko
from re import findall
from pandas import DataFrame
list_ipv4, list_netmask, list_int, list_hostname,
list_int_name = ([] for i in range(5))
host = ["192.168.163.135", "192.168.163.136",
"192.168.163.137"]
```

2. We collect the **ifconfig** command output inside the **for** loop from each device. We get specific data with the **findall** function, such as the hostname and interface name, inside the first or outer loop.

```
for ip in host:
device = {"host": ip, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux"}
 net_connect = Netmiko(**device)
 output = net_connect.send_command("ifconfig")
 hostname = findall("@(.*):", net_connect.find_prompt())
 int_name = findall("(.*): flags", output)
```

3. Inside the second or inner loop, we execute the **ifconfig -a** command with the interface name we collected in the first loop. The **ifconfig -a INTERFACE_NAME** CLI command is used to get the only output of a specified interface. So, we get only one interface in each iteration and collect the interface IP address and netmask.

```
 for interface in int_name:
 output = net_connect.send_command(f"ifconfig -a
 {interface}")
  ipv4 = findall("inet (.*)  netmask", output)
  netmask = findall("netmask (\d+.\d+.\d+.\d+)", output)
```

4. After we collect all four sets of data from the logs, we append them to the lists. We are still inside the second loop.

```
  list_ipv4.append(ipv4[0])
  list_netmask.append(netmask[0])
  list_hostname.append(hostname[0])
  list_int_name.append(interface)
```

5. Finally, we exit both loops and save the items of the lists to an Excel file.

```
df = DataFrame({"Hostname": list_hostname, "Interface
Name": list_int_name, "IP Address": list_ipv4, "Netmask":
list_netmask, })
df.to_excel("Interface Information.xlsx", index=False)
```

Example 8.8: Collect interface information of servers

```
from netmiko import Netmiko
from re import findall
from pandas import DataFrame
list_ipv4, list_netmask, list_int, list_hostname, list_int_name =
([] for i in range(5))
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux"}
  net_connect = Netmiko(**device)
  output = net_connect.send_command("ifconfig")
  hostname = findall("@(.*):", net_connect.find_prompt())
  int_name = findall("(.*): flags", output)
  for interface in int_name:
    output = net_connect.send_command(f"ifconfig -a {interface}")
    ipv4 = findall("inet (.*)  netmask", output)
    netmask = findall("netmask (\d+.\d+.\d+.\d+)", output)
    list_ipv4.append(ipv4[0])
    list_netmask.append(netmask[0])
    list_hostname.append(hostname[0])
    list_int_name.append(interface)
df = DataFrame({"Hostname": list_hostname, "Interface Name":
list_int_name, "IP Address": list_ipv4, "Netmask": list_netmask,
})
df.to_excel("Interface Information.xlsx", index=False)
```

In *Figure 8.6*, we can see the output of the script in *Example 8.8*. The hostname, interface name, interface IP address, and the netmask of this IP address are filled in the Excel file in order.

Hostname	Interface Name	IP Address	Netmask
Server-1	ens33	192.168.163.135	255.255.255.0
Server-1	lo	127.0.0.1	255.0.0.0
Server-2	ens33	192.168.163.136	255.255.255.0
Server-2	lo	127.0.0.1	255.0.0.0
Server-3	ens33	192.168.163.137	255.255.255.0
Server-3	lo	127.0.0.1	255.0.0.0

Figure 8.6: *Excel File Output of Example 8.8*

Collect type and permission of files

We can check the file list in Linux machines with the **ls** command in the terminal. The following output has five items with three folders and two files.

```
ubuntu@Server-1:~$ ls
Desktop  Downloads  nohup.out  Pictures   test.txt
```

We can run the **ls -l** command to get detailed information about the items, such as the item type, permissions, user information of creation, file size, and creation time. We have the following output after running the **ls -l** command in Server-1.

```
ubuntu@Server-1:~$ ls -l
total 28
drwxr-xr-x 2 ubuntu server-1 4096 Aug 28 20:38 Desktop
drwxr-xr-x 2 ubuntu server-1 4096 Aug 28 20:38 Downloads
-rw------- 1 ubuntu server-1    0 Sep  1 08:30 nohup.out
dr--r--r-- 2 ubuntu server-1 4096 Aug 28 20:38 Pictures
---------- 1 ubuntu server-1   20 Sep  1 08:19 test.txt
```

Each file's line starts with the **d** or **-** character. **d** means that the item is a directory, and - means that the item is a file. So, we can easily understand the item type, whether a file or a folder.

After the first character, the following three characters specify the user permission. These characters can be **r** for reading permission, **w** for writing permission, **x** for executing permission, and – for no permission.

```
"r" - Read Permission
"w" - Write Permission
"x" - Execute Permission
"-" - No Permission
```

We can change the permission of the items with the **chmod** command. We write - to remove permissions and **+** to add permissions.

To delete all permissions: chmod -rwx test.txt

To add all permissions: chmod +rwx test.txt

In *Example 8.9*, we collect the items as folder or file with its extension, file type as file or folder, and permissions as read, write, execute, or none.

1. We import the necessary functions from the **netmiko** and **RE** modules. We add a device variable to let netmiko to log in to the server.

   ```
   from netmiko import Netmiko
   from re import findall,split
   device = {"host": "192.168.163.135", "username": "ubuntu",
   "password": "ubuntu", "device_type": "linux"}
   ```

2. After we connect to the device, we run the **ls -l** command and use the **split** function to split each line into an item in a list. So, the **output** variable is a list. After that, we delete two items at the beginning of the list with the **del output[:2]** function because these two lines are unnecessary in the output of the **ls -l** command.

   ```
   net_connect = Netmiko(**device)
   output = split("\n",net_connect.send_command("ls -l"))
   del output[:2]
   ```

3. Inside the **for** loop, we iterate each item in the output list. We collect the file name with the **findall** function. In each item, the first value or character specifies the type as folder or file. So if we use **item[0]**, it gets this value. If we use **item[1:4]**, it gets the user permission value, such as **rwx** or a different value.

   ```
   for item in output:
    file_name = findall("\d+:\d+ (.*)",item)
    print(f"File/Directory Name: {file_name[0]}")
   ```

4. So, we create two **if** conditions: one to find the item type, and one to find the permission type. We had many options in the permission type. We add some permission types with their meanings, such as **rw-**, as the **Read/Write Permission**.

   ```
    if item[0] == "d":
     print("Type: Dictionary")
    else:
     print("Type: File")
    if item[1:4] == "r--":
   ```

```
         print(f"User Permission: Read as '{item[1:4]}'\n")
        elif item[1:4] == "rw-":
         print(f"User Permission: Read/Write as '{item[1:4]}'\n")
        elif item[1:4] == "rwx":
        print(f"User Permission: Read/Write/Execute as
        '{item[1:4]}'\n")
        elif item[1:4] == "---":
         print(f"User Permission: None as '{item[1:4]}'\n")
```

Example 8.9: Collect file type and permissions in a directory of a server

```
from netmiko import Netmiko
from re import findall,split
device = {"host": "192.168.163.135", "username": "ubuntu",
"password": "ubuntu", "device_type": "linux"}
net_connect = Netmiko(**device)
output = split("\n",net_connect.send_command("ls -l"))
del output[:2]
for item in output:
  file_name = findall("\d+:\d+ (.*)",item)
  print(f"File/Directory Name: {file_name[0]}")
  if item[0] == "d":
   print("Type: Directory")
  else:
   print("Type: File")
  if item[1:4] == "r--":
   print(f"User Permission: Read as '{item[1:4]}'\n")
  elif item[1:4] == "rw-":
   print(f"User Permission: Read/Write as '{item[1:4]}'\n")
  elif item[1:4] == "rwx":
   print(f"User Permission: Read/Write/Execute as
   '{item[1:4]}'\n")
  elif item[1:4] == "---":
   print(f"User Permission: None as '{item[1:4]}'\n")
```

Output:

```
File/Directory Name: Desktop
Type: Directory
User Permission: Read/Write/Execute as 'rwx'
File/Directory Name: Downloads
Type: Directory
```

```
User Permission: Read/Write/Execute as 'rwx'
File/Directory Name: nohup.out
Type: File
User Permission: Read/Write as 'rw-'
File/Directory Name: Pictures
Type: Directory
User Permission: Read as 'r--'
File/Directory Name: test.txt
Type: File
User Permission: None as '---'
```

Server configurations

Here, we will focus on configuring servers with Python scripts. We will use netmiko and paramiko modules in the following examples, and we can create users, install packages, transfer files both ways, reboot servers, and kill processes with the scripts.

Create users in servers

In *Example 8.10*, we create a user in the servers and collect their UID, GID, and group information to display in the output. We use the Jinja2 template from a file and data from the YAML file in the following:

```
info.yaml
user_name: test_user
group_name: test_group
command_list.txt
useradd {{user_name}}
addgroup {{group_name}}
usermod -a -G {{group_name}} {{user_name}}
id {{user_name}}
```

1. We import the necessary functions from the **netmiko**, **RE**, **jinja2**, and **yaml** modules. Then, we create a **host** variable for the device management IP addresses.

    ```
    from netmiko import Netmiko
    from re import findall, split
    from jinja2 import Environment, FileSystemLoader
    from yaml import safe_load
    ```

```
host = ["192.168.163.135", "192.168.163.136",
"192.168.163.137"]
```

2. We use the **Environment** function from the **jinja2** module with the **FileSystemLoader** function to load the jinja platform. After that, we call the **get_template** function to get the jinja codes.

```
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("command_list.txt")
```

3. We open the YAML file, read it with the **safe_load** function, and get the values. We get the username data from the file to display in the output.

```
with open("info.yml") as r:
 data = safe_load(r)
 user_name = data["user_name"]
```

4. We render or merge the Jinja commands with the YAML file with the **render** function and create a list of items divided line by line.

```
command = template.render(data)
command = split("\n", command)
```

5. Inside a **for** loop, we create a device variable and add the **secret** key with its value as **ubuntu**. It's the root user's password. We can set the root password by entering the **sudo passwd root** line in the terminal and putting the new password in the following line. After that, we log in to devices and execute the **command** variable. We get the hostname value with the **find_prompt** function, and then we check the **uid** information in the output.

```
for ip in host:
device = {"host": ip, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux", "secret": "ubuntu"}
 net_connect = Netmiko(**device)
 output = net_connect.send_config_set(command)
 hostname = findall("@(.*):", net_connect.find_prompt())
 result = findall("uid",output)
```

6. If there is a **uid** word in the output, it means the user has been created successfully. Otherwise, the code has failed to create a user in the server. If the **result** variable has a value, we collect the **uid**, **gid**, and **groups** values and display them in the output.

```
  if result:
   uid = findall("uid=(.*) gid",output)
   gid = findall("gid=(.*) ",output)
```

```
   groups = findall("groups=(.*)",output)
  print(f"{hostname[0]}: User '{user_name}' is created and
  assigned to a group")
  print(f"UID: {uid[0]} \nGID: {gid[0]} \nGroups:
  {groups[0]}\n")
  else:
   print("Failed to create user and group")
```

Example 8.10: Create users in servers

```
from netmiko import Netmiko
from re import findall, split
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("command_list.txt")
with open("info.yml") as r:
  data = safe_load(r)
  user_name = data["user_name"]
command = template.render(data)
command = split("\n", command)
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux", "secret": "ubuntu"}
  net_connect = Netmiko(**device)
  output = net_connect.send_config_set(command)
  hostname = findall("@(.*):", net_connect.find_prompt())
  result = findall("uid",output)
  if result:
   uid = findall("uid=(.*) gid",output)
   gid = findall("gid=(.*) ",output)
   groups = findall("groups=(.*)",output)
   print(f"{hostname[0]}: User '{user_name}' is created and
   assigned to a group")
   print(f"UID: {uid[0]} \nGID: {gid[0]} \nGroups:
   {groups[0]}\n")
  else:
   print("Failed to create user and group")
```

In the output, three device outputs create a **test_user**, showing UID, GID, and groups.

Output:

```
Server-1: User 'test_user' is created and assigned to a group
UID: 1005(test_user)
GID: 1008(test_user)
Groups: 1008(test_user),1009(test_group)
Server-2: User 'test_user' is created and assigned to a group
UID: 1006(test_user)
GID: 1009(test_user)
Groups: 1009(test_user),1010(test_group)
Server-3: User 'test_user' is created and assigned to a group
UID: 1006(test_user)
GID: 1011(test_user)
Groups: 1011(test_user),1012(test_group)
```

Install packages

In *Example 8.11*, we install a package from the internet to a server. So, the server must have an internet connection to download the package and install it.

1. We import the **Nemiko** function from the **netmiko** module and set a **host** variable as the server management IP address. After that, we create a device variable with its data to log in to the device by netmiko. We also add the **secret** key inside the variable because in Ubuntu, we must enter the admin mode to install a package or run the command by adding **sudo** at the beginning of the line, such as **sudo apt-get install PACKAGE_NAME**. Then, we add the **package** variable by adding the package name as a value.

   ```
   from netmiko import Netmiko
   host = "192.168.163.135"
   device = {"host": host, "username": "ubuntu", "password":
   "ubuntu", "device_type": "linux", "secret": "ubuntu"}
   package = "htop"
   ```

2. We connect to the device and send the command with the **send_config_set** function. This function executes the commands in the network devices' configuration terminal or admin mode. It's the same in Linux, and the command runs in the admin mode. So, we write **apt-get install PACKAGE_NAME**. In the following code, we add **-y** at the end of

the command. By default, if the size is large, such as 10MB or more, the Ubuntu machine asks us whether or not to continue to download the package from the internet. The server waits until we enter **Y** or **N**. It stays infinite if we do not enter **Y** into the question. Netmiko gives a timeout error when the command doesn't finish. To prevent this issue, we can add **-y** at the end of the command, regardless of the package size, so that the code works fine. We can also add the **read_timeout** parameter to extend the timeout if the command doesn't finish in the default timeout period. If we have a large file to download, it takes much more time, depending on the local internet connection. We can set the timeout value with this parameter.

```
net_connect = Netmiko(**device)
output =net_connect.send_config_set(f"apt-get install
{package} -y",read_timeout=1000)
print(output)
```

3. After we download the file, we test it by writing the **PACKAGE_NAME --version** command in the user mode of the server. Each package has version information, and we can see it with that command. If the package installation fails, we can see it with this output.

```
output = net_connect.send_command(f"{package} --version")
print(f"{host}: {package} --version{output}\n")
net_connect.disconnect()
```

Example 8.11: Install a package on a server

```
from netmiko import Netmiko
host = "192.168.163.135"
device = {"host": host, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux", "secret": "ubuntu"}
package = "htop"
net_connect = Netmiko(**device)
output = net_connect.send_config_set(f"apt-get install {package} -
y")
print(output)
output = net_connect.send_command(f"{package} --version")
print(f"{host}: {package} --version{output}\n")
net_connect.disconnect()
```

When the code executes, it automatically enters Ubuntu's admin or root user to run the configuration change command, like in Cisco or other network devices.

In Ubuntu, it's **sudo -s**. We already added the **secret** key and its value as **ubuntu**, which is the root user's password. We set this password after we log in to the server. After that, it executes the command without **sudo** at the beginning of the command. We add **-y** at the end of the line because if we download a large package, it will not ask the user to continue or stop the installation. If we don't enter **-y** and the size is large, the code throws an error because of timeout.

Output:

```
192.168.163.135:sudo -s
[sudo] password for ubuntu:
root@Server-1:/home/ubuntu# apt-get install htop -y
Reading package lists… 0%
Preparing to unpack …/htop_3.0.5-7build2_amd64.deb …
Progress: [0%] [……….] 87Progress: [ 20%] [################…….]
exit
ubuntu@Server-1:~$
192.168.163.135: htop --version
htop 3.0.5
```

In *Example 8.12*, we install multiple packages on various servers. We use the concurrent module to execute the commands simultaneously on three devices. So, we have speedy installation of the packages on many devices. We create a function, as we did in the previous example, to log in to the device and execute the commands. And we have a device IP list as the **host** variable. We combine **package_installation** function and list in the **ThreadPoolExecutor**.

Example 8.12: Install packages in servers simultaneously

```
from netmiko import Netmiko
from concurrent.futures import ThreadPoolExecutor
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
def package_installation (ip):
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux", "secret": "ubuntu"}
  package = ["htop","nano", "vim", "nmap"]
  for pack in package:
   net_connect = Netmiko(**device)
   net_connect.send_config_set(f"sudo apt-get install {pack} -y")
   output = net_connect.send_command(f"{pack} --version")
   hostname = net_connect.find_prompt()
   print(f"{hostname}: {pack} --version{output}\n")
```

```
    net_connect.disconnect()
with ThreadPoolExecutor(max_workers=5) as executor:
  result = executor.map(package_installation, host)
```

Transfer files with Paramiko

We can transfer files from our local PC to remote servers, as we did in the network devices in the previous chapters. We can use the **`paramiko`** module to transfer files both ways. We can create a function to connect devices with the SFTP protocol. After that, we can create two additional functions for uploading and downloading in both sides. We use the **`get`** and **`put`** functions to do this. Finally, we can call either the **`sftp_upload`** or the **`sftp_download`** function according to our request.

Example 8.13: Transfer files with the paramiko module

```
from paramiko import SSHClient, AutoAddPolicy
def sftp_connect():
  ssh = SSHClient()
  ssh.set_missing_host_key_policy (AutoAddPolicy())
  ssh.connect(hostname="192.168.163.137", username="ubuntu",
  password="ubuntu")
  sftp = ssh.open_sftp()
  return sftp
def sftp_upload(local_file,remote_file):
  sftp_connect().put(local_file,remote_file)
  sftp_connect().close()
def sftp_download(remote_file,local_file):
  sftp_d = sftp_connect()
  sftp_d.get(remote_file,local_file)
  sftp_d.close()
sftp_upload("test.txt","test.txt")
```

Before uploading the **`test.txt`** file from our local PC to the server, when we run the **`ls`** command in the default directory, there is no **`test.txt`** file.

```
ubuntu@Server-3:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Temp
lates  test  Videos
```

The default directory of the Ubuntu server is **`/home/USERNAME`**, which is **`/home/ubuntu`**. We can check the current directory in Linux by running the **`pwd`** command in the terminal.

```
ubuntu@Server-3:~$ pwd
/home/ubuntu
```

The `test.txt` file's content is in the following line:

```
Output of the "test.txt" file:
Hello, This is a text file.
```

After we execute the script and it finishes, we can check the directory with the `ls` command again. There is a `test.txt` file now.

```
ubuntu@Server-3:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Temp
lates  test  test.txt  Videos
```

When we check the file's content by running the `cat test.txt` command in the terminal, we can see the content, which is precisely the same on the local PC.

```
ubuntu@Server-3:~$ cat test.txt
Hello, This is a text file.
```

Reboot servers concurrently

In *Example 8.14*, we reboot or reload devices simultaneously. We use the **concurrent** module, as we did in the previous examples. We use the **reboot** command to reboot Ubuntu servers, and we add the **secret** key with its value in the **device** variable. As this command also executes in the administrator mode, we display the device hostname information when the reboot process starts.

Example 8.14: Reboot servers simultaneously

```
from netmiko import Netmiko
from re import findall
from concurrent.futures import ThreadPoolExecutor
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
command = "reboot"
def netmiko_reboot(ip):
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux", "secret": "ubuntu"}
  net_connect = Netmiko(**device)
  hostname = findall("@(.*):", net_connect.find_prompt())
  print(f"---Rebooting to:{hostname}---")
  net_connect.send_config_set(command)
  return
```

```
with ThreadPoolExecutor(max_workers=5) as executor:
  result = executor.map(netmiko_reboot, host)
```

When we execute the script, it gives the following output for all servers. After running the script, all three devices will reboot themselves.

Output:

```
---Rebooting to:Server-1---
---Rebooting to:Server-3---
---Rebooting to:Server-2---
```

Stop running processes by script

Each process has a unique PID, *process id* or *process identification number*. When we start a process or an application, it creates its own PID. We can see a complete list of processes and PIDs by entering the **ps fax** command. It shows all the running processes, with details, in Linux systems.

```
ubuntu@Server-1:~$ ps fax
  PID TTY       STAT   TIME COMMAND
    2 ?         S      0:00 [kthreadd]
    3 ?         I<     0:00  \_ [rcu_gp]
    4 ?         I<     0:00  \_ [rcu_par_gp]
    5 ?         I<     0:00  \_ [netns]
    7 ?         I<     0:00  \_ [kworker/0:0H-events_highpri]
   ……………….
```

In *Example 8.15*, we stop a *kill a process* process in Linux termination. We write **kill PID** to stop the process. So, in the beginning, we manually start an application, a calculator in Linux, by running the **gnome-calculator &** command. If we don't write **&**, we cannot write anything in the terminal until we quit the application. So, we write the **&** character to run the app in the background. After we enter the following command in the terminal, the calculator window opens on the Linux machine desktop, as shown in *Figure 8.7*. In the following output, the PID of this application is **17204**. If we terminate the process and rerun it, it will get another PID.

```
ubuntu@Server-1:~$ gnome-calculator &
[1] 17204
ubuntu@Server-1:~$ ps fax | grep 17204
  17204 pts/0   Sl   0:00   \_ gnome-calculator
```

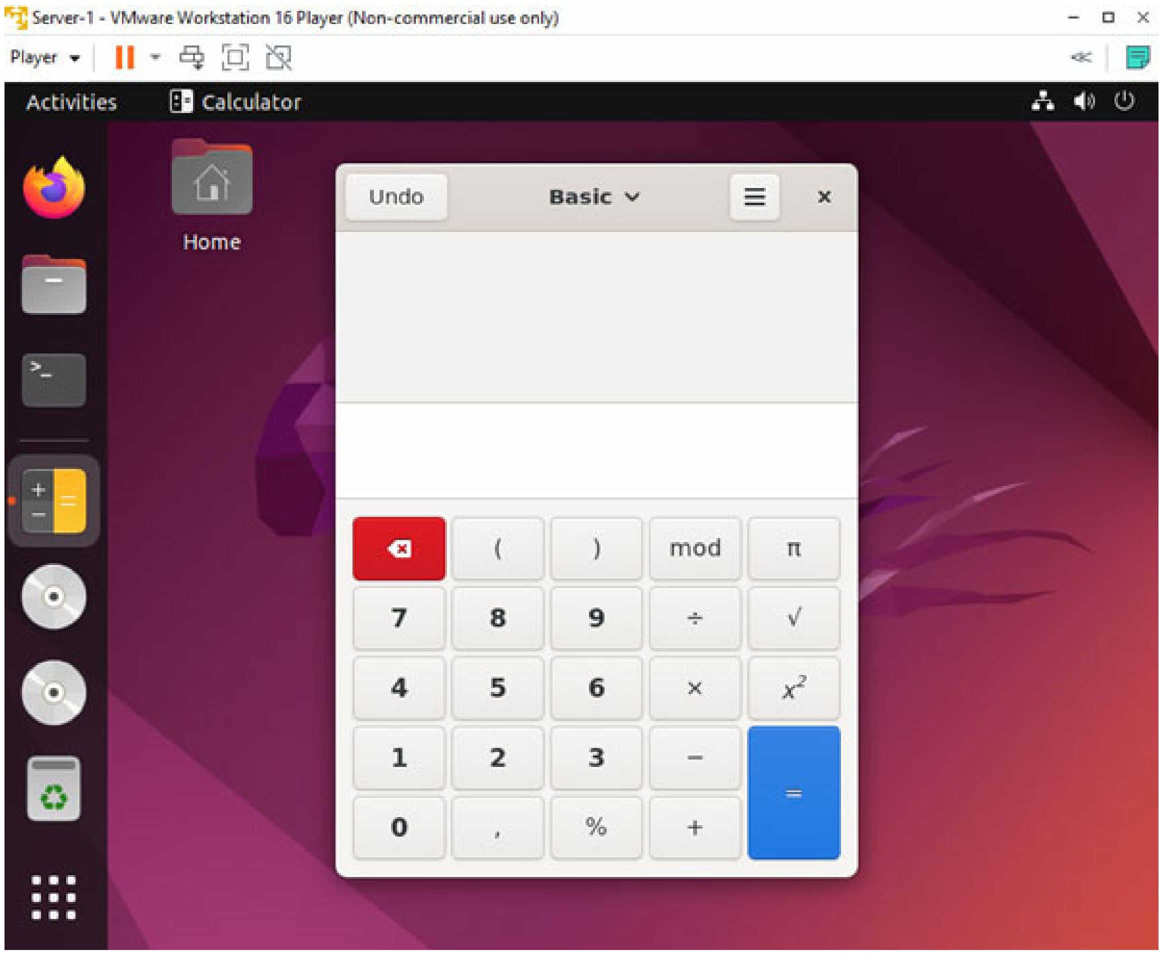


Figure 8.7: Desktop Screen of Server-1 After starting the "gnome-calculator" App

1. We import the necessary functions from the **`netmiko`** and **`RE`** modules. After that, we create a **`host`** variable of the server IP addresses we try to log in to.

```
from netmiko import Netmiko
from re import findall
host = ["192.168.163.135", "192.168.163.136",
"192.168.163.137"]
```

2. We add the **device** variable to connect devices and add the **command** variable to execute the **ps fax** command to find the PID of the process. And finally, the **`process`** variable to find the target process or application to stop the process in all devices.

```
for ip in host:
device = {"host": ip, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux"}
 command = "ps fax"
```

```
process = "gnome-calculator"
```

3. After we log in to the servers, we collect the **ps fax** command output. Then, we get the hostname information with the **find_prompt** function. We also have the PID with the **findall** function, and it's the first digit in the same line as the process name, such as **17204 pts/0 Sl 0:00 _ gnome-calculator**.

```
net_connect = Netmiko(**device)
output = net_connect.send_command(command)
hostname = findall("@(.*):", net_connect.find_prompt())
pid = findall(f"(\d+).*{process}", output)
```

4. If we don't find the PID in the previous code, the **pid** value is empty. So, the code finishes as **Process is not started**. Otherwise, it sends the **kill PROCESS_ID** command to the server to stop the process. When we rerun the **ps fax** command, if we find the PID again, code assigns the output to the **pid_new** variable. This means stopping the process has failed. If the **pid_new** variable is empty, stopping the process is successful. We display all the conditions as output.

```
if pid:
 net_connect.send_command(f"kill {pid[0]}")
 output = net_connect.send_command(command)
 pid_new = findall(f"(\d+).*{process}", output)
 if not pid_new:
  print(f"{hostname[0]}  '{process}' with {pid[0]}
  process-id is successfully stopped.")
 else:
  print(f"{hostname[0]}  '{process}' with {pid[0]}
  process-id is failed stopped.")
else:
 print(f"{hostname[0]}  '{process}' process is not
 started.")
```

Example 8.15: Stop running processes by script

```
from netmiko import Netmiko
from re import findall
host = ["192.168.163.135", "192.168.163.136", "192.168.163.137"]
for ip in host:
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux"}
```

```
command = "ps fax"
process = "gnome-calculator"
net_connect = Netmiko(**device)
output = net_connect.send_command(command)
hostname = findall("@(.*):", net_connect.find_prompt())
pid = findall(f"(\d+).*{process}", output)
if pid:
 net_connect.send_command(f"kill {pid[0]}")
 output = net_connect.send_command(command)
 pid_new = findall(f"(\d+).*{process}", output)
 if not pid_new:
  print(f"{hostname[0]}  '{process}' with {pid[0]} process-id
  is successfully stopped.")
 else:
  print(f"{hostname[0]}  '{process}' with {pid[0]} process-id
  is failed stopped.")
else:
 print(f"{hostname[0]}  '{process}' process is not started.")
```

When we execute the script, we get the following output. We manually enter **`gnome-calculator &`** commands in Server-1 and Server-2. So, the code finds the related PID as process-id from these servers and kills or stops the application. But in Server-3, we don't start this application, so the code cannot find the PID of the **`gnome-calculator`** application. It displays the output **`process is not started`**.

Output:

```
Server-1  'gnome-calculator' with 17204 process-id is successfully
stopped.
Server-2  'gnome-calculator' with 11633 process-id is successfully
stopped.
Server-3  'gnome-calculator' process is not started.
```

Conclusion

This chapter taught us about maintaining and configuring Linux servers with the **`netmiko`** and **`paramiko`** modules. We also checked some basic Linux commands, such as **`ls`**, **`cat`**, **`ifconfig`**, **`top`**. We collected logs, displayed them in the output, saved them to a text or Excel file, and sent them by email. We used connection modules to log in to devices to make some processes in the administrator mode, such as downloading and installing a software package, creating users, or even stopping a process or an application.

The next chapter will focus on creating scripts for network security in both network and system devices. We will make firewall configurations in servers and create access lists in network devices to keep the network secure.

Multiple choice questions

1. Which of the following tools is not a Virtual Machine (VM) tool?
    a. Vmware Player
    b. VirtualBox
    c. Vagrant
    d. KVM
2. Which command shows the CPU and memory levels in the same output in Ubuntu?
    a. `ps`
    b. `top`
    c. `free -m`
    d. `ifconfig`
3. Which command shuts down the server immediately in Ubuntu?
    a. `shutdown`
    b. `shutdown +10`
    c. `shutdown -c`
    d. `shutdown now`

Answers

1. c
2. b
3. d

Questions

1. Write a script to create a new process and find the process ID with the **`ps fax`** command.
2. Write a script to delete a package from the server and verify that it's deleted.
3. Write a script to shut down a server after 1 minute of the script being executed.

CHAPTER 9
Network Security with Python

This chapter will focus on the security features and services of network and system devices. We will use the netmiko module to connect devices to check and configure security services. We will also send alerts if there is a risky configuration in the machine by Python scripts, and we will check the network in the packet base to investigate any issues in depth.

Structure

In this chapter, we will cover the following topics:

- Activate security services
  - Install and activate the "Firewalld" service on servers
  - Configure firewall settings on servers
  - Create access lists in network devices
  - Manipulate network packets with scapy
- Check logs and configurations
  - Check CPU levels periodically with Crontab
  - Check router configuration for insecure password
  - Check port security configuration in routers
- Collect packets from ports with Pyshark

Objectives

We will install the **`firewalld`** service in Linux servers to configure the security features. We will check how to use this service with Linux commands, and we will activate and deactivate it according to our requirements. We will also create new security zones and make configurations to allow for accessing ports, services, and IP addresses.

Further on, we will configure **Access Lists** (**ACLs**) in network devices with the Jinja template. We will also manipulate network packets with the Scapy module and send ICMP request packets, get ICMP reply packets, and check in the Wireshark tool. Additionally, we will collect data from the network devices to check if any risky settings are configured and capture the network packets (with full details) in the pyshark module.

Activate security services

In this section, we focus on security services in network and system devices. We use firewall services, ACLs, or traffic policies to secure the network and system environment to protect from attacks outside.

In the data center, enterprise, or **internet service provider** (**ISP**) networks, all systems and network devices are connected; on the top, they are connected to firewall. Essential security starts at the top of the topology, and Firewall security is another part related to security engineers. This book focuses on initial or device-based security features like ACLs, traffic policies in network devices, or `firewalld` services in Linux servers.

In network devices like routers or switches, we configure ACLs to create a primary security feature in these devices. We can deny or permit both inbound and outbound traffic by developing policies and access lists. There are many options to identify the parameters, such as protocol, **Quality of Service** (**QoS**), and source and destination IP address.

In system devices like Linux servers, we can activate the `firewalld` security service to make the server environment more secure.

Install and activate the "Firewalld" service on servers

We use the `firewalld` service to secure servers in the primary step. It's a Linux OS firewall management tool or service that is also written in the Python language.

- It supports dynamic management firewall systems, including network zones.
- It's based on trust in a network or protocol inside or outside the network.

- It has support for IPv4 and IPv6 address settings.
- We can make changes in this service concurrently when we enter the configurations. So, it's a runtime environment without needing to restart the service or daemon.
- It's already installed on many Linux distros by default, such as CentOS, Fedora, and SUSE. We must download and install the **firewalld** service in Ubuntu to use it.

Before we start using the **firewalld** service, we must install it with the following command:

```
$ sudo apt install firewalld
```

All the **firewalld** service commands require an admin account. So, at the beginning of the command, we should add the **sudo** command to enter the root or admin user in each command entrance. We can also use the **sudo -s** command to change the current user of the terminal to the root account. When we enter it, we don't need to enter the **sudo** command in each line. With the **#** character, we are inside the root or admin user.

```
ubuntu@Server-1:~$ sudo -s
[sudo] password for ubuntu:
root@Server-1:/home/ubuntu#
```

In the following command examples, we will execute all commands as the **root** user so that we don't need to use **sudo** at the beginning of the line. If we enter the following commands in the standard user view in which we have a user account of a **ubuntu** name, we must enter the **sudo** command at the beginning of the line.

We can start the service with the **systemctl start firewalld** command. With the **firewall-cmd --state** command, we can verify whether or not the **firewalld** service starts. If the output of this command is **running**, this service is running or activated. If the output is **not running**, this service is not running or is deactivated. With this command, we can check the status of the **firewall** service, and it only shows whether it is running.

```
root@Server-1:/home/ubuntu# systemctl start firewalld
root@Server-1:/home/ubuntu# firewall-cmd --state
running
```

The **systemctl** command is a Linux service management command to check any of the services installed on the server. We can check any service status with this command. If we enter this command, it displays a large output of the service status as a summary. We can also check each service's status by writing the **`systemctl status SERVICE_NAME`** command.

We need to use **`systemctl status firewalld`** to check the details of the **`firewalld`** service. It shows the firewall details, such as the directory it is loaded from, an active state in the **`active (running)`** state, process ID as PID, tasks inside this service, and resource usage of this service from the server CPU and memory. In the end, the logs include the time of the record and the log information. We can check the service starting time from this log.

```
root@Server-1:/home/ubuntu# systemctl status firewalld
  firewalld.service - firewalld - dynamic firewall daemon
  Loaded: loaded (/lib/systemd/system/firewalld.service;
  enabled; vendor preset: enabled)
  Active: active (running) since Sat 2022-09-10 16:42:31 +03;
  3s ago
   Docs: man:firewalld(1)
   Main PID: 2625 (firewalld)
   Tasks: 2 (limit: 4584)
   Memory: 22.6M
  CPU: 672ms
   CGroup: /system.slice/firewalld.service
     └─2625 /usr/bin/Python3 /usr/sbin/firewalld --nofork --
     nopid
Aug 10 16:42:31 Server-1 systemd[1]: Starting firewalld -
dynamic firewall daemon…
Aug 10 16:42:31 Server-1 systemd[1]: Started firewalld -
dynamic firewall daemon.
```

With a similar usage of the systemctl command, we can stop any service by writing **`systemctl stop SERVICE_NAME`**. In the following command, we write the **`systemctl stop firewall`** command to stop the firewall service. After that, we can check the **`not running`** states. We can also check the details of the **`systemctl status firewalld`** command output. The active state is **`inactive (dead`**) after we stop the service. At the end of the output, we can find the logs about the service when it starts and stops. This log section keeps a maximum of five lines, so we can see only these lines in the output:

```
root@Server-1:/home/ubuntu# systemctl stop firewalld
root@Server-1:/home/ubuntu# firewall-cmd --state
not running
root@Server-1:/home/ubuntu# systemctl status firewalld
  firewalld.service - firewalld - dynamic firewall daemon
  Loaded: loaded (/lib/systemd/system/firewalld.service;
  enabled; vendor preset: enabled)
  Active: inactive (dead) since Sat 2022-09-10 16:53:10 +03; 8s
  ago
   Docs: man:firewalld(1)
  Process: 2799 ExecStart=/usr/sbin/firewalld --nofork --nopid
  (code=exited, status=0/SUCCESS)
   Main PID: 2799 (code=exited, status=0/SUCCESS)
   CPU: 389ms
Aug 10 16:45:52 Server-1 systemd[1]: Starting firewalld -
dynamic firewall daemon…
Aug 10 16:45:52 Server-1 systemd[1]: Started firewalld -
dynamic firewall daemon.
Aug 10 16:53:10 Server-1 systemd[1]: Stopping firewalld -
dynamic firewall daemon…
Aug 10 16:53:10 Server-1 systemd[1]: firewalld.service:
Deactivated successfully.
Aug 10 16:53:10 Server-1 systemd[1]: Stopped firewalld -
dynamic firewall daemon.
```

We can also enable or disable the firewall service on the boot of the server. So, we can decide whether the service is enabled when we start the server. We can use the generic **`systemctl enable/disable SERVICE_NAME`** command to enable or disable a service in the boot. So, we use the following commands to enable or disable the **`firewalld`** service on the boot.

We can check whether the setting is successfully done by listing the services with the **`systemctl`** command. The output of the following command is long, so we filter the necessary service with its name by writing **`grep firewalld`** after the pipeline character. We also note the **`type`** option as a service to display only the services.

```
root@Server-1:/home/ubuntu# systemctl enable firewalld
```

```
root@Server-1:/home/ubuntu# systemctl list-unit-files --
type=service | grep firewalld
firewalld.service                          enabled
enabled
root@Server-1:/home/ubuntu# systemctl disable firewalld
Removed /etc/systemd/system/multi-
user.target.wants/firewalld.service.
Removed /etc/systemd/system/dbus-
org.fedoraproject.FirewallD1.service.
root@Server-1:/home/ubuntu# systemctl list-unit-files --
type=service | grep firewalld
firewalld.service                          disabled      enab
led
```

In the output, we can see that the first word after the service name is the service status on the boot. If we enabled the service, it's **enabled**; otherwise, it's **disabled**.

After introducing the **firewalld** service in the Linux systems, we can start writing the active automation code and make changes to this service in Python. In *Example 9.1*, we create five functions to handle the previous processes by script. We develop functions to log in to servers, start or stop the firewalld service, and enable or disable the service on the boot. We use the concurrent feature of the Python language to make changes in all the servers simultaneously.

In *Table 9.1*, we create two files in the same directory as the script: **stop_firewalld.txt** and **start_firewalld.txt**. We write the previous commands in these text files to start or stop the service and check the status. We also add a package installation command in the **start_firewalld.txt** file to install the service before we activate it.

stop_firewalld.txt systemctl stop firewalld systemctl status firewalld firewall-cmd --state	start_firewalld.txt apt-get install firewalld -y systemctl start firewalld systemctl status firewalld firewall-cmd --state

Table 9.1: *Output of the "stop_firewalld.txt" and "start_firewalld.txt" Files*

In *Example 9.1*, we install firewalld service, and change status of this service in runtime and on boot

1. We import the necessary functions from the **netmiko** and **concurrent** modules. After that, we write all server management IP addresses in a variable.

```
from netmiko import Netmiko
from concurrent.futures import ThreadPoolExecutor
host = ["192.168.163.135", "192.168.163.136",
"192.168.163.137"]
```

2. We create a function to make the connection to the Linux servers and write a parameter as the **ip** we use when we get all items in order from the **host** variable. We enter the device connection information, including the **secret** parameter to log in to netmiko in the **root** or **admin** user. After that, we call the **Netmiko** function to log in to the devices and assign **Netmiko** functionto the **net_connect** variable. At the last line of this function, we return the **net_connect** variable, which we call in the following functions to log in to the devices:

```
def server_connection(ip):
device = {"host": ip, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux", "secret": "ubuntu"}
 net_connect = Netmiko(**device)
 return net_connect
```

3. In the following code, we create functions to start or stop the **firewalld** function in the connected servers. We use the **ip** parameter again for this function. We call the **server_connection** function to log in to the devices before we start or stop the service, and we assign the output of this function to the **connection** variable. The output of the **server_connection** function is the value of the return code, which is the **net_connect** variable. So, the **connection** variable equals the **net_connect** variable in the **server_connection** function.

   Now, we are inside the function and can send the configurations on the files with the **send_config_from_file** from the **netmiko** module. We write text files that we created in *Table 9.1*. We also add a timeout to **execute** commands more reliably. Finally, we display the output of the command with the print function.

```
def start_firewalld(ip):
 connection = server_connection(ip)
```

```
output =
connection.send_config_from_file("start_firewalld.txt",
read_timeout=1000)
 print(output)
def stop_firewalld(ip):
 connection = server_connection(ip)
output =
connection.send_config_from_file("stop_firewalld.txt",
read_timeout=1000)
 print(output)
```

4. In the following code, we create two functions to enable or disable any of the services in the **systemctl**. We create a variable called **service** to match the service we change on the boot. After that, we again call the **server_connection** function and assign it to the **connection** variable. So, the value of this variable is the **net_connect** variable in the **server_connection** function. After that, we run the **systemctl disable/enable SERVICE_NAME** command with the timeout parameter. After that, we also check the status of the service in the previous commands and display the output of **systemctl** command by filtering the service name.

```
def disable_service_on_boot(ip):
 service = "firewalld"
 connection = server_connection(ip)
connection.send_config_set(f"systemctl disable
{service}", read_timeout=1000)
output = connection.send_command(f"systemctl list-unit-
files --type=service | grep {service}")
 print(output)
def enable_service_on_boot(ip):
 service = "firewalld"
 connection = server_connection(ip)
connection.send_config_set(f"systemctl enable {service}",
read_timeout=1000)
output = connection.find_prompt() +
connection.send_command(f"systemctl list-unit-files --
type=service | grep {service}")
 print(output)
```

5. In the last part, we call the **`ThreadPoolExecutor`** function from the concurrent module. We set the value of the **`max_workers`** parameter as **`5`**. So, in each process, our script logs in to a maximum of five devices. Finally, we use the map function to call the specific function, which can be any of the previous functions we created, and the **`host`** variable. In the following code, we add the **`enable_service_on_boot`** function. So, when we execute the code, the script runs **`enable_service_on_boot`** to enable the **`firewalld`** service on the boot.

```
with ThreadPoolExecutor(max_workers=5) as executor:
  result = executor.map(enable_service_on_boot, host)
```

Example 9.1: Installing "firewalld" service, and changing status in runtime and on boot

```
from netmiko import Netmiko
from concurrent.futures import ThreadPoolExecutor
host = ["192.168.163.135", "192.168.163.136",
"192.168.163.137"]
def server_connection(ip):
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux", "secret": "ubuntu"}
  net_connect = Netmiko(**device)
  return net_connect
def start_firewalld(ip):
  connection = server_connection(ip)
  output =
  connection.send_config_from_file("start_firewalld.txt",
  read_timeout=1000)
  print(output)
def stop_firewalld(ip):
  connection = server_connection(ip)
  output =
  connection.send_config_from_file("stop_firewalld.txt",
  read_timeout=1000)
  print(output)
def disable_service_on_boot(ip):
  service = "firewalld"
  connection = server_connection(ip)
```

```
  connection.send_config_set(f"systemctl disable {service}",
  read_timeout=1000)
  output = connection.send_command(f"systemctl list-unit-files
  --type=service | grep {service}")
  print(output)
def enable_service_on_boot(ip):
  service = "firewalld"
  connection = server_connection(ip)
  connection.send_config_set(f"systemctl enable {service}",
  read_timeout=1000)
  output = connection.find_prompt() +
  connection.send_command(f"systemctl list-unit-files --
  type=service | grep {service}")
  print(output)
with ThreadPoolExecutor(max_workers=5) as executor:
  result = executor.map(enable_service_on_boot, host)
```

Configure firewall settings on servers

We can also create security zones in the **firewalld** service. The **firewalld** package automatically creates predefined zones, such as block, dmz, home, internal, and trusted, after we start the **firewalld** service. Let's check the details of some zones:

- The **public** zone is used for the public areas we don't trust on the computers, which is not harmful to our network.
- The**trusted** zone is used for all accepted network connections.
- The **drop** zone drops incoming packets to our server. All network packets are dropped, so there is no reply to these packets.

When we install and start the **firewalld** service with the **systemctl start firewalld** command, we can run the **firewall-cmd --get-zones** command to find all the zones in the Linux server.

```
root@Server-1:/home/ubuntu# firewall-cmd --get-zones
block dmz drop external home internal nm-shared public trusted
work
```

There are many options in the **firewall-cmd** command, as listed in *Table 9.2*. You can check more details about the parameters on the official firewalld website.

Command	Parameter	Description
firewall-cmd	**--version**	Check service version
firewall-cmd	**--state**	Check service status
firewall-cmd	**--get-zones**	List all zone names
firewall-cmd	**--list-all-zones**	List all zones with details
firewall-cmd	**--get-default-zone**	Find default zone
firewall-cmd	**--set-default-zone=internal**	Change default zone
firewall-cmd	**--get-zone-of-interface=ens33**	Check zone as interface-based
firewall-cmd	**--list-all --zone=public**	Show the details of the specified zone
firewall-cmd	**--get-active-zones**	List of active zones

Table 9.2: *Examples of the "firewall-cmd" command*

We can open and close ports in specific zones, and it's up to our requirements from the zones. We can open a port if we want to get traffic flow from that port number. We need to write the port number and the service name. For example, the **Hypertext Transfer Protocol (HTTP)** protocol's port number is **80**. When we write HTTP, we use the **--add-service** parameter to add the service. When we write the port number of the HTTP, we write **--add-port** with **80** to add the port. We must mention the protocol as UDP or TCP after the port number as **PORT_NUMBER/PROTOCOL**. Both commands' output is **successful** if the firewall configuration is successfully configured. We add **http** as a service and **80/tcp** as a port in the following code:

```
root@Server-1:/home/ubuntu# firewall-cmd --zone=public --add-service=http
success
root@Server-1:/home/ubuntu# firewall-cmd --zone=public --add-port=80/tcp
success
```

When we check the details of the **public** zone, we can see **http** in the **services** line and **80/tcp** in the **ports** line.

```
root@Server-1:/home/ubuntu# firewall-cmd --list-all --
zone=public
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens33
  sources:
  services: dhcpv6-client http ssh
  ports: 80/tcp
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

We use the **--remove-service** and **--remove-port** parameters to remove the service and port information from a zone when the service or port is not required to be accepted.

```
root@Server-1:/home/ubuntu# firewall-cmd --zone=public --
remove-service= http
success
root@Server-1:/home/ubuntu# firewall-cmd --zone=public --
remove-port=80/tcp
success
```

If we log in to the server via an SSH tool like **Secure Crt** or **Putty**, we enter via active zone with SSH service. If we enter the following command, it removes the SSH service. So, if we try to create a new session to the server in these SSH tools, it's rejected because we remove the SSH service from the active default zone of the server. The current session is not affected, and we are still connected to the server.

```
root@Server-1:/home/ubuntu# firewall-cmd --zone=public --
remove-service=ssh
success
```

We can open the SSH service again with the **--add-service** parameter.

```
root@Server-1:/home/ubuntu# firewall-cmd --zone=public --add-service=ssh
success
```

There are two essential parameters in the **firewall-cmd** command. The **--permanent** parameter makes the firewall configuration updates or changes permanent even after the server reboots. This parameter combines with other parameters.

```
root@Server-1:/home/ubuntu# firewall-cmd --permanent --zone=public --add-port=80/tcp
success
```

We have another parameter to reload the firewall to update the changes in the firewall configuration runtime. We use the **--reload** parameter to achieve this goal. It's recommended to use this parameter at the end of a configuration change in a new line, such as the **firewall-cmd --reload** command, without any other parameters.

```
root@Server-1:/home/ubuntu# firewall-cmd --reload
success
```

We can create a new zone with the **--new-zone** parameter. We should use the **--permanent** parameter in this line if we need this zone permanently, even after rebooting the server. We create a zone named **test123**. After we reload the configuration, it's created. We can check zone list with the **firewall-cmd --get-zones** command. We can see the new zone listed in the following output:

```
root@Server-1:/home/ubuntu# firewall-cmd --new-zone=test123 --permanent
success
root@Server-1:/home/ubuntu# firewall-cmd –reload
success
root@Server-1:/home/ubuntu# firewall-cmd --get-zones
block dmz drop external home internal nm-shared public test123 trusted work
```

We have more configuration parameters in the **firewall-cmd** command, as listed in *Table 9.3*:

Command	Parameter	Description

firewall-cmd	**PARAMETER --permanent**	Change configuration permanently
firewall-cmd	**--reload**	Reload the firewall configuration
firewall-cmd	**--new-zone**	Create a new zone
firewall-cmd	**--zone=home --change-interface=eth0**	Change or add the zone to an interface
firewall-cmd	**--zone="public" --add-forward-port=port=80:proto=tcp:toport=8080**	Add the forwarding port with port number, protocol, and the destination port
firewall-cmd	**--list-all --zone=public**	Show the details of the specified zone

Table 9.3: *Examples of the Zone Configuration Commands*

In *Example 9.2*, we add a new zone in a single server. Then, we make changes in the zone configuration, such as adding an interface, service, port, and forwarding port. After that, we display the new zone with the details to see the new configurations. We use the Jinja2 template with the YAML file. In *Table 9.4*, we create two files: **info.yaml** and **command_list.txt**. We enter the entire command in the text file, except for the specific values in the YAML file. Instead of this, we write the key inside double curly brackets **{{ KEY }}**. In the following files, we only use the **new_zone** key, but we use the **interface** and **service** keys in the script.

info.yaml	**command_list.txt**
new_zone: test interface: lo service: http	**firewall-cmd --new-zone={{new_zone}} --permanent firewall-cmd --reload firewall-cmd --get-zones**

Table 9.4: *Output of the "info.yaml" and "command_list.txt" Files*

In the *Example 9.2* we configure Linux firewall zones with the jinja template.

1. We import the necessary functions from the **netmiko**, **RE**, **jinja2**, and **yaml** modules.

```
from netmiko import Netmiko
from re import split
```

```
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
```

2. We create a function to get the data from the YAML file and combine or render this file with the Jinja template.

```
def from_jinja():
 env = Environment(loader=FileSystemLoader("."))
 template = env.get_template("command_list.txt")
 with open("info.yml") as r:
  data = safe_load(r)
 command = split("\n", template.render(data))
 return [command, data]f
```

We return two values: **command** and **data**. The **command** variable is the full command to add the data from the YAML file; each item has different commands. And the **data** variable is a Python dictionary variable.

```
Output of the "command": ['firewall-cmd --new-
zone=cccccccc --permanent', 'firewall-cmd --reload',
'firewall-cmd --get-zones']
Output of the "data": {'new_zone': 'cccccccc'}
```

3. In the following function, we create a new zone for the server. To do that, we connect with the **Netmiko** function and send the **commands** variable output. So, we need to call the first item or **item-0** of the **from_jinja** function. We return a list in that function, so the first item is the **commands** variable. If we write **from_jinja()[0]**, we assign all the configurations to the new **commands** variable. After that, we send the configuration commands to the device. The new zone must be created in this step. Then, we return the **net_connect** variable, which we use to log in to the server to configure the new zone in the following function: **zone_configuration**.

```
def add_zone(ip):
device = {"host": ip, "username": "ubuntu", "password":
"ubuntu", "device_type": "linux", "secret": "ubuntu"}
 net_connect = Netmiko(**device)
 commands = from_jinja()[0]
 net_connect.send_config_set(commands, read_timeout=1000)
 return net_connect
```

4. In the following function, we configure the new zone with the following commands:

```
firewall-cmd --reload
firewall-cmd --get-active-zones
firewall-cmd --zone={new_zone} --add-service=http
firewall-cmd --zone={new_zone} --add-port=80/tcp
firewall-cmd --zone={new_zone} --add-forward-
port=port=80:proto=udp:toport=8080:toaddr=10.10.10.1
```

   We get the `new_zone` value from the YAML file. First, we call the `from_jinja()[2]` function with the second item of the returned list. It's the `data` variable, a dictionary of a single item with a key and its value. We need to get its value, so we assign `data[new_zone]` as a value to the `new_zone` variable. According to the YAML file, the value of `new_zone` in the following function is `test`, so our new zone name is `test`.

```
def zone_configuration(ip):
 interface = "lo"
 data = from_jinja()[1]
 new_zone = data["new_zone"]
commands = [f"firewall-cmd --permanent --zone={new_zone}
--change-interface={interface}",
     "firewall-cmd --reload",
     "firewall-cmd --get-active-zones",
  f"firewall-cmd --zone={new_zone} --add-service=http",
  f"firewall-cmd --zone={new_zone} --add-port=80/tcp",
  f"firewall-cmd --zone={new_zone} --add-forward-
  port=port=80:proto=udp:toport=8080:toaddr=10.10.10.1"
     ]
```

5. We use the `add_zone` function to connect to the server in the same function. Then, we send the commands by the `send_config_set` function from the `netmiko` module. Finally, we check the new zone detailed configuration with the `sudo firewall-cmd --list-all --zone={new_zone}` command.

```
 connect = add_zone(ip)
 connect.send_config_set(commands)
out=connect.send_config_set(f"sudo firewall-cmd --list-
all --zone={new_zone}")
```

```
print(out)
```

6. At the end, we call the **zone_configuration** function.

```
zone_configuration("192.168.163.135")
```

Example 9.2: Firewall Zone Configuration with the jinja Template

```
from netmiko import Netmiko
from re import split
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
def from_jinja():
  env = Environment(loader=FileSystemLoader("."))
  template = env.get_template("command_list.txt")
  with open("info.yml") as r:
   data = safe_load(r)
  command = split("\n", template.render(data))
  return [command, data]
def add_zone(ip):
  device = {"host": ip, "username": "ubuntu", "password":
  "ubuntu", "device_type": "linux", "secret": "ubuntu"}
  net_connect = Netmiko(**device)
  commands = from_jinja()[0]
  net_connect.send_config_set(commands, read_timeout=1000)
  return net_connect
def zone_configuration(ip):
  interface = "lo"
  data = from_jinja()[1]
  new_zone = data["new_zone"]
  commands = [f"firewall-cmd --permanent --zone={new_zone} --
  change-interface={interface}",
      "firewall-cmd --reload",
      "firewall-cmd --get-active-zones",
      f"sudo firewall-cmd --zone={new_zone} --add-
      service=http",
      f"sudo firewall-cmd --zone={new_zone} --add-port=80/tcp",
       f"firewall-cmd --zone={new_zone} --add-forward-
       port=port=80:proto=udp:toport=8080:toaddr=10.10.10.1"
      ]
```

```
  connect = add_zone(ip)
  connect.send_config_set(commands)
  out=connect.send_config_set(f"sudo firewall-cmd --list-all --
  zone={new_zone}")
  print(out)
zone_configuration("192.168.163.135")
```

When we execute the script in *Example 9.2*, we get the following output. The zone name is listed in the first line as **`test`**. In the interfaces, we set the loopback interface **`lo`**. In the services, we have the **`http`** service enabled. In the ports, we have **`80/tcp`** and a forwarding port configuration in **`forward-ports`**.

Output:

```
test (active)
  target: default
  icmp-block-inversion: no
  interfaces: lo
  sources:
  services: http
  ports: 80/tcp
  protocols:
  forward: no
  masquerade: no
  forward-ports:
  port=80:proto=udp:toport=8080:toaddr=10.10.10.1
  source-ports:
  icmp-blocks:
  rich rules:
```

Create access lists in network devices

In network devices like routers or switches, we can secure the inbound and outbound traffic with the ACLs. It's the primary protection to secure network devices from outside attacks. We can configure the ACLs with many parameters. ACLs are based on permitting or denying networks by checking the network packets. We can add protocol, source, and destination IP addresses.

In *Example 9.3*, we use nornir framework with the **`jinja2`** module to render the commands with the YAML file. We need to enter the following commands in the Cisco routers.

```
configure terminal
ip access-list extended 100
10 deny ip 10.11.0.0 0.0.255.255 192.32.0.0 0.0.255.255
20 deny ip 15.12.0.0 0.0.255.255 192.12.0.0 0.0.255.255
30 permit ip 120.1.0.0 0.0.255.255 192.168.0.0 0.0.255.255
40 permit tcp 10.1.1.0 0.0.0.255 172.16.1.0 0.0.0.255 eq telnet
50 permit ip 140.1.0.0 0.0.255.255 192.168.0.0 0.0.255.255
90 deny pim any any dscp cs1
100 deny ip any any
```

The format of the rules is in *Table 9.5*. We divide each part into sections to understand it easier. If we divide each code into pieces, we have a sequence number, permission value as **`permit`** or **`deny`**, protocol name, source, and destination IP address. In some commands, we have an optional parameter with its value, such as **`eq telnet`** or **`dscp cs1`**.

Sequence Number	Permit / Deny	Protocol	Source IP	Destination IP	Parameter	Value
10	deny	ip	10.11.0.0 0.0.255.255	192.32.0.0 0.0.255.255		
20	deny	ip	15.12.0.0 0.0.255.255	192.12.0.0 0.0.255.255		
30	permit	ip	120.1.0.0 0.0.255.255	192.168.0.0 0.0.255.255		
40	permit	tcp	10.1.1.0 0.0.0.255	172.16.1.0 0.0.0.255	eq	telnet
50	permit	ip	140.1.0.0 0.0.255.255	192.168.0.0 0.0.255.255		
90	deny	pim	any	any	dscp	cs1
100	deny	ip	any	any		

Table 9.5: *Example 9.3 ACL Rules*

In the following code, we create a **`command.txt`** text file. Inside it, we enter the commands that we execute in the device. But for each value, we use **`{{`**

}} double curly brackets and write a key inside it. We get the values of these keys from the YAML file. In the first line, we enter the ACL 100 with the **ip access-list extended ACL_NUMBER** command. We get the ACL number from the YAML file. In the following lines, we use **jinja2** for a loop because all remaining ACL rules are similar; only the parameters change. We write the following line with each parameter one-by-one. We also write the **if** condition for the last two keys as **parameter** and **value**, which are only used in rules with sequence numbers 40 and 90. We already saw examples with the **for** and **if** conditions. We also enter a **show** command to see the configuration output at the end of the file.

commands.txt

```
ip access-list extended {{acl_number}}
{% for acl in access_lists %}
{{acl["sequence"]}} {{acl["permission"]}} {{acl["protocol"]}}
{{acl["source_ip"]}} {{acl["dest_ip"]}} {% if acl['eq'] %}eq{%
endif %} {{acl["eq"]}} {% if acl['dscp'] %}dscp{% endif %}
{{acl["dscp"]}}
{% endfor %}
do show access-lists {{acl_number}}
```

After configuring the text file, we add the keys and values in the YAML file as **info.yaml** in the same directory as the script and text file. We have a dictionary named **acl_number**, representing the ACL number of our configuration as 100. After that, we have the **access_lists** key. A **-** dash characterizes its value as items on a list. We add all values inside these items with their keys, so when the jinja renders both files, it creates a configuration batch as expected:

```
info.yaml
acl_number: 100
access_lists:
- sequence: 10
 permission: deny
 protocol: ip
 source_ip: 10.11.2.0 0.0.255.255
 dest_ip: 192.32.1.0 0.0.255.255
- sequence: 20
 permission: deny
 protocol: ip
 source_ip: 15.12.2.0 0.0.255.255
 dest_ip: 192.12.1.0 0.0.255.255
- sequence: 30
```

```
- sequence: 40
 permission: permit
 protocol: tcp
 source_ip: 10.1.1.0 0.0.0.255
 dest_ip: 172.16.1.0 0.0.0.255
 eq: telnet
- sequence: 50
 permission: permit
 protocol: ip
 source_ip: 140.1.2.0 0.0.255.255
 dest_ip: 192.168.1.0 0.0.255.255
- sequence: 90
 permission: deny
 protocol: pim
```

permission: permit protocol: ip source_ip: 120.1.2.0 0.0.255.255 dest_ip: 192.168.1.0 0.0.255.255	source_ip: any dest_ip: any dscp: cs1 - sequence: 100 permission: deny protocol: ip source_ip: any dest_ip: any

Table 9.6: *Output of the "info.yaml" File*

The script of the nornir file is similar to the previous examples. We only change the Jinja file format with the YAML file. In the configuration, we render both text and YAML file and assign its output to the **`config`** variable. Inside it, we have the absolute configuration that needs to be sent to the devices. With the nornir framework, we log in to the devices simultaneously and execute the commands with the **`netmiko_send_config`** function from the **`nornir_netmiko`** module. After that, we display the output of the commands we run on the devices.

Example 9.3: Creating ACLs in network devices

```
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_config
from jinja2 import Environment, FileSystemLoader
from yaml import safe_load
from re import split
env = Environment(loader=FileSystemLoader("."))
template = env.get_template("commands.txt")
with open("info.yml") as r:
  data = safe_load(r)
config = split("\n",template.render(data))
connect = InitNornir()
result = connect.run(task=netmiko_send_config,
config_commands=config)
print_result(result)
```

Manipulate network packets with scapy

We can sniff, scan and forge network packets with the third-party **`scapy`** module in Python. It can probe, scan, or attack networks, and it's a powerful packet manipulation program with a significant usage area in the security part of networking. We must download the module with the **`pip install scapy`** command in the terminal of the IDE or on the PC on which your script is located. You can check out more details about the **`scapy`** module at the following official documentation:

https://scapy.readthedocs.io/

There are some basic functions in scapy to use in network automation.

- **ARP**: We can create **Address Resolution Protocol** (**ARP**) packets with the scapy module. We use the **`ARP`** function to create the ARP request packets to any network device or IP address.
- **Ether**: The **`ether`** function creates an ethernet packet.
- **Srp**: The **`srp`** function sends and receives packets at layer 2. We use it to scan the network.
- **Summary**: The **`summary`** function displays the status of the packets we create with the scapy module. It gives a piece of basic information, such as packet type or destination data.

  ```
  from scapy.all import *
  request = ARP()
  print(request.summary())
  ```

  Output: **`ARP who has 0.0.0.0 says 192.168.1.25`**

  Home Wi-Fi IP information from the **`ipconfig`** command output in Windows Terminal:

  ```
  Wireless LAN adapter Wi-Fi:
    Connection-specific DNS Suffix  . : home
    IPv4 Address. . . . . . . . . . . : 192.168.1.25
    Subnet Mask . . . . . . . . . . . : 255.255.255.0
    Default Gateway . . . . . . . . . : 192.168.1.1
  ```

- **Show**: The **`show`** function has more detailed packet information as compared to the **`summary`** function.

  ```
  ###[ ARP ]###
    hwtype     = 0x1
    ptype      = IPv4
    hwlen      = None
    plen       = None
    op         = who-has
  ```

```
hwsrc     = ac:67:5d:9f:24:6e
psrc      = 192.168.1.25
hwdst     = 00:00:00:00:00:00
pdst      = 0.0.0.0
```

In *Example 9.4*, We send an **Internet Control Message Protocol** (**ICMP**) packet to the Google DNS IP address `8.8.8.8` with the `scapy` module:

1. We use multiple functions from the `scapy` module. We can import each function by writing its name with a dividing comma or by adding the `*` character after import. This character means '*import all the functions from the module*'. So, we don't need to write all the necessary functions to import.

   ```
   from scapy.all import *
   ```

2. We enter the destination IP address for the ICMP packet and sequence number.

   ```
   ip_address = IP(dst="8.8.8.8")
   icmp_sequence= ICMP(seq=1111)
   ```

3. Network packets are built with a block of packets, which are layers. In scapy, we use the `/` character to stack layers in order.

   ```
   pckt= ip_address / icmp_sequence
   ```

4. In the final part, we send the created packets with the `send` function.

   ```
   send(pckt)
   ```

When we execute the script, it sends an ICMP request packet to the `8.8.8.8` IP address with the `1111` sequence number on the internet. If the IP address is reachable, that device returns an ICMP reply packet with the same sequence number.

Output:

```
.
Sent 1 packets.
```

Example 9.4: Sending ICMP packet with scapy

```
from scapy.all import *
ip_address = IP(dst="8.8.8.8")
icmp_sequence= ICMP(seq=1111)
pckt= ip_address / icmp_sequence
send(pckt)
```

When we execute the script, it sends an ICMP request packet to the **`8.8.8.8`** IP address with the **`1111`** sequence number on the internet. If the IP address is reachable, that device returns an ICMP reply packet with the same sequence number.

We need to install the **`Wireshark`** tool to see the ICMP request and reply packets. It's an open-source packet capture program that we can download and install from its official web page at the following link:

https://www.wireshark.org/#download

After the installation, when we open the tool, we must choose the correct adaptor name that connects our PC to the internet, as shown in *Figure 9.1*. If you are using a Wi-Fi connection, it's a **`Wi-Fi`** adaptor.

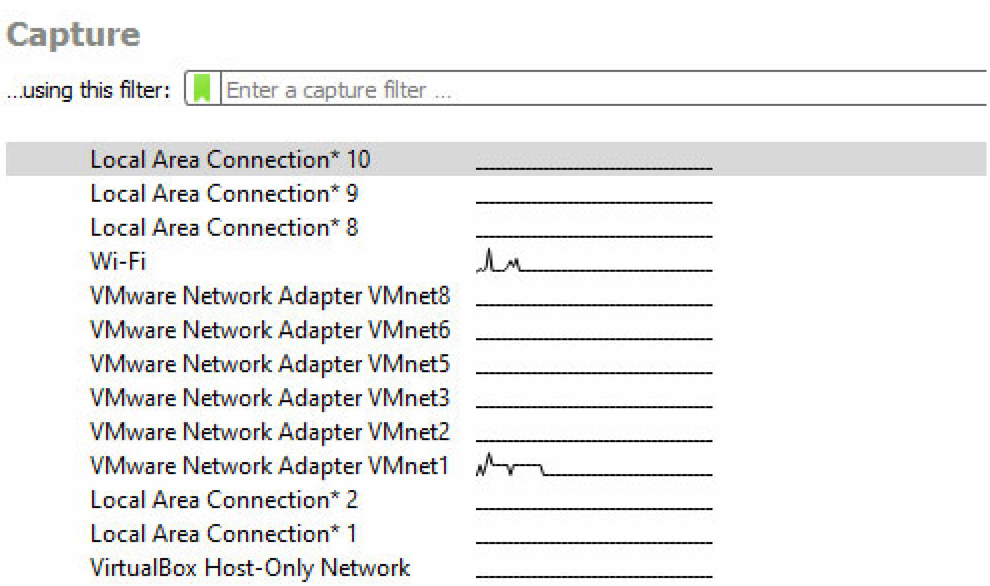


Figure 9.1: *PC Adaptor List in Wireshark*

After entering the adaptor, we can execute the script. We can see the ICMP request and ICMP reply packets to the **`8.8.8.8`** IP address in *Figure 9.2*. The source IP address is our Wi-Fi adaptor IP address, **`192.168.1.25`**, as a private IP address. It can be different on your PC. You can check the Wi-Fi IP address via **`ipconfig`** in Windows or **`ifconfig`** in Linux or Mac. Protocol as ICMP and packet type as request or reply messages are shown in *Figure 9.2*. We can also see the sequence number we set as **`1111`**.

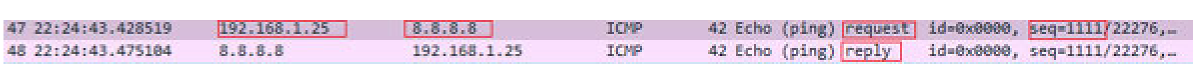


Figure 9.2: *Output of the Wireshark*

In *Example 9.5*, we scan the network to find the host information, such as the IP address and the MAC (physical or hardware) address. If our Wi-Fi network subnet is in **`192.168.1.1/24`**, we can check all IP addresses with ARP requests from **`192.168.1.0/24`** to **`192.168.1.255/24`** with 256 IP addresses. If we change the subnet mask to **`/16`**, code checks 256*256 hosts with ARP request. But it takes too much time to check it, so it's better to use a **`/24`** mask. With *Example 9.5*, we can check all devices connected to our local home modem or any other local network.

1. We import all the functions from the **`scapy`** module.

   ```
   from scapy.all import *
   ```

2. We call two functions: **`ARP`** and **`Ether`**. We use the **`ARP`** function to send ARP requests to a network address, **`pdst`** to choose the destination network, and the Ether function to create an ethernet packet. We chose destination MAC address as **`ff:ff:ff:ff:ff:ff`**, which is a broadcast. You can check the details of the ARP packet process on the internet.

   ```
   arp = ARP(pdst="192.168.1.1/24")
   ether = Ether(dst="ff:ff:ff:ff:ff:ff")
   ```

3. We stack the ether and arp variables in order. After that, we use the **`srp`** function to send and receive packets at layer 2 in the networking layers. We add a **`broadcast`** variable with the timeout parameter as 1. We call the first item of this list and assign it to a **`host`** variable.

   ```
   broadcast = ether / arp
   hosts = srp(broadcast, timeout=1)[0]
   ```

4. Finally, we try to get the IP and MAC address of the devices in the same network as **`192.168.1.0/24`**. We use a **`for`** loop to get the data. We collect IP addresses with the **`psrc`** value and MAC addresses with the **`hwsrc`** value.

   ```
   for device in hosts:
    print(f"IP: {device[1].psrc}    MAC: {device[1].hwsrc}")
   ```

Example 9.5: Network scanner with scapy

```
from scapy.all import *
arp = ARP(pdst="192.168.1.1/24")
ether = Ether(dst="ff:ff:ff:ff:ff:ff")
broadcast = ether / arp
```

```
hosts = srp(broadcast, timeout=1)[0]
for device in hosts:
  print(f"IP: {device[1].psrc}    MAC: {device[1].hwsrc}")
```

When we execute the script, it sends ARP request packets to all IP addresses in the subnet. So, it sends 256 packets in *Example 9.5*. The IP and MAC address information is listed in the following output. MAC addresses should be shown, including digits and alphabetic characters. The following output is just an example.

Output:

```
Begin emission:
Finished sending 256 packets.
…
*………………..*…………………………………………………………………….*……….*……………………………
………………..*……………………………………………….
Received 259 packets, got 5 answers, remaining 254 packets
IP: 192.168.1.1    MAC: XX:XX:XX:XX:XX:XX
IP: 192.168.1.25    MAC: XX:XX:XX:XX:XX:XX
IP: 192.168.1.124    MAC: XX:XX:XX:XX:XX:XX
IP: 192.168.1.133    MAC: XX:XX:XX:XX:XX:XX
IP: 192.168.1.228    MAC: XX:XX:XX:XX:XX:XX
```

Check logs and configurations

In this part, we focus on collecting logs to secure our network devices in our environment. We create periodic tasks to execute Python scripts to contain CPU levels of devices to check if there is a risk to them. We check the insecure password or unencrypted data in the Cisco devices to alert the network administrator, and we also check whether the port security configuration on routers exists.

Check CPU levels periodically with Crontab

We collected and saved the CPU levels in the earlier chapters. We collected all the logs at once. For some logs, we need to collect periodically, such as CPU levels. We can create a script to contain the CPU levels every 5 minutes with an infinite `while` loop, so the code collects CPU levels every 5 minutes. But this is not a good choice to execute an endless script. In this situation, we can get help from operating system schedulers. We can use the **`Task Scheduler`** in Windows and the **`crontab`** service in Linux or MAC. In such periodical tasks, crontab service is essential. We focus on this service in *Example 9.6*. You can check the internet for Windows's **`Task Scheduler`** service.

We need to configure some settings in the Linux machine. We need to make an SSH connection from the Linux machine to the Cisco routers. If you can log in, you can continue to set up the crontab service.

1. We need to run Linux commands in the admin account, so we change our user to the **`root`** user. After that, we enter the **`Desktop`** directory and create the Python file in this directory; you can create it in any directory on your Linux system. Our CPU-level collection code needs a third-party module to install the **`netmiko`** module. So, in the same **`script`** directory, we run the **`pip install netmiko`** command to install it.

```
ubuntu@Server-1:~$ sudo su
[sudo] password for ubuntu:
root@Server-1:/home/ubuntu/Desktop# cd
/home/ubuntu/Desktop
root@Server-1:/home/ubuntu/Desktop# pip install netmiko
```

2. We create a file in the same directory as **`cpu_levels_periodically.py`**, a Python file. When we run **`ls -l`** after we create the file, we can see the file in the following output.

```
root@Server-1:/home/ubuntu/Desktop# touch
cpu_levels_periodically.py
root@Server-1:/home/ubuntu/Desktop# ls -l
total 20
-rw-r--r-- 1 root root 1294 Eyl 13 13:52
cpu_levels_periodically.py
```

3. Crontab task scheduler service is running tasks periodically. In this example, our job is to execute a Python file with the Python3 tool. Usually, if we want to execute a Python file in Python version 3, we need to write **`Python3 FILE_NAME.py`**. So, we call the Python3 package. In crontab, we also need to call this package, but we need to call it from its directory. We need to write **`which Python3`** to find its

entire path. It's in the **/usr/bin** directory, and the full path of the package is **/usr/bin/Python3**, as shown in the following output.

```
root@Server-1:/home/ubuntu/Desktop# which Python3
/usr/bin/Python3
```

4. We are ready to set up the **crontab** service. If we log this service in to that Linux machine for the first time, it asks which editor package to open with. The most common and simple text editor package is nano. It also recommends the **nano** package. We need to enter the **1** value in the following output.

```
root@Server-1:/home/ubuntu/Desktop# crontab -e
no crontab for root - using an empty one
Select an editor. To change later, run 'select-editor'.
1. /bin/nano        <---- easiest
2. /usr/bin/vim.basic
3. /usr/bin/vim.tiny
4. /bin/ed
Choose 1-4 [1]: 1
```

5. After entering the number, the crontab service page is opened in the terminal with the nano text editor. Lines that start with the **#** dash character are the comment line in Linux, like in the Python language. So, it's an introduction to the service. We need to add the tasks in this file. Firstly, we need to set the timing to schedule the task.

   There are five parts to the orderly timing: minute, hour, day of the month, month, and day of the week. They are shown with ***** characters, and **/** means **every** in the timing. Consider the following example:

```
* * * * * - Every Minute
*/3 * * * * - Every 3 Minutes
* */5 * * * - Every 5 Hours
0 15 * * * - At 15:00 everyday
* * * * 1 - At every minute on Monday
```

   You can check the details of the timing on the internet. You can easily calculate or check the output of your schedule from the following link to understand the timing in the **crontab**.

   https://crontab.guru/

6. After the timing, we must set the task details in the same line. We want to collect the CPU levels of each device every minute, so the

timing is `* * * * *`. Then, we execute the Python file with the **python3** package in the **Desktop**" directory. So, we enter that directory by writing the full path with the **cd** command. Then, we add the **&&** double ampersand character, call the package from the **/usr/bin/Python3** path, and write the Python file in the following line. After we fill the **crontab** file, we save and exit it. The crontab service automatically starts the task after saving and exiting the file. The task starts at the beginning of the minute.

```
* * * * * cd /home/ubuntu/Desktop && /usr/bin/Python3
cpu_levels_periodically.py
```

7. We can add the following example code to the **cpu_levels_periodically.py** file we created in desktop, we enter the file with **nano cpu_levels_periodically.py** and paste them. In *Example 9.6*, we collect 5 seconds of CPU value from each device and append it to a file with their IP addresses. If the CPU level is higher than 70 percent, we create another file called **high_cpu_risk_devices.txt** and append the CPU levels to this text file.

```
root@Server-1:/home/ubuntu/Desktop# nano
cpu_levels_periodically.py
```

Example 9.6: Check CPU levels periodically with crontab

```
from netmiko import Netmiko
from re import findall
from datetime import datetime
from concurrent.futures import ThreadPoolExecutor
host = ["10.10.10.1", "10.10.10.2", "10.10.10.3"]
def collect_cpu(ip):
  device = {"host": ip, "username": "admin", "password":
  "cisco", "device_type": "cisco_ios"}
  command = "show processes cpu"
  print(f"\n---Try to Login:{ip}---\n")
  net_connect = Netmiko(**device)
  output = net_connect.send_command(command)
  cpu_5s = findall("five seconds: (\d+)",output)
  time = datetime.now().strftime("%d.%m.%Y %H:%M:%S")
  if int(cpu_5s[0]) > 90:
```

```
    cpu_risk = "Fatal CPU Level"
    with open(f"high_cpu_risk_devices.txt", "a") as w:
     w.write(f"---Time:{time}--- \nIP: {ip} \nCPU:{cpu_5s[0]}
     \nStatus:{cpu_risk}\n\n")
   elif 70 < int(cpu_5s[0]) < 90:
    cpu_risk = "High CPU Level"
    with open(f"high_cpu_risk_devices.txt", "a") as w:
     w.write(f"---Time:{time}--- \nIP: {ip} \nCPU:{cpu_5s[0]}
     \nStatus:{cpu_risk}\n\n")
   else:
    cpu_risk = "No Risk"
   with open (f"{ip}_cpu_levels.txt","a") as w:
    w.write(f"---Time:{time}--- \nIP: {ip} \nCPU:{cpu_5s[0]}
    \nStatus:{cpu_risk}\n\n")
with ThreadPoolExecutor(max_workers=50) as executor:
  result = executor.map(collect_cpu, host)
```

When we check the newly created files in the same directory with our code, it collects CPU levels every minute.

```
root@Server-1:/home/ubuntu/Desktop# cat
10.10.10.1_cpu_levels.txt
---Time:13.09.2022 15:08:04---
IP: 10.10.10.1
CPU:15
Status:No Risk
---Time:13.09.2022 15:09:03---
IP: 10.10.10.1
CPU:11
Status:No Risk
---Time:13.09.2022 15:10:03---
IP: 10.10.10.1
CPU:8
Status:No Risk
---Time:13.09.2022 15:11:04---
IP: 10.10.10.1
CPU:10
Status:No Risk
```

We can monitor the text file in real time with the **`tail -f`** command. So, we don't need to run the **`cat`** command every minute. The text file is updated when new lines are added to it.

```
root@Server-1:/home/ubuntu/Desktop# tail -f
10.10.10.1_cpu_levels.txt
---Time:13.09.2022 15:12:03---
IP: 10.10.10.1
CPU:15
Status:No Risk
---Time:13.09.2022 15:13:03---
IP: 10.10.10.1
CPU:9
Status:No Risk
```

Check router configuration for insecure passwords

In *Example 9.7*, we collect the username and password data in the Cisco devices and check whether the password is simple text or secret. If it's not a secret password for any user, the code alerts that the password is insecure for a specific username.

We execute the **`show run`** command on all devices simultaneously. Then, we try to find the **`username`** word in all lines and add to a list. In the **`for`** loop, we check for the **word** secret in the output. If we have a **`secret`** word, it means that the password is secure. Otherwise, the password is not safe for that username.

Example 9.7: Check username and passwords in routers

```
from netmiko import Netmiko
from re import findall
from concurrent.futures import ThreadPoolExecutor
host = ["10.10.10.1", "10.10.10.2", "10.10.10.3"]
def collect_cpu(ip):
  device = {"host": ip, "username": "admin", "password":
  "cisco", "device_type": "cisco_ios"}
  command = "show run"
  net_connect = Netmiko(**device)
```

```
  output = net_connect.send_command(command)
  username = findall("username.*",output)
  for user in username:
   secret = findall("secret", user)
   username = findall("username (\S+) ",user)
   if secret:
    print(f"{ip}: '{username[0]}' has a secret password. It's
    SECURE")
   else:
    print(f"{ip}: '{username[0]}' has no secret password. It's
    INSECURE")
with ThreadPoolExecutor(max_workers=50) as executor:
  result = executor.map(collect_cpu, host)
```

If we check the username and password in three routers, we see the password in simple text in some examples as **cisco**. In the **test123** and **test12** usernames in **Router-1**, passwords are secret. There is also the word **secret** before the secret password.

```
Router-1#show run | include username
username admin privilege 15 password 0 cisco
username test privilege 15 password 0 cisco
username test123 secret 5 $1$jjsA$EyYryqyh2SkUijYoc0K7s.
username test12 privilege 15 secret 5
$1$09LY$V1R8vRxCOR2pC/q6eren6.
```

```
Router-2#show run | include  username
username admin privilege 15 password 0 cisco
```

```
Router-3#show run | include  username
username admin privilege 15 password 0 cisco
```

We check all the passwords in the routers. We have an IP address and username value for each password, and the last line shows us whether it is secure.

Output:

```
10.10.10.2: 'admin' has no secret password. It's INSECURE
10.10.10.3: 'admin' has no secret password. It's INSECURE
10.10.10.1: 'admin' has no secret password. It's INSECURE
10.10.10.1: 'test' has no secret password. It's INSECURE
10.10.10.1: 'test123' has a secret password. It's SECURE
```

```
10.10.10.1: 'test12' has a secret password. It's SECURE
```

Check port security configuration in routers

In *Example 9.8*, we check the unused and **no shutdown** ports. If a port is active and there is no peer on the remote side, it has a risk of vulnerability. Someone can make cabling on purpose or mistakenly, affecting all network traffic. So, it's essential to network port security that you shut down all unused ports. We can check the risky interfaces with the Python script. We collect the interface information from the **show ip interface brief** command:

```
Router-1#show ip interface brief
Interface            IP-Address   OK? Method Status
Protocol
GigabitEthernet0/0   10.10.10.1   YES
NVRAM  up                     up
GigabitEthernet0/1   unassigned   YES
unset  down                   down
GigabitEthernet0/2   unassigned   YES unset  administratively
down down
GigabitEthernet0/3   unassigned   YES unset  administratively
down down
```

If the interface status is **up up** or **administratively down**, there is no risk. Otherwise, we have a risk. So, we create our code to find both words in the items in the following code. If we cannot see both, we have a risk. That's why we write **not port_shutdown** and **not port_up** in the **if** condition. We display all risky interfaces with the device IP address and interface names.

Example 9.8: Check port shutdown status for security

```
from netmiko import Netmiko
from re import findall
from concurrent.futures import ThreadPoolExecutor
host = ["10.10.10.1", "10.10.10.2", "10.10.10.3"]
def collect_cpu(ip):
  device = {"host": ip, "username": "admin", "password":
  "cisco", "device_type": "cisco_ios"}
  command = "show ip interface brief"
```

```
  net_connect = Netmiko(**device)
  output = net_connect.send_command(command)
  interfaces = findall("GigabitEthernet.*",output)
  for port in interfaces:
   int_name = findall("(GigabitEthernet\d+/\d+)", port)
   port_shutdown = findall("administratively down", port)
   port_up = findall("up\s+up", port)
   if not port_shutdown and not port_up:
   print(f"{ip}: '{int_name[0]}' is 'no shutdown' There is a
   risk")
with ThreadPoolExecutor(max_workers=50) as executor:
  result = executor.map(collect_cpu, host)
```

In the output, one port is not shut down and is not connected to another device. So, the interfaces are empty and active. If someone can make cabling on these ports, it has a risk to our network. The script shows us all the risky interfaces in detail.

Output:

```
10.10.10.1: 'GigabitEthernet0/1' is 'no shutdown' There is a
risk
10.10.10.2: 'GigabitEthernet0/3' is 'no shutdown' There is a
risk
10.10.10.3: 'GigabitEthernet0/2' is 'no shutdown' There is a
risk
```

Collect packets from ports with Pyshark

We can collect packets with packet capture tools like wireshark, which is the most popular packet capture tool in networking. We have a third-party module called **pyshark**, which is a simple packet capture module. We must install the module by running the **pip install pyshark** command in the terminal. In *Example 9.9*, we collect packets from the local PC interface.

1. We import the **pyshark** module to use it.

   ```
   import pyshark
   ```

2. We execute the **LiveCapture** function from the **pyshark** module and assign it to a **capture** variable. Inside the parentheses, we should write

the local PC interface, **Wi-Fi** in the following example. So, we collect the packets from the Wi-Fi interface in our local PC.

```
capture = pyshark.LiveCapture(interface='Wi-Fi')
```

3. After that, we sniff the packet from the network with the **sniff_continuously** function. We use the **packet_count** as three, meaning it collects three packets from the specific interface. We add timeout as 1 second to wait.

```
packets = capture.sniff_continuously(packet_count=3)
```

4. Finally, we call the **sniff_continuously** function with the **packet_count** parameter as 3. It means that three packets should be collected from the specific interface. We use it in a **for** loop.

```
for pckt in packets:
 print (f"\n\nPacket: \n{pckt}")
```

Example 9.9: Capture packets in an interface by pyshark

```
import pyshark
capture = pyshark.LiveCapture(interface='Wi-Fi')
packets = capture.sniff_continuously(packet_count=3)
for pckt in packets:
  print (f"\n\nPacket: \n{pckt}")
```

When we execute the code, we can see detailed packet information for each packet with its layers. We can manipulate or catch a packet we need with the **findall** function in more advanced usage of the **pyshark** module.

Conclusion

This chapter taught us about securing our network and system devices by configuring and checking the Python script. We configured security services in Linux servers and security features like ACLs in network devices. We collected logs from routers to check for security risks on the configuration and to alert the engineer if necessary. We also manipulated and sniffed network packets in any interface or port and investigated issues at the packet level.

The next chapter will focus on creating a Python software tool by writing small pieces of scripts using classes and functions. We will automate our tasks with a simple and fast solution by a software tool. We will combine most script examples in the previous chapters in a single script, like an automation tool.

Multiple choice questions

1. Which of the following commands in Ubuntu activates the "firewalld" service on the boot?
    a. `systemctl enable firewalld`
    b. `systemctl start firewalld`
    c. `systemctl activate firewalld`
    d. `firewalld enable`
2. Which of the following services is used to run tasks periodically in Linux systems?
    a. NetworkManager
    b. Openvpn
    c. Task Scheduler
    d. Crontab
3. Which command changes the zone to the default zone in the firewalld??
    a. `--get-default-zone`
    b. `--get-active-zones`
    c. `--set-default-zone`
    d. `--get-zones`

Answers

1. a
2. d
3. c

Questions

1. Write a script to create a new interface with an IP address and bind it to a new zone that will be the default zone in the server.
2. Write a script to collect packets and save each packet in a different text file from the Wi-Fi interface with the **`pyshark`** module.

CHAPTER 10
Deploying Automation Software

This chapter will focus on creating a network automation tool based on command prompt selection, including various network automation script codes. We will combine all scripts inside different files and classes according to their purposes. We have a tool that can do a lot in the network devices or servers, such as collecting logs, configuring devices, and more.

Structure

In this chapter, we will cover the following topics:

- Introduction to InquirerPy module
- Automation tool design
- Create main tool script
- Create sub tasks scripts
    - Network device scripts
    - Server scripts
    - Other remaining scripts

Objectives

We will install the third-party `InquirerPy` Python module to create a command prompt output that can ask a user to select an item. It's a Python **Command-line Interface** (**CLI**) with a selection of items. With the simple usage of this module, we will create a CLI-based network automation tool. We can collect logs, configure devices, transfer files, install packages, calculate subnets, ping tests, plot data, and do much more with a single automation tool. It's a flexible tool whose usage we can expand according to our requirements. We will divide the scripts into small pieces of code as functions, and then we will combine the functions into the classes according to their purposes.

Introduction to InquirerPy module

In this chapter, we will create a single main script that combines other small scripts. We will deploy automation software or tools to make automation faster and use the third-party **`InquirerPy`** module to design this automation tool. First of all, we must install this module in the Pycharm terminal via **`pip install InquirerPy`**.

The **`InquirerPy`** module is a re-implementation of the **`PyInquirer`** project with more features and bug fixes. There are also the **`questionary`** or **`columbo`** modules as alternatives.

The **`InquirerPy`** module provides a selection of questions or items to ask the user, and action is taken based on the answers. It's a questionnaire module that asks users questions and gets responses from them.

There are two syntax uses of the InquirerPy module: *classic syntax* and *alternate syntax*. We can use both versions and alternate syntax in our scripts. You can check the details of both syntaxes from the following official website:

https://inquirerpy.readthedocs.io/en/latest/

We use the **`inquirer`** class from the **`InquirerPy`** module. When we use this module, we use the **`from InquirerPy import inquirer`** line in our scripts.

To use this module, we must also change the running settings in the Pycharm tool, as we did in *Chapter 8: Monitor and Manage Servers*, to use the **`getpass`** function. If we use an IDE tool like Pycharm, the code gets stuck and does not ask for the password by default when we execute the code. We need to change the setting in the Pycharm tool. In *Figure 10.1*, we enter the **`Run`** tab and the **`Edit Configurations`** section.

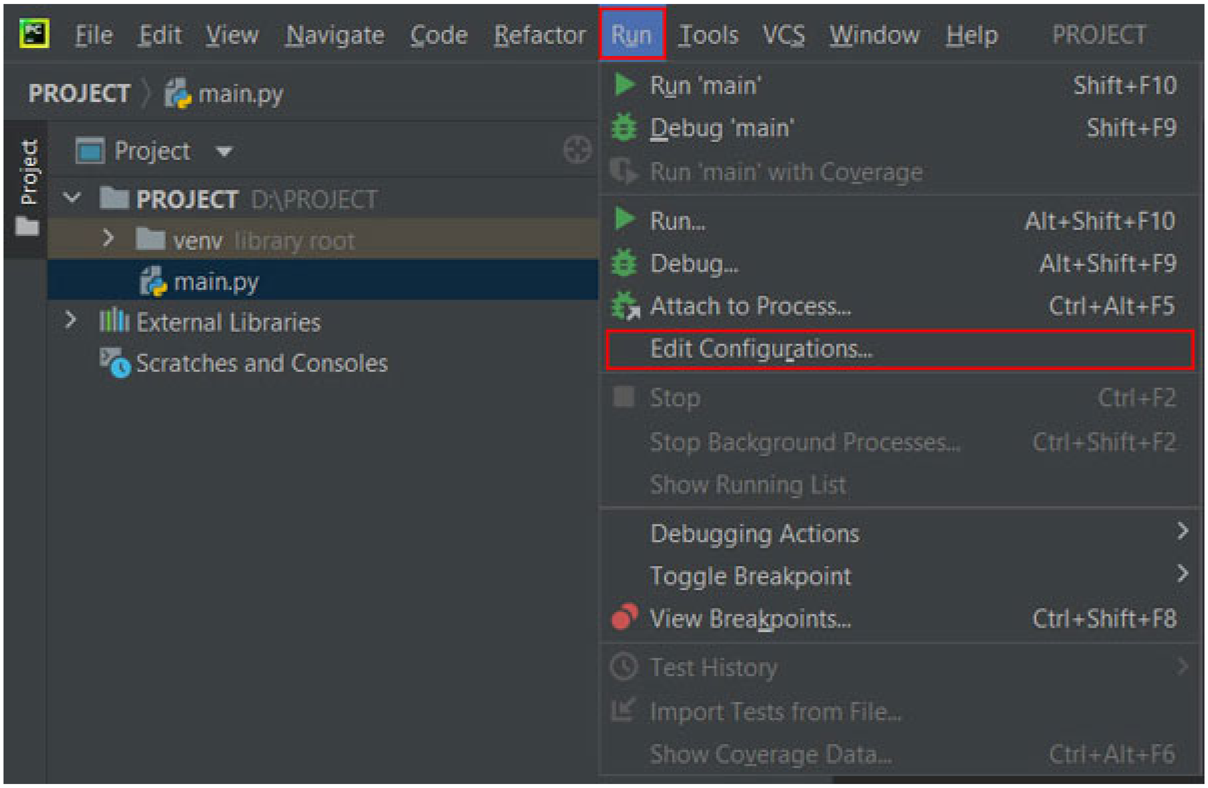


Figure 10.1: *Modifying Pycharm Configuration-1*

In *Figure 10.2*, we need to enable the **Emulate terminal in output console** feature, and close the window by clicking on the **`Apply`** button. After that, we can use **`InquirerPy`** without any problem.

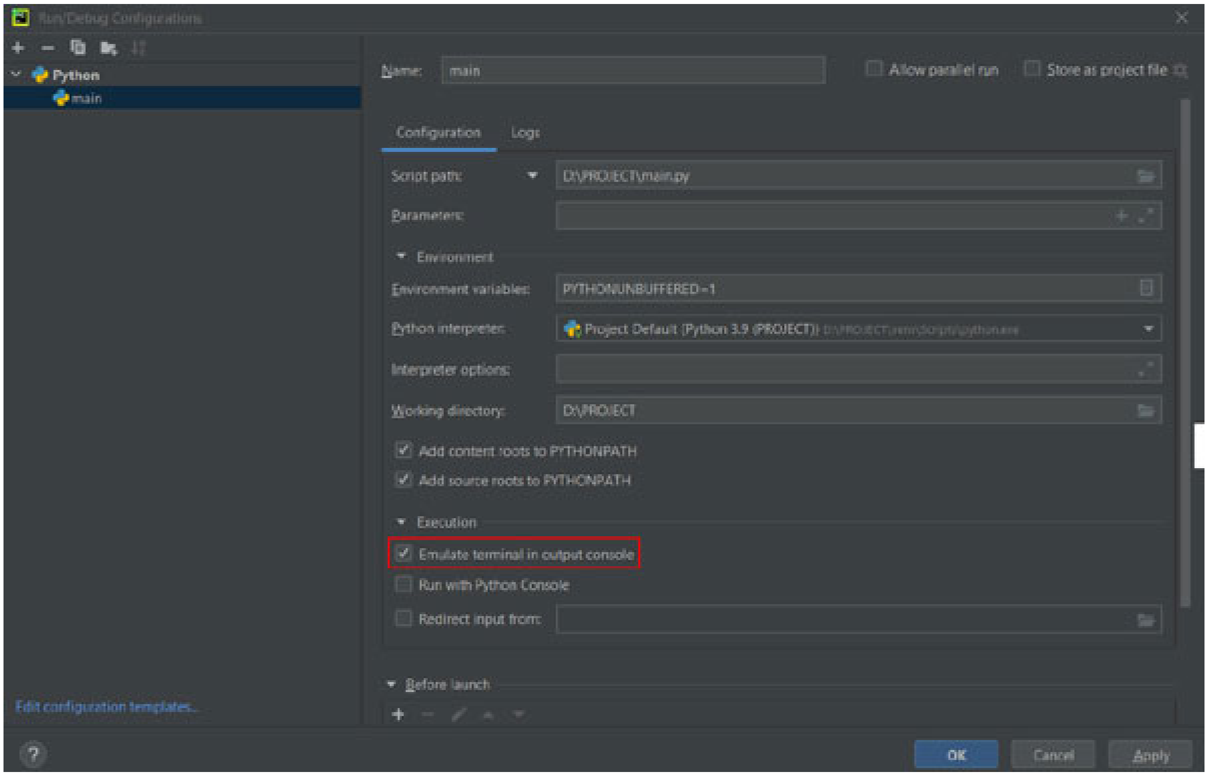


***Figure 10.2:** Modifying Pycharm Configuration-2*

The **`InquirerPy`** module has various prompts. We use **`text`**, **`select`**, **`confirm`**, and **`secret`** prompts in our script.

- **The "text" prompt**: As its name signifies, it's a text prompt that accepts user inputs and returns the output value. In the following code, we import the necessary function and then call the **`text`** class. We write the message to display in the output and run the **`execute`** function. We use the **`message`** parameter to display an introductory output to the user. We assign the value to a **`name`** variable. So, if we print the **`name`** variable, we can see the user input in the script's output. It's similar to Python's built-in **`input`** function, with advanced features.

```
from InquirerPy import inquirer
name = inquirer.text(message="What is your favorite
color:").execute()
print (f"Your favorite color is {name}")
```

When we execute the code in the following example, we enter **`yellow`** as user input. And code displays the **`yellow`** value from the name variable.

Output:

```
? What is your favorite color: yellow
Your favorite color is yellow
```

- **The "select" prompt:** It displays all items in a list to select from. It's a selector prompt that asks the user to select an item from the list in the output. We use it to ask the user which script to execute in our automation tool.

  We call the **`select`** class with the **`execute`** function, as we did in the **`text`** class. Inside the list, we have the **`message`** and **`choices`** parameters. Both are mandatory parameters in the select prompt. The **`message`** parameter displays an introductory message to the user. The **`choices`** parameter stores each item within a list to ask the user to choose one of them.

  In the following code, we ask for the favorite fruit and add four items to the **`choices`** parameter list. We assign all lines to a **`fruit`** variable to call it in the **`print`** function.

```
from InquirerPy import inquirer
fruit = inquirer.select(
 message="What's your favorite fruit:",
 choices=["Banana", "Apple", "Blueberry", "Orange"]
 ).execute()
print (f"Your favorite fruit is {fruit}")
```

  When we execute the script, the script output ask the user to choose any of the items in *Figure 10.3*. We can select any of the items or choices with the help of the arrow keys on the keyboard and then press the *Enter* button.

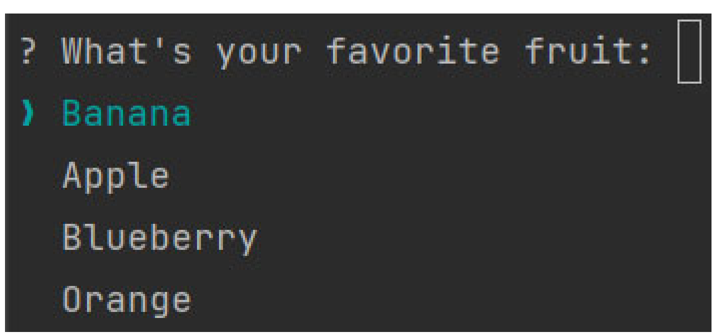


Figure 10.3: *Example of the "select" prompt*

If we select **Apple** for example, the code displays the selected items in the following output.

Output:

```
? What's your favorite fruit: Apple
Your favorite fruit is Apple
```

- **The "confirm" prompt:** It asks the user whether or not to confirm. If the user enters **y**, it gives the output as **True** in the boolean data type. Otherwise, it provides **False** as the output. We use this prompt when the user chooses the send configurations to a device. The changes affect the devices, so we must be more careful when we run the configuration change scripts. That's why we ask the user whether or not to continue with the **confirm** prompt.

  The usage of the confirm prompt is similar to that of te text and select prompts. When we execute the following script, if we enter **y**, the output of the **confirm** variable is **True**. Otherwise, it's **False**.

```
from InquirerPy import inquirer
confirm = inquirer.confirm(message="Confirm:").execute()
print (confirm)
```

Output:

```
? Confirm: Yes
True
```

Output:

```
? Confirm: No
False
```

- **The "secret" prompt:** It changes the user input to a secret character in the output. If the user writes **test** as an input, it displays ********. We use this prompt when we ask users to enter device passwords.

  The usage of this prompt is also similar. We add the message parameter to display in the output. We assign the value of this code to the **password** variable. If we print that variable, we can see our input in the output.

```
from InquirerPy import inquirer
password = inquirer.secret(message="Enter Device
Password:").execute()
print (password)
```

When executing the code, we enter **text** input as the following output.

Output:

```
? Enter Device Password: ****
test
```

Automation tool design

We created a network automation tool that provides 17 scripts for various purposes, such as collecting logs from network devices, installation of packages in servers, ping tests, and file transfers.

We design it with classes and functions. We divide functions into classes according to their purpose. We also divide classes into different Python files, such as files for network devices and files for system devices.

We can summarize the Python files, classes, and functions in the following list. We can divide it into smaller parts according to our structure so that the maintenance and readability are better. We have four Python files in our network automation tool.

In **main.py**, we call the classes and functions to execute the scripts. In this file, we use the **InquirerPy** module to write the visual part of our script with multiple choices of scripts. In each selection, we call a function from the other Python files.

In **network_devices.py**, we write all scripts related to the network devices, such as collecting logs, configuring devices, or file transfers. We have three classes in this file with various functions in the file.

In **servers.py**, we write all scripts related to the servers similar to **network_devices.py**. We have only one class, and all functions are inside this class; we can also divide them into different classes according to their purposes.

In **others.py**, we write the remaining scripts that we don't categorize with **network_devices.py** and **servers.py**. We have two classes: **tools** and **plotting**.

As a result, we have a network automation tool with 4 Python files, 6 classes, and 17 functions. We can add more scripts for more specific usage according to our requirements. Even with the main structure in the following steps, we have a powerful automation tool for network and system devices. This tool is written for Cisco on the network and Ubuntu on the system side, but we can add other vendors like Juniper, Huawei, or Nokia in networking and Windows in system.

Automation tool structure:

```
-main.py
-network_devices.py
  -class: collect_logs
  -def: from_one_device
  -def: from_multiple_devices
  -def: collect_device_info
  -def: collect_cpu_usage
  -def: send_logs_by_email
  -class: configure_device
  -def: config_with_netmiko
  -def: config_with_nornir
  -class: transfer_files
  -def: scp_upload_to_routers
  -def: sftp_upload_to_servers
-servers.py
  -class: config_and_collect_logs
  -def: config_collect_logs
  -def: collect_resource_usage
  -def: collect_interface_information
  -def: package_installation
-others.py
  -class: tools
  -def: subnet_calculator
  -def: ping_test
  -class: plotting
  -def: cpu_plot
  -def: interface_bandwidth_plot
```

In the following sections, we combine all of the codes in a single piece of code. At the end, we have an automation tool with many usage areas.

We also have text and YAML files in our tool to add device lists and configuration commands in a bundle. We create two folders: **input** and **output** folders. In the **input** directory, we have the **command_list.txt** file containing all the commands we send to devices. We also have the **device_list.txt** file that includes the device management IP addresses to log in to multiple devices.

In the **output** directory, we save all the output files if the script gives output as a file. For the **nornir** module, we use the **hosts.yaml** file, which contains device login information like the IP address, platform information, username, and password. This file is in the same directory as the main Python files. Otherwise, nornir cannot reach the YAML file.

According to this design, we have a directory structure in the following code:

Automation Tool Structure:

```
/
main.py
network_devices.py
servers.py
others.py
hosts.yaml
input/
  command_list.txt
  device_list.txt
output/
  Output Files
```

Create main tool script

We can start to write the **main.py** Python file to call all scripts in this file. We select the options or items with the **InquirerPy** module.

After that, we must import all the classes in various Python files mentioned in the earlier code. We can import each class one by one in the following example.

```
from InquirerPy import inquirer
from network_devices import collect_logs, configure_device,
transfer_files
from servers import config_and_collect_logs
from others import tools, plotting
```

We can also use the * character to import all classes or functions from a specific Python file or module. If we have many classes to import, this usage is better. In our automation tool, we use the following script:

```
from InquirerPy import inquirer
from network_devices import *
from servers import *
from others import *
```

We can write our code after importing all the necessary modules or Python files. We call some functions from the InquirerPy module, such as select, text, secret, and confirm functions.

We start by writing the first selection of our code. We use the **select** prompt with items like **Collect Logs**, **Device Configuration**, **File Transfer**, **Server Configuration**, **Others**, and **Exit**. The **Exit** item exits from the selection and finishes the code without action. We also add Exit item in each subtask.

```
main_task = inquirer.select(
  message="Choose a Main Task:",
  choices=["Collect Logs", "Device Configuration", "File
  Transfer", "Server Configuration", "Others",
  "Exit"]).execute()
```

After choosing an item in the output, we create an **if** condition to perform an action according to the item. We run the **main.py** file to start our automation tool. When we execute the **main.py** file, it gives an output, as shown in *Figure 10.4*. If the user selects **Collect Logs**, it continues with the items inside the **Collect Logs** item.

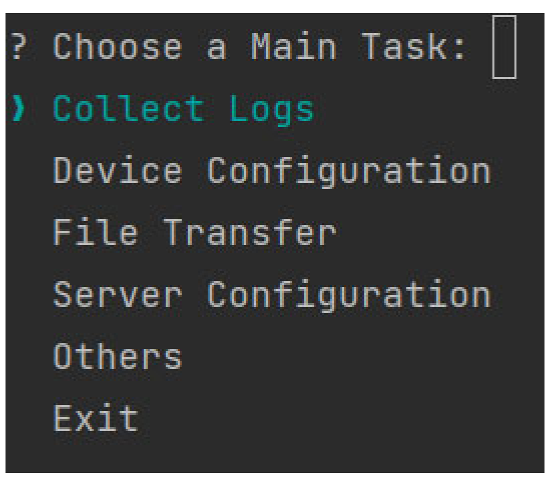


Figure 10.4: 1st Screen after executing the "main.py" script

Figure 10.5 shows the subtasks in the `Collect Logs` item. If we choose one of the items, such as `Collect CPU Usage`, the tool calls the `collect_cpu_usage` function from the `collect_logs` class, which is inside the `network_devices.py` Python file. It creates a file to write the output of the script.

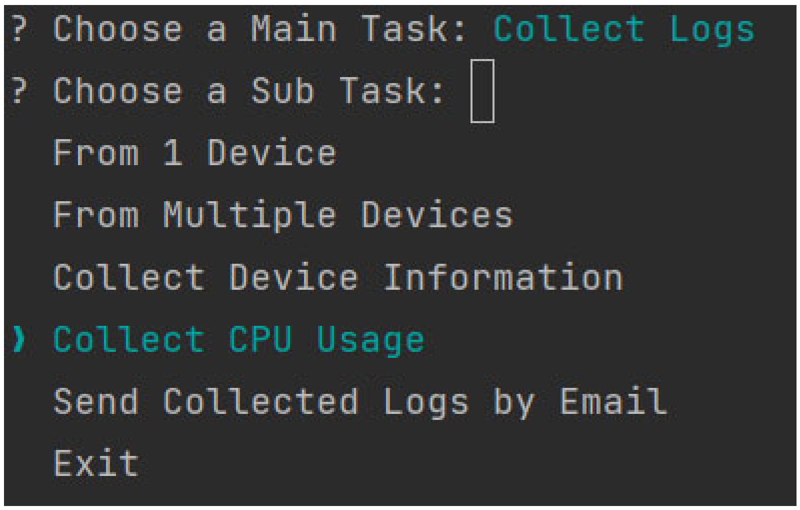


***Figure 10.5:** After selecting "Collect Logs" from the items*

We start with the first selection, which is `Collect Logs`. We assigned the select prompt value as the `main_task` in the previous piece of code. So, if the choice is `Collect Logs`, we continue to the subtask in the following code. There are six items, including the `Exit` item, and five of them execute a function from the related code.

We ask the user to select one of the six items with the `select` prompt again. If the user chooses one of them, the code starts executing the function. We use the `if` condition for each item again. For example, if we decide `From 1 Device`, it calls the function `from_one_device` from the `collect_logs` class that is inside the `network_devices.py` Python file.

The output displays three `text` prompts, i.e., `IP Address`, `Username`, and `Command`, which we try to execute on the device. It also displays one `secret` prompt to enter the `Password` value in hidden characters. At the end, it calls the `from_one_device` function with the four values that the user enters.

After that, we call each subtask inside in the **if** condition. Other scripts don't need any user entrance. Many get the device IP addresses from the

input/device_list.txt file for the netmiko connections. For the nornir scripts, the code receives the device data from the **hosts.yaml** file.

```
if main_task == "Collect Logs":
  sub_task = inquirer.select(
   message="Choose a Sub Task:",
   choices=["From 1 Device", "From Multiple Devices", "Collect
   Device Information", "Collect CPU Usage", "Send Collected
   Logs by Email", "Exit"]).execute()
  if sub_task == "From 1 Device":
  ip = inquirer.text(message="IP Address: ").execute()
  username = inquirer.text(message="Username: ").execute()
  password = inquirer.secret(message="Password: ").execute()
  command = inquirer.text(message="Command: ").execute()
  collect_logs.from_one_device(ip, username, password, command)
  elif sub_task == "From Multiple Devices":
  collect_logs.from_multiple_devices()
  elif sub_task == "Collect Device Information":
  collect_logs.collect_device_info()
  elif sub_task == "Collect CPU Usage":
  collect_logs.collect_cpu_usage()
  elif sub_task == "Send Collected Logs by Email":
  collect_logs.send_logs_by_email()
  elif sub_task == "Exit":
   print("Exited from the tool.")
```

We create the **if** conditions according to the structure in the earlier piece of code. If the first selection is the **Device Configuration**, we add two subtasks with choices again and write the **if** conditions with the related functions. For example, if we choose **Configure With Netmiko**, it displays the **confirm** prompt. It has one more step according to the previous examples. We ask the user to confirm the **device_list.txt** and **command_list.txt** files to control. If we enter **y**, the code starts the **config_with_netmiko** function. Otherwise, it finishes from the code. There is an additional step for important changes.

```
elif main_task == "Device Configuration":
  sub_task = inquirer.select(
   message="Choose a Sub Task:",
```

```
    choices=["Configure With Netmiko", "Configure With Nornir",
    "Exit"]).execute()
  if sub_task == "Configure With Netmiko":
    result = inquirer.confirm(message="\n**IP addresses in
    'input/device_list.txt'\n**Commands in
    'input/command_list.txt'\n").execute()
    if result:
      configure_device.config_with_netmiko()
    else:
      print("Exited from the tool.")
  elif sub_task == "Configure With Nornir":
    result = inquirer.confirm(message="\n**IP addresses in
    'hosts.yaml'\n**Commands in
    'input/command_list.txt'\n").execute()
    if result:
      configure_device.config_with_nornir()
    else:
      print("Exited from the tool.")
  elif sub_task == "Exit":
    print("Exited from the tool.")
```

If the first selection is the **File Transfer** item, it continues with two subtasks. In both tasks, the code also asks for the source and destination file names.

```
elif main_task == "File Transfer":
  sub_task = inquirer.select(
    message="Choose a Sub Task:",
    choices=["Upload with SCP to Routers", "Upload with SFTP to
    Servers", "Exit"]).execute()
  if sub_task == "Upload with SCP to Routers":
    src_file = inquirer.text(message="Source File on PC:
    ").execute()
    dest_file = inquirer.text(message="Destination File:
    ").execute()
    transfer_files.scp_upload_to_routers(src_file, dest_file)
  elif sub_task == "Upload with SFTP to Servers":
    src_file = inquirer.text(message="Source File on PC:
    ").execute()
```

```
    dest_file = inquirer.text(message="Destination File:
    ").execute()
    transfer_files.sftp_upload_to_servers(src_file, dest_file)
  elif sub_task == "Exit":
    print("Exited from the tool.")
```

If the first selection is **Server Configuration**, the code asks for four functions to execute. We have the `Configure or Collect Info`, `Collect Resource Usage`, `Collect Interface Information`, and `Install Packages` subtasks to execute with their functions. Specific functions are mentioned in each `if` condition.

```
elif main_task == "Server Configuration":
  sub_task = inquirer.select(
    message="Choose a Sub Task:",
    choices=["Configure or Collect Info", "Collect Resource
    Usage",
        "Collect Interface Information ", "Install Packages",
        "Exit"]).execute()
  if sub_task == "Configure or Collect Info":
    result = inquirer.confirm(message="\n**IP addresses in
    'input/device_list.txt'\n**Commands in
    'input/command_list.txt'\n").execute()
    if result:
      config_and_collect_logs.config_collect_logs()
    else:
      print("Exited from the tool.")
  elif sub_task == "Collect Resource Usage":
    config_and_collect_logs.collect_resource_usage()
  elif sub_task == "Collect Interface Information ":
    config_and_collect_logs.collect_interface_information()
  elif sub_task == "Install Packages":
    package_name = inquirer.text(message="Enter Package Name:
    ").execute()
    result = inquirer.confirm(message="\n**IP addresses in
    'input/device_list.txt'\n").execute()
    if result:
      config_and_collect_logs.package_installation(package_name)
    else:
```

```
    print("Exited from the tool.")
  elif sub_task == "Exit":
   print("Exited from the tool.")
```

As the last item, we have `"Others"` in the first selection. There are four functions to execute in this code.

```
elif main_task == "Others":
  sub_task = inquirer.select(
   message="Choose a Sub Task:",
   choices=["Subnet Calculator", "Ping Test", "Plotting CPU
   Levels", "Plotting Interface Bandwidth", "Exit"]).execute()
  if sub_task == "Subnet Calculator":
   ip_address = inquirer.text(message="Enter an IP address:
   ").execute()
   subnet_mask = inquirer.text(message="Enter a Subnet Mask (1
   to 32): ").execute()
   tools.subnet_calculator(ip_address, subnet_mask)
  elif sub_task == "Ping Test":
   ip_address = inquirer.text(message="Enter an IP address:
   ").execute()
   ping_count = inquirer.text(message="Enter Quantity of Ping
   Packets: ").execute()
   tools.ping_test(ip_address, ping_count)
  elif sub_task == "Plotting CPU Levels":
   ip_address = inquirer.text(message="Enter an IP address:
   ").execute()
   plotting.cpu_plot(ip_address)
  elif sub_task == "Plotting Interface Bandwidth":
   ip_address = inquirer.text(message="Enter an IP address:
   ").execute()
   interface_name = inquirer.text(message="Enter the Interface
   Name: ").execute()
   plotting.interface_bandwidth_plot(ip_address,
   interface_name)
  elif sub_task == "Exit":
   print("Exited from the tool.")
```

We can write the device IP address `input/device_list.txt` file, as we did in the previous examples.

```
device_list.txt:
10.10.10.1
10.10.10.2
10.10.10.3
```

We can write the commands we try to send to the devices in the `input/command_list.txt` file.

```
command_list.txt
show version
show ip interface brief
show interface description
```

We can also use the `hosts.yaml` file to get the device information for the `nornir` module connections.

```
hosts.yaml
Router-1:
   hostname: 10.10.10.1
   platform: ios
   username: admin
   password: cisco
Router-2:
   hostname: 10.10.10.2
   platform: ios
   username: admin
   password: cisco
Router-3:
   hostname: 10.10.10.3
   platform: ios
   username: admin
   password: cisco
```

Create subtask scripts

We can write the script files to execute in the devices such as, such as `network_devices.py`, `servers.py`, and `others.py`, with related classes and functions. We already write the `main.py` that is the main tool file we wrote in the earlier section. The codes in these three files have already been written in the previous chapters, and we can change small parts of them to combine with this structure. You can check related examples in the earlier chapters if you need to check the details of the following examples.

Network device scripts

At the beginning of the **network_devices.py** file, we import the following modules with the necessary functions:

```
from netmiko import Netmiko
from concurrent.futures import ThreadPoolExecutor
from re import findall, split
from pandas import DataFrame
import smtplib
from email import message
import mimetypes
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_config,
netmiko_file_transfer
from paramiko import SSHClient, AutoAddPolicy
```

In the **collect_logs** class, we call the following examples in the same file.

In the following code, we write the **from_one_device** function to log in to a single device, collect a single command log, and display it as output.

Example 10.1: Collect logs from a single device

```
class collect_logs:
  def from_one_device(ip, username, password,command):
   device = { "host": ip, "username": username, "password":
   password, "device_type": "cisco_ios"}
   net_connect = Netmiko(**device)
   show_output = net_connect.send_command(command)
   print(show_output)
```

In the following code, we collect multiple logs from many devices with the **from_multiple_devices** function by using the **netmiko** module. This function is in the **collect_logs** class.

Example 10.2: Collect logs from multiple devices

```
  def from_multiple_devices():
   with open("input/device_list.txt") as r:
```

```
  device_list = r.read().splitlines()
 with open("input/command_list.txt") as r:
  command_list = r.read().splitlines()
 def concurrent(ip):
    device = {"host": ip, "username": "admin", "password":
    "cisco", "device_type": "cisco_ios"}
   net_connect = Netmiko(**device)
   hostname = net_connect.find_prompt()
   for command in command_list:
    output = net_connect.send_command(command,
    strip_command=False)
    print(f"{hostname} {output}\n")
    with open(f"output/{ip} logs.txt", "a") as w:
    w.write(f"{hostname} {output}\n\n")
 with ThreadPoolExecutor(max_workers=25) as executor:
  executor.map(concurrent, device_list)
```

In the following code, we collect device information like IP address, hostname, vendor type, device model, and software version with the `collect_device_info` function. This function is in the `collect_logs` class.

Example 10.3: Collect device information

```
def collect_device_info():
 with open("input/device_list.txt") as r:
  device_list = r.read().splitlines()
 ip_list, version_list, model_list, vendor_list,
 hostname_list = ([] for i in range(5))
 for ip in device_list:
  device = {"host": ip, "username": "admin", "password":
  "cisco", "device_type": "cisco_ios"}
  print(f"\n---Try to Login:{ip}---\n")
  net_connect = Netmiko(**device)
  output = net_connect.send_command("show version")
  version = findall("Version (.*),", output)
  model = findall("Cisco (.*)\(revision", output)
  vendor = findall("Cisco", output)
  hostname = findall("(.*)#", net_connect.find_prompt())
  ip_list.append(ip)
  version_list.append(version[0])
```

```
        model_list.append(model[0])
        vendor_list.append(vendor[0])
        hostname_list.append(hostname[0])
    df = DataFrame(
        {"IP Address": ip_list, "Hostname": hostname_list, "Vendor
        Type": vendor_list, "Model": model_list, "Version":
        version_list})
    df.to_excel("output/Version List.xlsx",
    sheet_name="Vendors", index=False)
```

In the following example, we collect CPU usage information from various devices and save it in an Excel file with the `collect_cpu_usage` function. This function is in the `collect_logs` class.

Example 10.4: Collect CPU usage information

```
def collect_cpu_usage():
    with open("input/device_list.txt") as r:
        device_list = r.read().splitlines()
    ip_list, cpu_list_5s, cpu_list_1m, cpu_list_5m,
    cpu_list_risk = ([] for x in range(5))
    for ip in device_list:
        device = {"host": ip, "username": "admin", "password":
        "cisco", "device_type": "cisco_ios"}
        print(f"\n---Try to Login:{ip}---")
        net_connect = Netmiko(**device)
        output = net_connect.send_command("show processes cpu")
        cpu_5s = findall("CPU utilization for five seconds: (\d+)",
        output)
        cpu_1m = findall("one minute: (\d+)", output)
        cpu_5m = findall("five minutes: (\d+)", output)
        ip_list.append(ip)
        cpu_list_5s.append(cpu_5s[0] + "%")
        cpu_list_1m.append(cpu_1m[0] + "%")
        cpu_list_5m.append(cpu_5m[0] + "%")
        if int(cpu_5m[0]) > 90:
            cpu_risk = "Fatal CPU Level"
        elif 70 < int(cpu_5m[0]) < 90:
            cpu_risk = "High CPU Level"
        else:
```

```
      cpu_risk = "No Risk"
    cpu_list_risk.append(cpu_risk)
  df = DataFrame(
    {"IP Address": ip_list, "CPU Levels for 5 Seconds":
    cpu_list_5s, "CPU Levels for 1 Minute": cpu_list_1m, "CPU
    Levels for 5 Minutes": cpu_list_5m, "CPU Risk":
    cpu_list_risk})
  df.to_excel("output/CPU Levels.xlsx", index=False)
```

In the following code, we collect logs, save them as output file and send an email to ourselves with the `send_logs_by_email` function. This function is in the `collect_logs` class.

Example 10.5: Send collected logs via email

```
def send_logs_by_email():
  with open("input/device_list.txt") as r:
    device_list = r.read().splitlines()
  with open("input/command_list.txt") as r:
    command_list = r.read().splitlines()
  def concurrent(ip):
    print(f"---Try to Login:{ip}---")
    device = {"host": ip, "username": "admin", "password":
    "cisco", "device_type": "cisco_ios"}
    net_connect = Netmiko(**device)
    hostname = net_connect.find_prompt()
    for command in command_list:
        output = net_connect.send_command(command,
        strip_command=False)
      with open(f"output/{ip} logs.txt", "a") as w:
        w.write(f"{hostname} {output}\n\n")
  with ThreadPoolExecutor(max_workers=25) as executor:
    executor.map(concurrent, device_list)
  print("\nSending Email")
  mail_from = "example@gmail.com"
  mail_password = "16-DIGIT-PASSWORD"
  mail_to = "example@gmail.com"
  mail_subject = "Device Logs"
  mail_content = "Hi,\nYou can find the all device log files
  in the attachment."
```

```
  send = message.EmailMessage()
  send.add_header("From", mail_from)
  send.add_header("To", mail_to)
  send.add_header("Subject", mail_subject)
  send.set_content(mail_content)
  for file in device_list:
   filename = f"output/{file} logs.txt"
   with open(filename, "rb") as r:
    attached_file = r.read()
   mime_type, encoding = mimetypes.guess_type(filename)
   send.add_attachment(attached_file,
   maintype=mime_type.split("/")[0],
   subtype=mime_type.split("/")[1], filename=filename)
  with smtplib.SMTP_SSL("smtp.gmail.com", 465) as smtp:
   smtp.login(mail_from, mail_password)
   smtp.sendmail(mail_from, mail_to, send.as_string())
```

We continue with a new class called **configure_device** in the same Python file. In the following code, we configure the network devices using the **netmiko** module with the **config_with_netmiko** function. This function is in the **configure_device** class.

Example 10.6: Configure network devices with netmiko

```
class configure_device:
 def config_with_netmiko():
  with open("input/device_list.txt") as r:
   device_list = r.read().splitlines()
  def concurrent(ip):
   device = {"host": ip, "username": "admin", "password":
   "cisco", "device_type": "cisco_ios"}
   net_connect = Netmiko(**device)
   output = net_connect.send_config_from_file
   (config_file="input/command_list.txt", strip_command=False)
   print(output)
  with ThreadPoolExecutor(max_workers=25) as executor:
   executor.map(concurrent, device_list)
```

In the following example, we configure the network devices using the **nornir** module with the **config_with_nornir** function. This function is in the **configure_device** class.

Example 10.7: Configure network devices with nornir

```
def config_with_nornir():
  with open("input/command_list.txt") as r:
    command_list = r.read().splitlines()
  connect = InitNornir(config_file="hosts.yaml")
  result = connect.run(task=netmiko_send_config,
  config_commands=command_list)
  print_result(result)
```

We continue with a new class called **transfer_files** in the same Python file. In the following example, we upload files from the local PC to the network devices with the **scp_upload_to_routers** function. This function is in the **transfer_files** class.

Example 10.8: Upload files to the network devices

```
class transfer_files:
  def scp_upload_to_routers(src_file, dest_file):
    if not dest_file:
      dest_file = src_file
    connect = InitNornir(config_file="hosts.yaml")
    result = connect.run(task=netmiko_file_transfer,
    source_file=src_file, dest_file=dest_file, direction="put")
    print_result(result)
```

In the following example, we upload files from the local PC to the servers with the **sftp_upload_to_servers** function. This function is in the **transfer_files** class.

Example 10.9: Upload files to the servers

```
  def sftp_upload_to_servers(src_file, dest_file):
    with open("input/device_list.txt") as r:
      device_list = r.read().splitlines()
    ssh = SSHClient()
    ssh.set_missing_host_key_policy(AutoAddPolicy())
    for ip in device_list:
      ssh.connect(hostname=ip, username="ubuntu",
      password="ubuntu")
      sftp = ssh.open_sftp()
      sftp.put(src_file,dest_file)
```

Server scripts

We continue to fill the **servers.py** Python file to connect servers and configure or collect logs from them. In this file, we have a single class called **config_and_collect_logs**, which has four functions. We import the following modules and their necessary functions according to the functions we use in this file:

```
from netmiko import Netmiko
from concurrent.futures import ThreadPoolExecutor
from re import findall
from pandas import DataFrame
```

We write our single class called **config_and_collect_logs** in the **servers.py** Python file. In the following example, we configure servers or collect logs from them with the **config_collect_logs** function. This function is in the **config_and_collect_logs** class.

Example 10.10: Configure servers or collect logs from servers

```
class config_and_collect_logs:
  def config_collect_logs():
    with open("input/device_list.txt") as r:
      device_list = r.read().splitlines()
    def concurrent(ip):
      device = {"host": ip, "username": "ubuntu", "password":
      "ubuntu", "device_type": "linux", "secret": "ubuntu"}
      net_connect = Netmiko(**device)
      hostname = net_connect.find_prompt()
      output =
      net_connect.send_config_from_file(config_file="input/comman
      d_list.txt", strip_command=False)
      print(f"{hostname} {output}\n")
    with ThreadPoolExecutor(max_workers=25) as executor:
      executor.map(concurrent, device_list)
```

In the following example, we collect CPU and memory resources usage of servers with the **collect_resource_usage** function. This function is in the **config_and_collect_logs** class.

Example 10.11: Collect resource usage information of servers

```
  def collect_resource_usage():
```

```
memory_total, memory_free, memory_used, cpu_used, host_list
= ([] for i in range(5))
with open("input/device_list.txt") as r:
 device_list = r.read().splitlines()
for ip in device_list:
 device = {"host": ip, "username": "ubuntu", "password":
 "ubuntu", "device_type": "linux", "secret": "ubuntu"}
 net_connect = Netmiko(**device)
 mem_output = net_connect.send_command("free -m",
 strip_command=False)
 cpu_output = net_connect.send_command("top -n 1 | grep
 %Cpu", strip_command=False)
 hostname = findall("@(.*):", net_connect.find_prompt())
 total = findall("Mem:\s+(\d+)", mem_output)
 free = findall("Mem:\s+\d+\s+(\d+)", mem_output)
 used = findall("Mem:\s+\d+\s+\d+\s+(\d+)", mem_output)
 cpu = findall("\d+,\d+", cpu_output)
 memory_total.append(f"{total[0]} MB")
 memory_free.append(f"{free[0]} MB")
 memory_used.append(f"{used[0]} MB")
 cpu_used.append(f"% {cpu[0]}")
 host_list.append(hostname[0])
df = DataFrame({"Hostname": host_list, "Total Memory":
memory_total, "Free Memory": memory_free, "Memory Usage":
memory_used, "CPU Usage": cpu_used})
df.to_excel("output/CPU-Memory Usage.xlsx", index=False)
```

In the following example, we collect the interface information of the servers, such as hostname, interface name, IP address, and netmask, with the **collect_interface_information** function. This function is in the **config_and_collect_logs** class.

Example 10.12: Collect Interface Information of Servers

```
def collect_interface_information():
 list_ipv4, list_netmask, list_int, list_hostname,
 list_int_name = ([] for i in range(5))
 with open("input/device_list.txt") as r:
  device_list = r.read().splitlines()
 for ip in device_list:
```

```
    device = {"host": ip, "username": "ubuntu", "password":
    "ubuntu", "device_type": "linux", "secret": "ubuntu"}
    net_connect = Netmiko(**device)
    output = net_connect.send_command("ifconfig")
    hostname = findall("@(.*):", net_connect.find_prompt())
    int_name = findall("(.*): flags", output)
    for interface in int_name:
      output = net_connect.send_command(f"ifconfig -a
      {interface}")
     ipv4 = findall("inet (.*)  netmask", output)
     netmask = findall("netmask (\d+.\d+.\d+.\d+)", output)
     list_ipv4.append(ipv4[0])
     list_netmask.append(netmask[0])
     list_hostname.append(hostname[0])
     list_int_name.append(interface)
  df = DataFrame({"Hostname": list_hostname, "Interface Name":
  list_int_name, "IP Address": list_ipv4, "Netmask":
  list_netmask, })
  df.to_excel("output/Interface Information.xlsx",
  index=False)
```

In the following example, we install packages on the servers with the `package_installation` function. This function is in the `config_and_collect_logs` class.

Example 10.13: Install package on the servers

```
def package_installation(package):
  with open("input/device_list.txt") as r:
    device_list = r.read().splitlines()
  def concurrent(ip):
    device = {"host": ip, "username": "ubuntu", "password":
    "ubuntu", "device_type": "linux", "secret": "ubuntu"}
    net_connect = Netmiko(**device)
    net_connect.send_config_set(f"sudo apt-get install
    {package} -y")
    output = net_connect.send_command(f"{package} --version")
    hostname = net_connect.find_prompt()
    print(f"{hostname}: {package} --version{output}\n")
  with ThreadPoolExecutor(max_workers=25) as executor:
```

```
    executor.map(concurrent, device_list)
```

Other remaining scripts

We continue to fill the **others.py** Python file to execute the remaining scripts in the **others** category. This file has two classes: **tools** and **plotting**. In each class, we have two functions. We import the following modules and their necessary functions according to the functions we use in **others.py** file:

```
from re import findall
from subprocess import Popen, PIPE
from matplotlib import pyplot as plt
from netmiko import Netmiko
from time import sleep
from datetime import datetime
```

We write the first class named **tools** in the **servers.py** Python file. Inside it, we write the **subnet_calculator** function, similar to *Example 4.14* in *Chapter 4, Collecting and Monitoring Logs*. We *remove* the following lines:

```
enter_ip = input("\nEnter an IP address: ")
  mask = input("\nEnter a Subnet Mask (1 to 32):  address: ")
```

We replace the variable names from **enter_ip** to **ip_address** and from **mask** to **subnet_mask** in the following example. The remaining part is the same as *Example 4.14*. We write the class named **tools**. Our function name for subnet calculator is **subnet_calculator**, with two parameters: **ip_address** and **subnet_mask**.

```
class tools:
  def subnet_calculator(ip_address,subnet_mask)
```

In the following example, we start the ping test from our local PC by choosing the ping packet quantity with the **ping_test** function. This function is in the **tools** class.

Example 10.14: Ping Test from the Local Device

```
  def ping_test(ip,ping_count):
    output = ""
    print(f"\n---Try to Ping: {ip} ---")
    data = Popen(f"cmd /c ping {ip} -n {ping_count}",
    stdout=PIPE, encoding="utf-8")
```

```
    for line in data.stdout:
     output = output + "\n" + line.rstrip('\n')
    print(output)
```

We write our second class named **plotting** in the **others.py** Python file. In the following example, we collect the CPU data in a period and plot it with the **cpu_plot** function. This function is in the **plotting** class.

Example 10.15: Plot CPU levels of a network device

```
class plotting:
  def cpu_plot(ip):
    host = {"host": ip, "username": "admin", "password":
    "cisco", "device_type": "cisco_ios"}
    count = 7
    delay = 3
    command = "show processes cpu"
    cpu_levels = []
    time_list = []
    net_connect = Netmiko(**host)
    for i in range(1, count):
     print(f"Get CPU levels count: {i}")
     output = net_connect.send_command(command)
     time = datetime.now().strftime("%H:%M:%S")
     time_list.append(time)
     sleep(delay)
     cpu_data = findall("CPU utilization for five seconds:
     (\d+)%/", output)
     cpu_levels.append(int(cpu_data[0]))
     print("CPU Level: ", cpu_data[0])
    plt.plot(time_list, cpu_levels)
    plt.xlabel("Time")
    plt.ylabel("CPU Levels in %")
    plt.grid(True)
    plt.show()
```

In the following example, we collect inbound and outbound interface traffic data and plot it with the **interface_bandwidth_plot** function. This function is in the **plotting** class.

Example 10.16: Collect logs from a single device

```
def interface_bandwidth_plot(ip, interface):
 host = {"host": ip, "username": "admin", "password":
 "cisco", "device_type": "cisco_ios"}
 count = 5
 delay = 3
 inbound_rate = []
 outbound_rate = []
 time_list = []
 net_connect = Netmiko(**host)
 for i in range(1, count):
  output = net_connect.send_command(f"show interfaces
  {interface}")
  time = datetime.now().strftime("%H:%M:%S")
  time_list.append(time)
  input_level = findall("5 minute input rate (\d+)", output)
  output_level = findall("5 minute output rate (\d+)",
  output)
  inbound_rate.append(int(input_level[0]))
  outbound_rate.append(int(output_level[0]))
  sleep(delay)
  print("Input Level: ", input_level[0])
  print("Output Level: ", output_level[0])
 plt.plot(time_list, inbound_rate, color="blue",
 label="Inbound")
 plt.plot(time_list, outbound_rate, color="red", label=
 "Outbound")
 plt.xlabel("Time")
 plt.ylabel("Interface Levels in MBs")
 plt.title(f"Interface Rate of {host['host']} - {interface}")
 plt.show()
```

Conclusion

This chapter discussed creating a custom CLI-based automation tool. We planned a tool design by dividing the scripts into files and classes according to their similarities or usage purposes. We connected the network devices and servers with the netmiko, nornir, and paramiko modules. We collected data, ran configurations, and transferred files to them.

The next chapter will focus on **Amazon Web Services** (**AWS**) cloud infrastructures. We write automation scripts to create and manage AWS services like servers and storage by Python coding.

Multiple choice questions

1. Which of the following prompts changes the input characters into hidden characters?
    a. password
    b. secret
    c. hidden
    d. encrypted
2. Which of the following modules is not an alternative to the "PyInquirer" module?
    a. columbo
    b. InquirerPy
    c. selectors
    d. questionary

Answers

1. b
2. c

Questions

1. Write main.py file by adding coloring from the `InquirerPy` function.
2. Add a new subtask called `Save Device Configuration into a File` to the `collect_logs` class.

CHAPTER 11

Automate Cloud Infrastructures with Python

This chapter will focus on network automation in a cloud platform, such as Amazon's AWS Cloud Platform. We will deploy a cloud environment in the AWS platform to execute the Python scripts. We will manage various services like **EC2** for instance, **S3** and **EBS** for storage, and **IAM** for user account management. Additionally, we will use the `boto3` module to handle all tasks without entering the AWS Console web page.

Structure

In this chapter, we will cover the following topics:

- Cloud environment deployment
  - Introduction to AWS
  - Installation of Boto3 and AWS CLI
- EC2 ınstance management
  - Manage EC2 ınstances with Python
  - Connection to EC2 ınstances
- S3 bucket management
- EBS volume management
  - Manage EBS volumes
  - Create snapshots of EBS volumes
  - Attach EBS volume to EC2 ınstance
- IAM user management

Objectives

We will be introducing Amazon's AWS cloud platform with essential services like EC2, S3, EBS, and IAM. We will install the necessary Python module called `boto3` to write the automation scripts. We'll also check AWS Console and AWS CLI usage to verify that our scripts work successfully. We will create Linux servers as EC2 instances and manage them with the scripts. Paramiko and netmiko modules will be brought into use to log in to these instances, and we will manage S3 and EBS storage with simple scripts, create a backup of EBS volumes via snapshots, and attach or bind EBS volumes to a particular EC2 instance. Ultimately, we will manage user access and permissions with Python codes. Lastly, we will create users and add or delete roles from those users.

Cloud environment deployment

In the previous chapter, we created **Amazon Web Services** (**AWS**) Cloud technology automation scripts. Three significant brands lead in Cloud computing services: Amazon's AWS, Google's **Google Cloud Platform** (**GCP**), and Microsoft's Azure. We will focus on AWS, the pioneer in cloud services; AWS is one of the most popular services for system, cloud, and DevOps engineers.

Introduction to AWS

AWS is a web-based platform that includes various services. Each service has its purpose, and we will focus on EC2, S3, EBS, and IAM, the most used services in AWS.

- **EC2** (**Elastic Compute Cloud**) is an instance-based server system in the AWS platform. We can create Linux or Windows servers with EC2 service.
- **S3** (**Simple Storage Service**) is a bucket-based storage system on the AWS platform. We can create hosting or straightforward cloud storage to keep our files in the cloud.
- **EBS** (Elastic Block Store) is a volume-based storage system on the AWS platform. We can also keep files in the cloud as an S3 service, but with more enhanced features like attaching volumes to EC2 instances or snapshot EBS volumes that back up the volume in a different system.
- **IAM** (**Identity and Access Management**) is a user management system on the AWS platform. We can create and delete users and change user

privileges in this service.

We can handle all these services in the AWS platform from the following website. Instead of this, we log in to this platform with a Python module, that is, **`boto3`**, which is a third-party module and completes all tasks through custom Python scripts. So, we can automate the AWS platform with simple scripts. For example, we can create 10 EC2 instances with a single code or add a new service privilege to all users in our AWS account. So, the **`boto3`** module is a straightforward and powerful module to automate AWS services.

Installation of Boto3 and AWS CLI

- **Create AWS account**: Before we start the installation of modules and necessary tools, we connect to the AWS platform with our account. You can continue installing the **`boto3`** module if you already have an account. Otherwise, you need to create an AWS account with your email address from the following link:

  https://aws.amazon.com/

  Creating an account is free in AWS and other cloud services, but running services are paid, such as EC2, S3, or EBS. So, it costs us if we create and start an EC2 instance. It's hourly usage in USD, and the cost for this test is less than a dollar. However, remember to shut down or terminate services like an EC2 instance or S3 buckets after you finish the test. Otherwise, AWS charges for running objects. You can check the billing service after you start some services.
- **Install boto3 module**: We can continue installing the **`boto3`** module, a third-party Python module provided by AWS. We must run the **`pip install boto3`** command in the IDE terminal. You can check the details of the **`boto3`** module from the official web page at the following link:

  https://boto3.abmazonaws.com/v1/documentation/api/latest/index.html
- **Create user in IAM**: After that, we need to create a user inside our AWS account. After we log in to AWS, we need to open the IAM service. You can search IAM or check the following link. In the **`Users`** section, we can click on the **`Add users`** button to create a new user.

  https://us-east-1.console.aws.amazon.com/iamv2/home#/home

In *Figure 11.1*, we enter a username, i.e., **test_user**, and choose access type as **Access key - Programmatic access**; then, we continue to the next step.

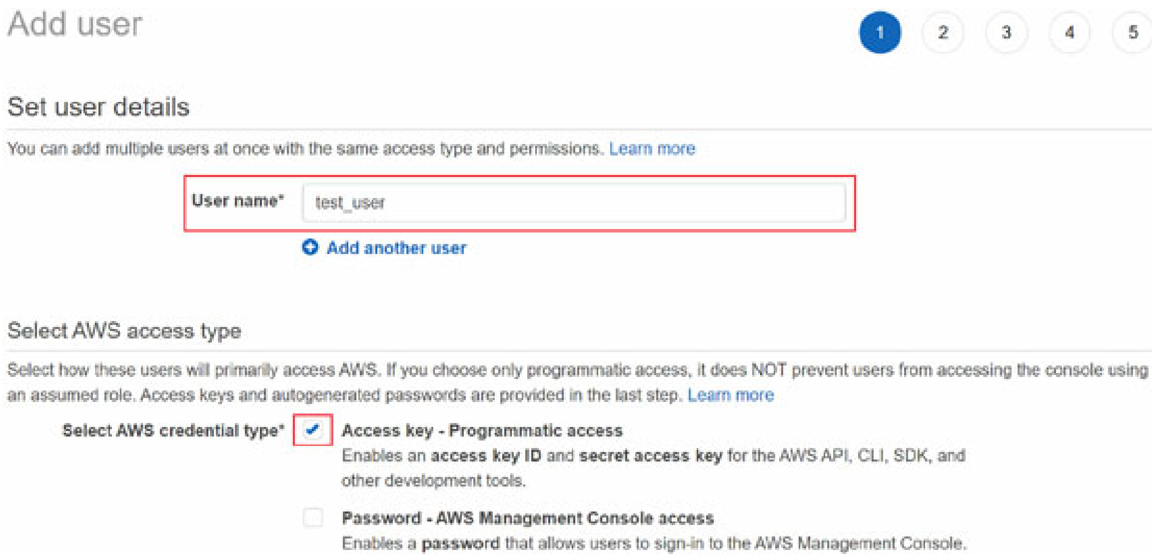


Figure 11.1: *Adding User-1*

In *Figure 11.2*, in the next step, we choose the `Attach existing policies directly` tab and add policies to the new user. We can search for and enable the following access permissions for full access to EC2, S3, and IAM services.

```
Policy for Permissions:
AmazonEC2FullAccess
AmazonS3FullAccess
IAMFullAccess
```

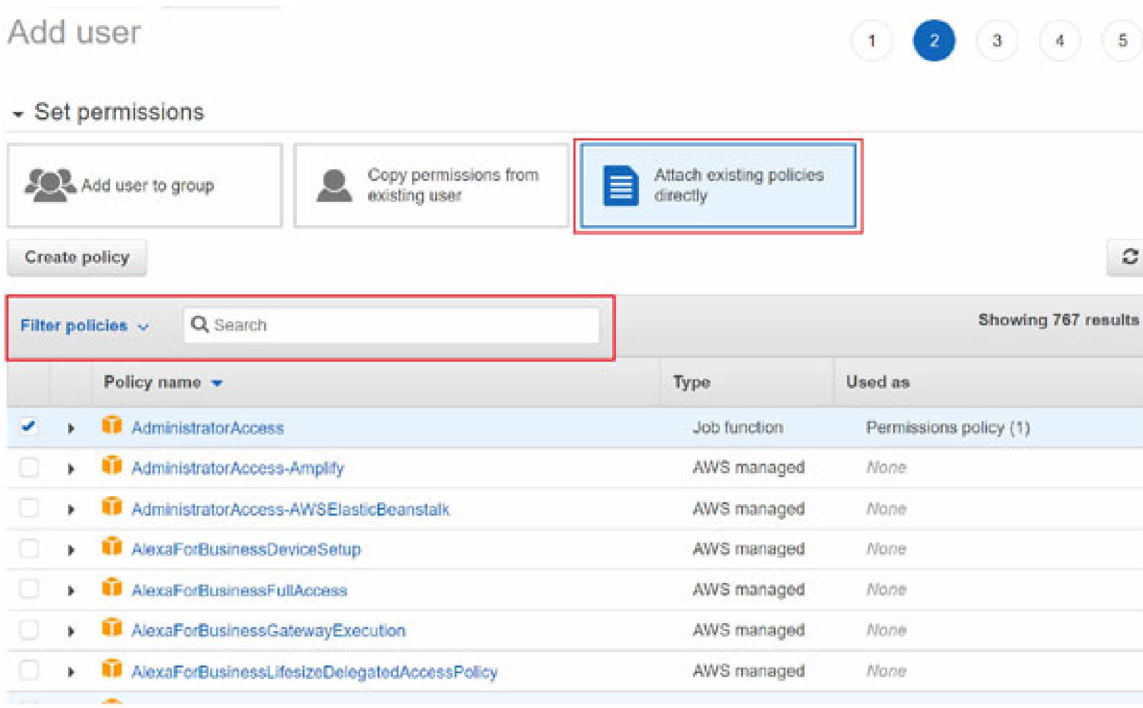


Figure 11.2: *Adding User-2*

We can pass step 3 and check the new user details in step 4, as shown in *Figure 11.3*. We can verify the username, AWS access type, and permissions we enter.

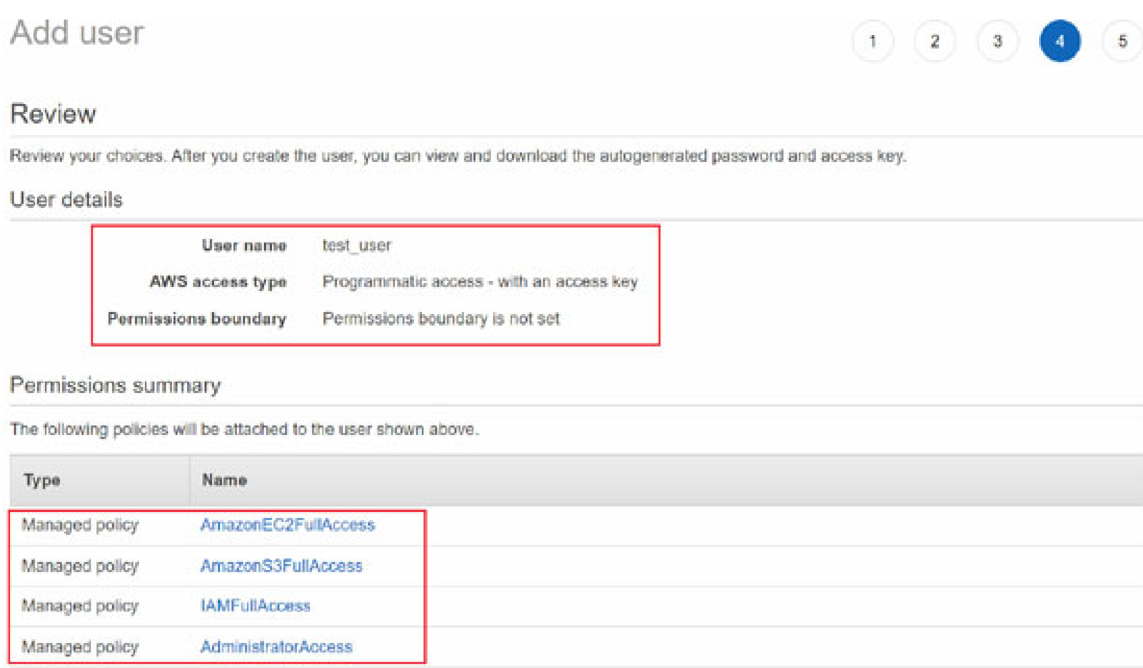


Figure 11.3: *Adding User-3*

In *Figure 11.4*, we successfully create the user after we continue to the last step. We must download the **`.csv`** file by clicking on the **`Download .csv`** button and store it on our local PC or in a safe place. This file has the **`Access key ID`** and **`Secret access key`** information, which is unique to this user. So, we should not share it with anyone.

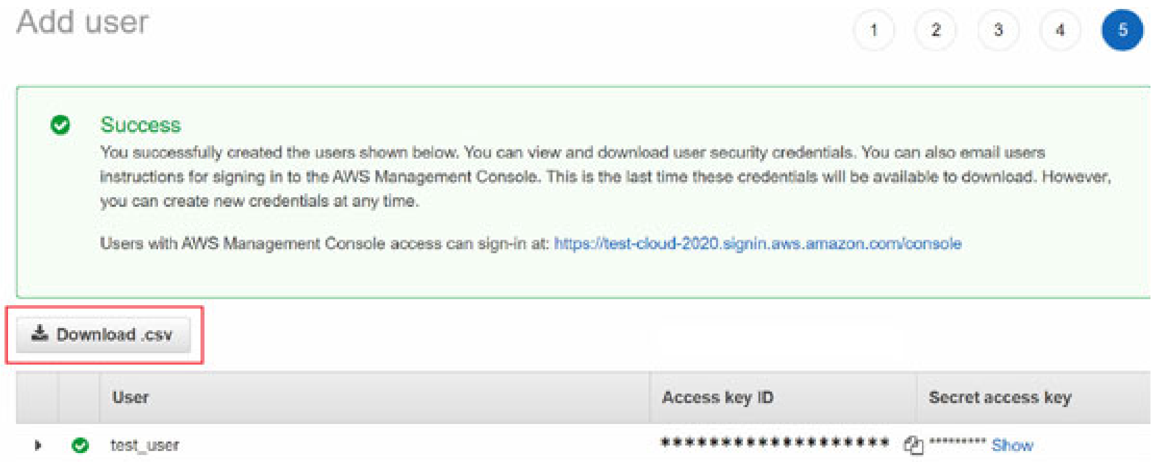


Figure 11.4: *Adding User-4*

We use this user's details to log in to AWS via IAM console, AWS CLI, or with boto3 Python scripts.

- **Install AWS CLI**: There are various options to manage AWS. We can log in to the AWS console from the AWS website; this is the most generic option. We can also use the **AWS CLI**, to use AWS command line in PC. It supports only **64-bit** devices. You can check the installation steps on the following official page.

  https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

  For Windows:

  1. Download and install the AWS CLI tool from the following link from AWS.

     https://awscli.amazonaws.com/AWSCLIV2.msi

  2. Verify the installation with the following command in the command prompt. If you see the version information, AWS CLI is successfully installed. Otherwise, reinstall the tool from the previous link or check the official installation document of AWS CLI.

     ```
     C:\> aws --version
     ```

```
aws-cli/2.7.24 Python/3.8.8 Windows/10 exe/AMD64
prompt/off
```

For Linux:

1. Download the AWS CLI ZIP file on the Linux machine with the "curl" package. If you don't have it, you must enter "sudo apt-get install curl" to install that package. After that, download the ZIP file with the following command:

   ```
   $ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
   ```

2. We need to unzip the downloaded file with the following command in the same directory that we downloaded the file in.

   ```
   $ unzip awscliv2.zip
   ```

3. Installation is complete, but we must enter an additional command to use the "**aws**" command in the AWS CLI without running the "**sudo**" command each time.

   ```
   $ sudo ./aws/install
   ```

4. We can verify the installation with the following command, which we also run for the installation on a Windows device.

   ```
   $ aws --version
   ws-cli/2.7.24 Python/3.8.8 Linux/4.14.133-113.105.amzn2.x86_64 botocore/2.4.5
   ```

We must configure the authentication parameters to use AWS CLI. All commands are the same for Linux, Windows, or MAC machines. We need to configure AWS CLI with the **aws configure** command. There are four parameters we must enter. We need to copy the **AWS Access Key ID** and **AWS Secret Access Key** values from the generated User credential CSV Excel file. After that, we enter the region name into the **Default region name**. We use **eu-west-2**, which is **London**. We write the **Default output format** parameter as a final parameter in **JSON** format.

```
C:\> aws configure
AWS Access Key ID [None]: AAAAAA
AWS Secret Access Key [None]: BBBBBB
Default region name [None]:
Default output format [None]:
```

We can modify the parameters by running the **aws configure** command again in the command prompt. These parameters are saved in the **config** and

credential files. We can find these files in the following path in Windows. You can also modify the parameters you enter by changing these files in the text editor.

C:\Users\USERNAME\.aws

In *Table 11.1*, we can see the values that we configured in the previous **aws configure** command. These are the default values when we try to log in to AWS with AWS CLI and the **boto3** module. We can change the default values by adding parameters. In the following section, we check the **boto3** parameters to change these default values by parameters.

config	credentials
[default]	[default]
region = eu-west-2	aws_access_key_id = AAAAAA
output = json	aws_secret_access_key = BBBBBB

Table 11.1: *Output of the "config" and "credentials" Files*

You can check all the details about the AWS CLI usage and commands in the following official documentation link:

https://docs.aws.amazon.com/cli

For example, we can check all EC2 instances with the **aws ec2 describe-instances** command in the command prompt. The output will be empty if you haven't created any instance yet. If there is an instance in our account in the **eu-west-2** region, we can see the similarities with the following output. It shows all details about the instance we create, such as **ImageId**, **InstanceId**, and **InstanceType**.

```
C:\> aws ec2 describe-instances
{
  "Reservations": [
   {
     "Groups": [],
     "Instances": [
      {
        "AmiLaunchIndex": 0,
        "ImageId": "ami-09e2d756e7d78558d",
        "InstanceId": "i-***************",
        "InstanceType": "t2.micro",
        "KeyName": "ec2-keypair",
        "LaunchTime": "2022-09-12T04:21:22+00:00",
        "Monitoring": {
```

```
            "State": "disabled"
        }
    }
}
```

EC2 instance management

Elastic Compute Cloud (**EC2**) is one of the most popular AWS services. It's a cloud computing system that includes server systems. We can check the details of the EC2 service in the AWS platform by searching for **`EC2`** in the **`Search`** tab. We directly created the instance from this service page.

https://console.aws.amazon.com/ec2/v2/home

In this chapter, we manage all EC2 services by Python scripts. You can check the AWS console web platform to familiarize yourself with essential AWS services.

Manage EC2 instances with Python

We created an EC2 instance with the **boto3** module in AWS. There are various instance types in AWS; servers like Linux or Windows, network devices like Cisco virtual routers, Juniper routers, Palo Alto firewalls, and many more virtual machines are in the AWS platform marketplace. This chapter focuses on Linux servers and manages them with Python scripts.

Before starting the code, you can check the **`AMI (Amazon Machine Images) Catalog`** on the EC2 platform page under the **`Images`** tab. You can see all images in the **`AWS Marketplace AMIs`** section, as shown in *Figure 11.5*.

All images that are installed on the instances have unique IDs that start with **`ami-*********`**. In the following examples, we will always use the **`ami-06672d07f62285d1d`** ID, which creates the **`Amazon Linux 2 AMI (HVM) - Kernel 5.10, SSD Volume Type`** EC2 instance as a Linux machine. AMI IDs may change if you change the AWS platform region from the AWS console.

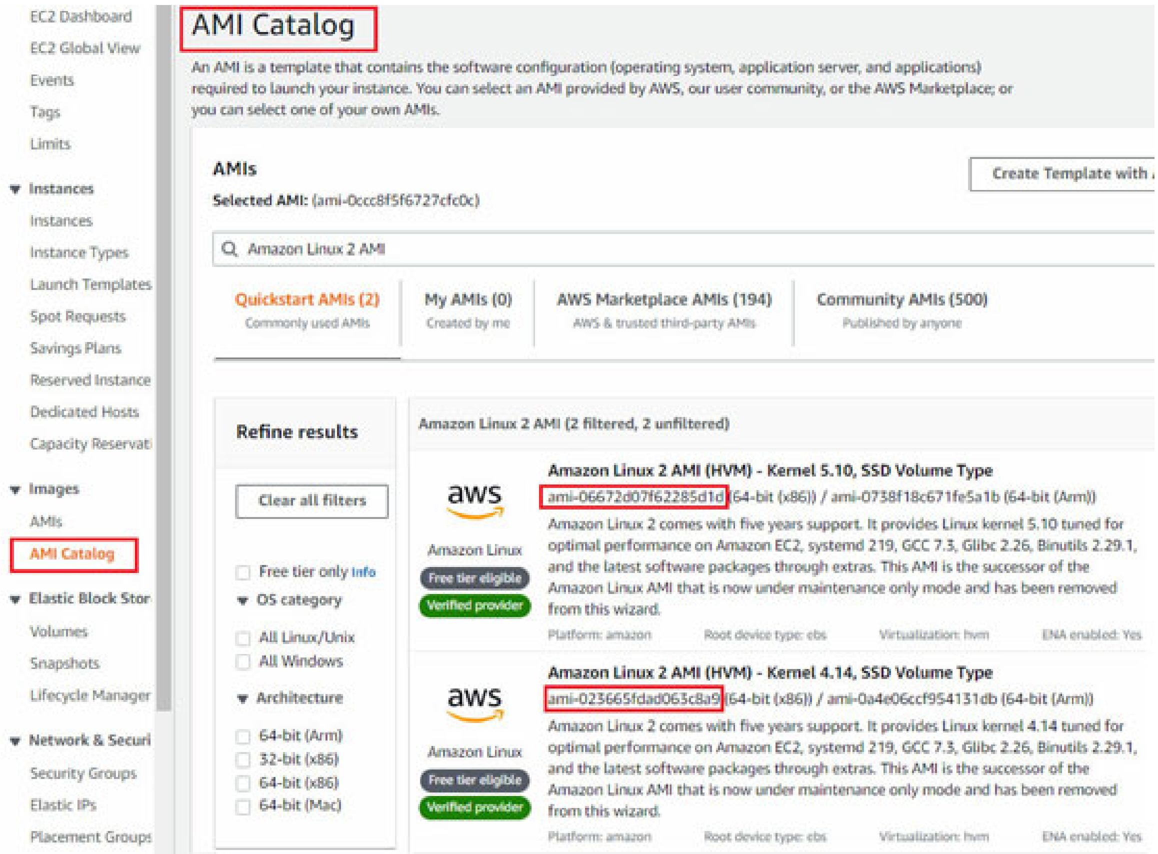


Figure 11.5: AMI Catalog in AWS Console

When we create an EC2 instance in the AWS console, as in the AWS web page, it gives an SSH connection permission to the instance in its security parameter in the security group by default. So, we can directly log in to the instance via SSH connection. However, when we create an instance with the **`boto3`** module, it may choose a security group without permission for an SSH connection. As a result, we can change the default security group parameter to permit SSH protocol.

We enter **`Network & Security > Security Groups`** page in the EC2 instance. As shown in *Figure 11.6*, enter the **`Security Groups`** page and choose the default group name on the top. Then, enter the **`Inbound rules`** tab and click on **`Edit inbound rules`**.

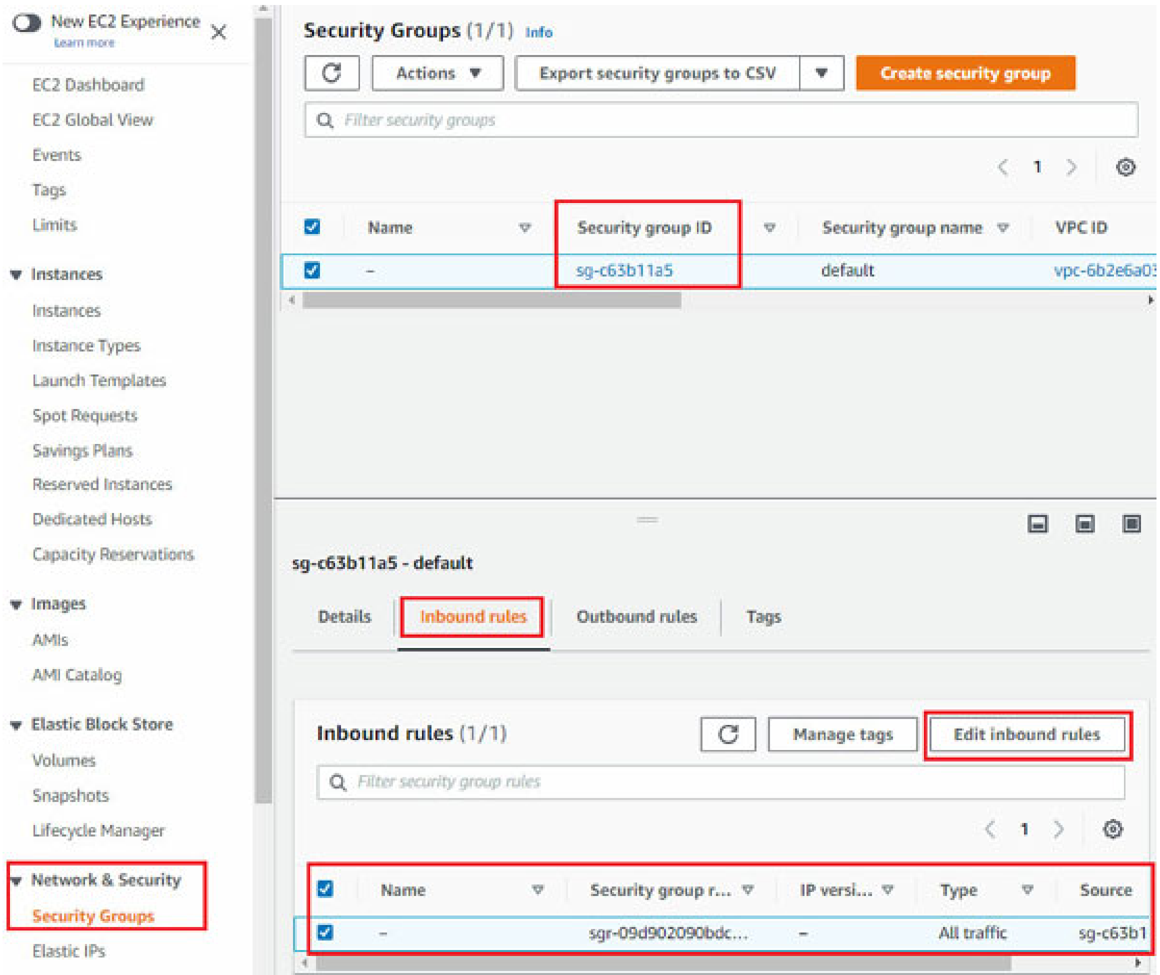


Figure 11.6: *Chaning Default Security Group-1*

In the opening page shown in *Figure 11.7*, we delete the default rule and add a new rule by choosing `SSH` in the `Type` section and `0.0.0.0/0` in the `Source` section. So, we can log in to the instance via SSH on the internet:

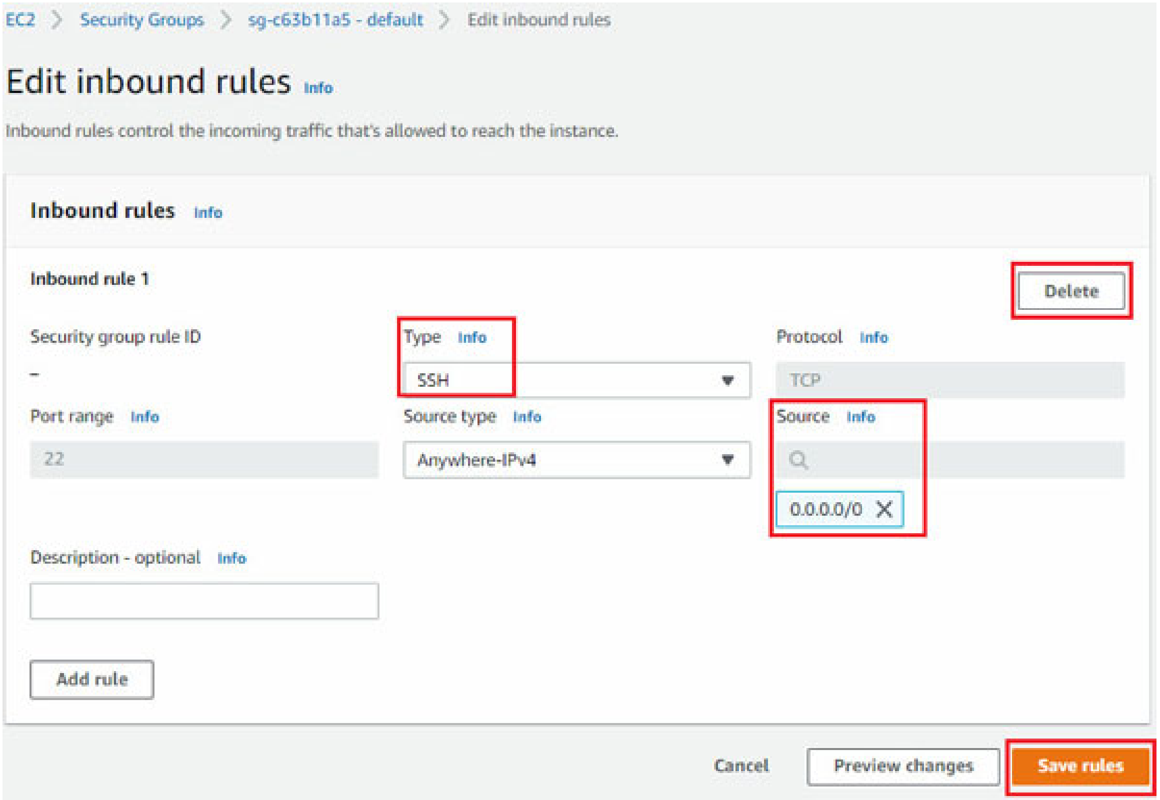


Figure 11.7: *Chaning Default Security Group-2*

There are two essential functionalities in the **`boto3`** module: **`client`** and **`resource`**. We must use one of them to make API calls to an AWS service with **`boto3`**. The **`client`** provides a low-level interface to the AWS service. On the other hand, the **`resource`** is a higher-level abstraction compared to clients. We often use the **`resource`** function in this chapter, which is more straightforward and capable than the **`client`** function.

Key pairs: Before connecting the EC2 instances by the Python code, we must create a key pair. It's required to make a secure connection to access an EC2 instance. We use the **`boto3`** module to create key pairs by Python code in *Example 11.1*.

1. We import the **`boto3`** module to use the functions for AWS services.

```
import boto3
```

2. We call the **`resource`** function and write the service name as **`ec2`**. Then, we assign resource function to a variable as **`ec2`**.

```
ec2 = boto3.resource("ec2")
```

The service name is the first parameter in the **`resource`** function. We can also write the following code by naming the service name parameter:

```
ec2 = boto3.resource(service_name="ec2")
```

The script uses default parameters that we configured with the **aws configure** in the previous section. So, when we run the code in *Example 11.1*, it creates a key pair in the **eu-west-2** region. We have an optional parameter to change the region by writing the **region_name** parameter. In the following code, we change the region to **eu-central-1**, so the code creates the key pair in that region.

```
ec2 = boto3.resource("ec2", region_name="eu-central-1")
```

3. In the following step, we create a key pair with the **create_key_pair** function by writing the key name **ec2-keypair**. This value can be anything but it should be unique to the current region.

```
key_pair = ec2.create_key_pair(KeyName="ec2-keypair")
```

4. From the following code, we get the string value of the key as an output variable.

```
output = str(key_pair.key_material)
```

5. In the last step, we save the output variable of the key pair value to a file with a **.pem** extension. It saves the file in the same directory as our script. We use this key pair to communicate with the AWS services using **boto3**.

```
with open("ec2-keypair.pem","w") as wr:
    wr.write(output)
```

Example 11.1: Create key pair for a specific region

```
import boto3
ec2 = boto3.resource("ec2")
key_pair = ec2.create_key_pair(KeyName="ec2-keypair")
output = str(key_pair.key_material)
with open("ec2-keypair.pem","w") as wr:
    wr.write(output)
```

After executing the code, it creates a file named **ec2-keypair.pem** in the same directory as our Python script. If we rerun the script, the code gives an error output: key pair named **ec2-keypair** is already created. We need to change the **KeyName** value. We should also change the **.pem** file name to be stored locally as a different key pair file.

Ouput:

```
-----BEGIN RSA PRIVATE KEY-----
MULTILINE KEY-VALUE
-----END RSA PRIVATE KEY-----
```

We can also check whether the key pairs in the AWS console have been created. First of all, we must choose the correct region name. We used **`London`** as **`eu-west-2`** as the coding in our example. We configured it in the previous section with the **`aws configure`** command in the command prompt. You can also change the region from there. We need to search as **`key pairs`** in the AWS search tab or enter the EC2 instance section and enter the **`Key Pair`** section under the **`Network & Security`** tab. *Figure 11.8* shows the key pair name we created with Python as **`ec2-keypair`**. We can create multiple key pairs with a unique key pair name for different usage.

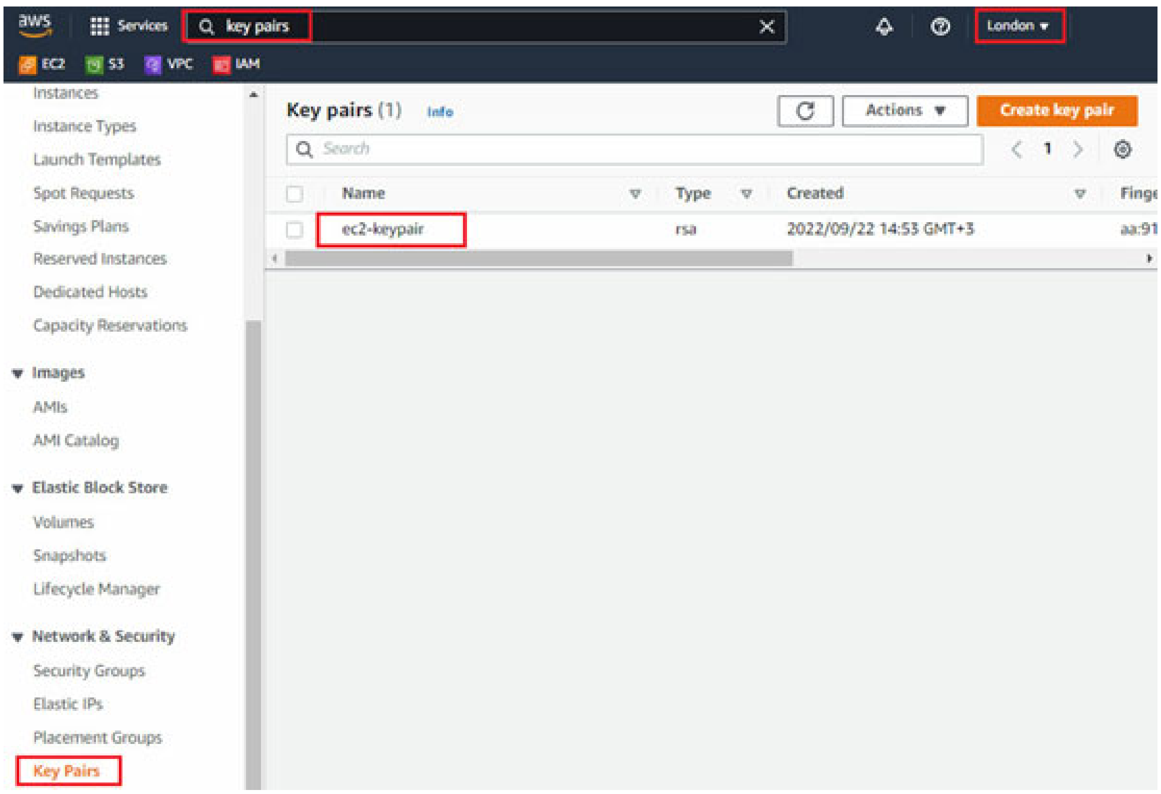


Figure 11.8: *Checking Key Pairs*

Remember that in all the upcoming examples, we create AWS services, and it may charge you according to the AWS pricing policy. So, ensure that you terminate (stop and delete objects) the deployed objects in any services, like EC2, S3 or EBS, to prevent high charges from AWS. You can check each instance's or other services' hourly prices in AWS. It depends on the region. But for testing and learning the AWS platform, the cost of a couple of hours is less then 1 USD. You can check the details on the AWS platform.

After changing the default value of the security group and creating the key pair, we can start managing EC2 instances with the **boto3** module.

In *Example 11.2*, we write our first AWS platform automation script with the boto3 module in the Python language. We create an instance with a given instance image ID as AMI. We use the **ami-06672d07f62285d1d** image name as **Amazon Linux 2 AMI (HVM) - Kernel 5.10, SSD Volume Type**.

1. We import the **boto3** module to connect to the AWS platform via a Python script.

   ```
   import boto3
   ```

2. We call the **resource** function; inside it, we write a service named **ec2**, and then we assign it to a variable named "**ec2**".

   ```
   ec2 = boto3.resource("ec2")
   ```

3. In the last step, we call the **create_instances** function to create an instance in the **eu-west-2** region as **London**. Inside this function, we add **ImageId** as the AMI value, **Mincount** as 1, and **MaxCount** as 3, creating three instances from the same AMI. We set **InstanceType** as **t2.micro** as the resource type of the virtual machine. Then we set **KeyName** as **ec2-keypair** to connect instances via key pairs.

   ```
   instances = ec2.create_instances(
    ImageId="ami-06672d07f62285d1d",
    MinCount=1,
    MaxCount=3,
    InstanceType="t2.micro",
    KeyName="ec2-keypair"
    )
   ```

Example 11.2: Create an EC2 instance with boto3 module

```
import boto3
ec2 = boto3.resource("ec2")
instances = ec2.create_instances(
   ImageId="ami-06672d07f62285d1d",
   MinCount=1,
   MaxCount=3,
   InstanceType="t2.micro",
   KeyName="ec2-keypair"     )
```

When we execute the script in *example 11.2*, it takes less than a minute to create and start three instances in the current default region. We can check the

EC2 instances in the AWS console, as in *Figure 11.9*. In the **`Instance`** tab, three instances are running, and they all have a unique instance ID. The instance type is also configured as we set in the script. The availability zone or the region is **`eu-west-2b`**, which is in the default region as **`eu-west-2`**. We can check the instance state in which all are running. In the last column, we can also see the key pair name as **`ec2-keypair`**.

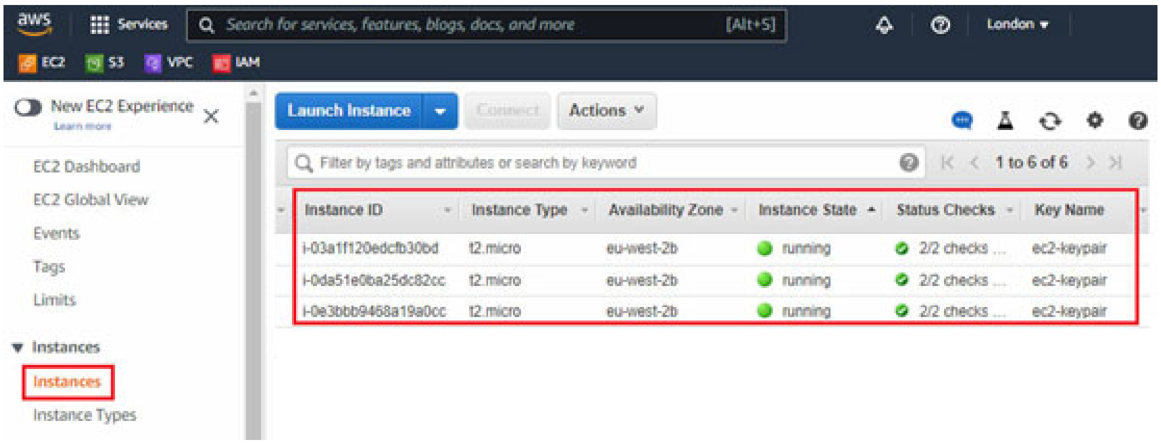


Figure 11.9: *Checking Instances in AWS Console*

After your study finishes, you can click on each instance or choose all of them and push right-click on the mouse. You can see a section as the `Instance State`, you should enter it with the `Terminate` button to stop and delete the instances. The terminated instance can still be shown in the instance list, but after a while, it disappears from the instance list. If it's in the *running* state, AWS costs you for each usage hour. We can create many instances simultaneously with a couple of lines of Python code. So, it's crucial to use scripting in the AWS platform automation.

We can choose the `EC2 Instance Connect (browser-based SSH connection)` as a **`Connection`** method and click on the `Connect` button with the default user as **`ec2-user`**. We connect to the instance in the new web page automatically via the AWS console. We can check the OS by entering **`uname -a`**, a Linux OS.

We also connect with the **`A standalone SSH client`** connection method in the following section using the **`ec2-keypair.pem`** file.

In *Example 11.3*, we get the instance information from each and display it in the output. We get the instance data from the default region as **`eu-west-2`**. We can change the region by adding the **`region_name`** in the **`resource`** function. We get all instances one by one in a **`for`** loop. We get instance İD from the **`id`** value, instance state from **`state['Name']`** value, public IPv4 address from **`public_ip_address`** value, AMI from **`image.id`** and finally, instance type from **`instance_type`**. We can see a list of the instances with their details in the output when we run the script. We can add more values to get from the AWS platform. You can check the official documentation link for more details:

https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

Example 11.3: List all EC2 instances in a region

```
import boto3
ec2 = boto3.resource("ec2")
instance_data = ec2.instances.all()
for info in instance_data:
  print("-"*20,f"\nEC2 instance ID: {info.id}")
  print(f"Instance State: {info.state['Name']}")
  print(f"Instance Public IP: {info.public_ip_address}")
  print(f"Instance AMI: {info.image.id}")
  print(f"Instance Type: {info.instance_type}")
  print("-"*20,"\n")
```

Output:

```
--------------------
EC2 instance ID: i-*************
Instance State: running
Instance Public IP: *************
Instance AMI: ami-06672d07f62285d1d
Instance Type: t2.micro
--------------------
--------------------
EC2 instance ID: i-*************
Instance State: running
Instance Public IP: *************
Instance AMI: ami-06672d07f62285d1d
Instance Type: t2.micro
--------------------
--------------------
EC2 instance ID: i-*************
Instance State: running
Instance Public IP: *************
Instance AMI: ami-06672d07f62285d1d
Instance Type: t2.micro
```

In *Example 11.4*, we create various functions to manage EC2 instances, such as starting, stopping, rebooting, and terminating. Terminating an instance means deleting an instance in the AWS platform. Even if the instance is terminated, it would be displayed in the instance list as a **`terminate`** for a while.

We use the **`InquirerPy`** module that we described in *Chapter 10: Deploying Automation Software*. We ask the user to enter the EC2 instance ID and choose the action for that instance ID, such as start, stop, reboot, or terminate. We create a function for each task and call them according to the user's choice with the **`if`** condition.

We use the **`input`** function to ask the user for the instance ID.

```
instance_id = input("Enter Instance ID: ")
```

After that, we use the resource function with the AWS service name **`ec2`**. We call the **`Instance`** function with the instance ID the user enters in the parentheses.

```
ec2 = boto3.resource("ec2")
instance = ec2.Instance(instance_id)
```

In each function that we create, we call a specific **`boto3`** function:

```
start()      #To start an instance
stop()       #To stop an instance
reboot()     #To reboot an instance
terminate()  #To terminate an instance
```

The following **`wait_until`** commands check the instance state every 15 seconds until it reaches the related state. It generates an error message after 40 failed checks.

```
wait_until_running()     #Checks the instance whether it's in the
"running" state.
wait_until_stopped()     #Checks the instance whether it's in the
"stopped" state.
wait_until_terminated()  #Checks the instance whether it's in the
"terminated" state.
```

To execute the **`InquirerPy`** module, we need to change the running settings in the Pycharm tool, as we did in the last chapter. We enter the section `Run/Edit Configurations` tab. In the opening window, we enable the `Emulate terminal in output console` option and apply it.

Example 11.4: Manage EC2 instance

```
import boto3
from InquirerPy import inquirer
def ec2_start():
    instance.start()
    print(f"Starting EC2 instance: {instance.id}")
    instance.wait_until_running()
    print(f"----\nEC2 Instance ID: {instance.id} \nStatus:
    Started")
def ec2_stop():
    instance.stop()
    print(f"Stopping EC2 instance: {instance.id}")
    instance.wait_until_stopped()
    print(f"----\nEC2 Instance ID: {instance.id} \nStatus:
    Stopped")
def ec2_reboot():
    instance = ec2.Instance(instance_id)
    instance.reboot()
    print(f"----\nEC2 Instance ID: {instance.id} \nStatus:
    Rebooted")
def ec2_terminate():
    instance.terminate()
    print(f"Terminating EC2 instance: {instance.id}")
    instance.wait_until_terminated()
    print(f"----\nEC2 Instance ID: {instance.id} \nStatus:
    Terminated")
instance_id = input("Enter Instance ID: ")
main_task = inquirer.select(
  message="Choose action for Instance:",
  choices=["Start EC2 Instance", "Stop EC2 Instance", "Reboot EC2
  Instance", "Terminate EC2 Instance", "Exit"]).execute()
ec2 = boto3.resource("ec2")
instance = ec2.Instance(instance_id)
if main_task == "Start EC2 Instance":
    ec2_start()
elif main_task == "Stop EC2 Instance":
    ec2_stop()
elif main_task == "Reboot EC2 Instance":
    ec2_reboot()
elif main_task == "Terminate EC2 Instance":
```

```
    ec2_terminate()
elif main_task == "Exit":
    print("Exit from the task.")
```

In *Example 11.5*, we change the settings of an instance. We replace the instance type from the current type with the **t2.small** type. We import the **boto3** module, call the resource function with the **ec2** service, and then use the **instance** function in *Example 11.4*. Before changing the attribute, we must stop the instance, and then we execute the **stop** function to stop.

After we verify that the instance has stopped with the **wait_until_stopped** function, we run the **modify_attribute** function with a parameter as the **InstanceType**. The value of this parameter is **t2.small**, which is one of the AWS platform instance types.

After we change the instance type, we start the instance with a new type: **t2.small**.

Example 11.5: Manage EC2 Instance

```
import boto3
instance_id = "i-0aabe4b7adb32fd87" #Enter your instance ID in
string
ec2 = boto3.resource("ec2")
instance = ec2.Instance(instance_id)
print(f"Instance ID: {instance_id} \nCurrent Instance Type:
{instance.instance_type}")
instance.stop()
print("---\nInstance is stopping")
instance.wait_until_stopped()
print("---\nInstance is stopped")
instance.modify_attribute(InstanceType={"Value": "t2.small"})
print("---\nInstance type is changed")
instance.start()
print("---\nInstance is starting")
instance.wait_until_running()
print(f"---\nInstance started with the new instance type.")
print(f"Instance ID: {instance_id} \nCurrent Instance Type:
{instance.instance_type}")
```

We add various print functions to the script because stopping and starting the instance takes time. So, we can see the process of our script. Initially, we get an instance ID and collect the instance type. After the operation finishes, we collect the same data from the instance. The instance type changes from **t2.micro** to **t2.small** at the end of the output.

```
Instance ID: i-0aabe4b7adb32fd87
Current Instance Type: t2.micro
---
Instance is stopping
---
Instance is stopped
---
Instance type is changed
---
Instance is starting
Instance started with the new instance type.
Instance ID: i-0aabe4b7adb32fd87
Current Instance Type: t2.small
```

In *Example 11.6*, we collect all instances in the default region with the instance state. It can be **running**, **stopped**, and **terminated**. We create six instances, out of which we stop two and terminate two. So we have two instances in the **running**, **stopped**, and **terminated** states each, with a total of six instances, as shown in *Figure 11.10*. You can create all instances via the AWS console.

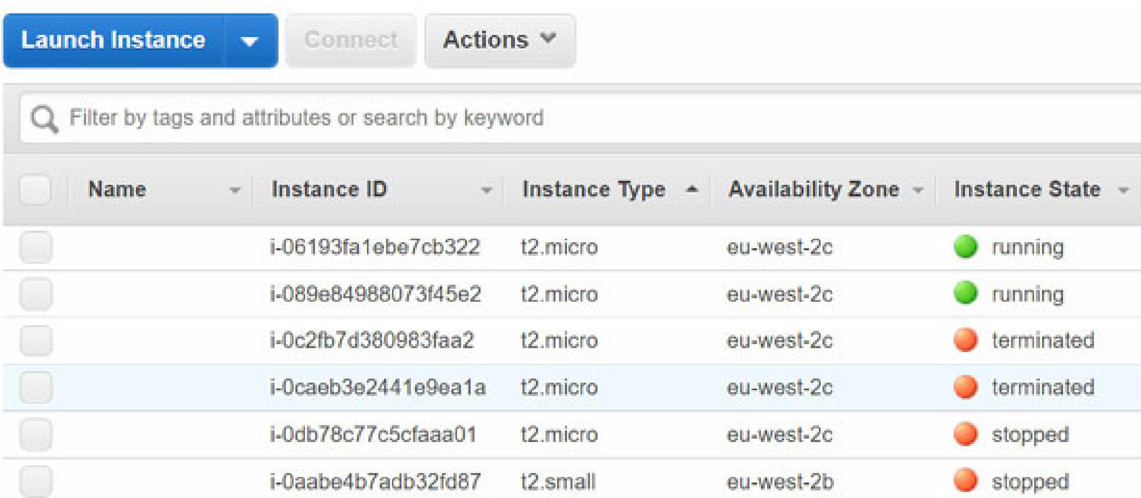


Figure 11.10: *Instance List in AWS Console*

We use the filter function. Inside parentheses, we add the **name** parameter **instance-state-name** with the values **["stopped", "terminated","running"]**. If the instance state matches with one of them, it filters the instance.

```
import boto3
ec2 = boto3.resource("ec2")
instances = ec2.instances.filter(
    Filters=[{"Name": "instance-state-name", "Values": ["stopped",
    "terminated", "running"]}])
```

After that, we display all filtered instances in each line with their instance state. So, we use the **for** loop to get all of them one-by-one. We call the **id** value for the instance ID and the **state['Name']** value for the instance state.

```
for info in instances:
    print(f"Instance ID: {info.id} - InstanceState:
    {info.state['Name']}")
```

In the **instance-state-name** filtering, there are additional values, such as **pending**, **shutting-down**, and **stopping** states. We don't use these values in this example. After executing the script, we can see the following output with six instances, including the current states.

Output:

```
Instance ID: i-0c2fb7d380983faa2 - InstanceState: terminated
Instance ID: i-0db78c77c5cfaaa01 - InstanceState: stopped
Instance ID: i-0caeb3e2441e9ea1a - InstanceState: terminated
Instance ID: i-089e84988073f45e2 - InstanceState: running
Instance ID: i-06193fa1ebe7cb322 - InstanceState: running
Instance ID: i-0aabe4b7adb32fd87 - InstanceState: stopped
```

The output changes if we change the values list in the **instance-state-name**. For example, if we try to get the instances that are **running**, we only write the **running** string as an item of the **Values**.

```
instances = ec2.instances.filter(
    Filters=[{"Name": "instance-state-name", "Values":
    ["running"]}])
```

Output:

```
Instance ID: i-089e84988073f45e2 - InstanceState: running
Instance ID: i-06193fa1ebe7cb322 - InstanceState: running
```

Connection to EC2 instances

We already write scripts to manage the EC2 instances. In this section, we will try to connect instances and collect data, as we did in the previous chapters.

These instances can be a virtual router, firewall, or server. So, we use the same method to log in to these instances by running paramiko or netmiko modules.

We can also connect to instances directly from a PC. We use the key pair file with an extension like **.pem**. We can connect an instance with its public IP address, as shown in *Figure 11.11*. We enter the `EC2` service page in the AWS console. Then, we click on one of the running instances. At the bottom, a page shows a `Description` tab. Inside this tab, we can see the public IP address of the instance. There is also a public DNS link in the same tab. We use these public IP addresses or Public DNS links to log in via SSH tools or modules in Python.

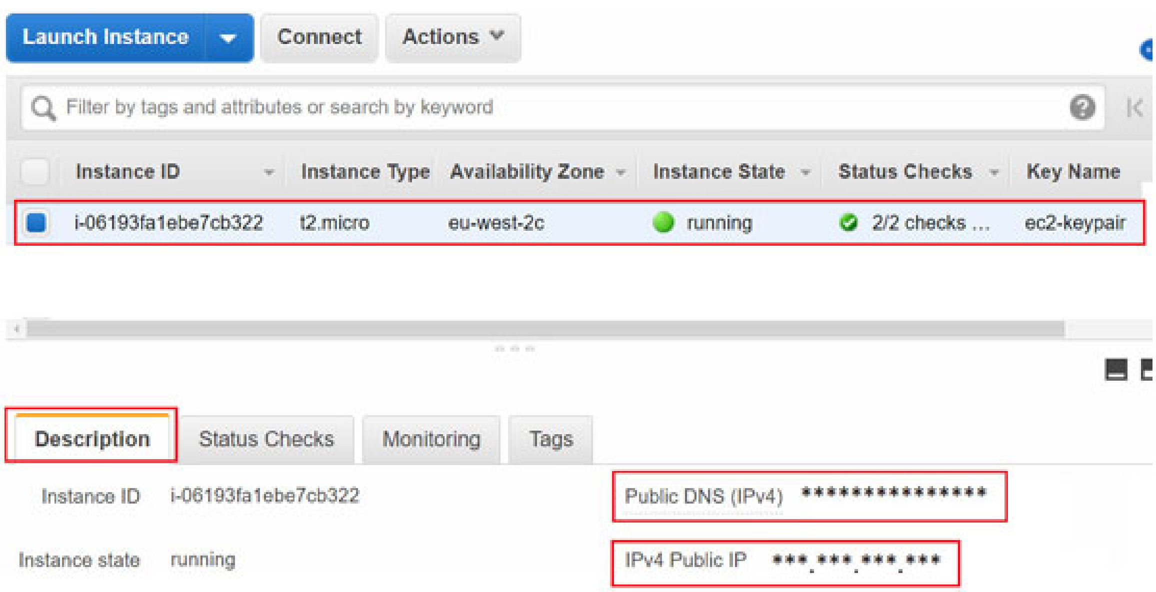


***Figure 11.11:** Finding Public IP Address of an Instance in AWS Console*

In **Linux** machines, we need to go to the same directory with the **ec2-keypair.pem** file in the terminal. After that, we enter the following command in the terminal to change the permission of the **.pem** file.

```
# chmod 400 ec2-keypair.pem
```

Then, we execute the **ssh** command with the **-i** parameter to use the key pair file to authenticate the remote device. After that, we write the default username as **ec2-user**, with the **@** character. Finally, we write the public IP address we get from the AWS Console.

```
# ssh -i ec2-keypair.pem ec2-user@PUBLIC_IP
```

In **Windows** machines, we can use a similar way in command prompt. First, we change the permissions of the key pair file and enter the PowerShell tool in Windows. Then, we go to the directory where our key pair file is and enter the following commands in PowerShell. If all the commands' output is successful, the operation finishes. If one of them fails, you can check for a solution on the internet.

```
icacls.exe "ec2-keypair.pem" /reset
icacls.exe "ec2-keypair.pem" /grant:r "$($env:username):(r)"
icacls.exe "ec2-keypair.pem" /inheritance:r
```

After changing the file permissions, we can use the command prompt or PowerShell to connect to the instance via SSH protocol.

```
ssh -i ec2-keypair.pem ec2-user@PUBLIC_IP
```

As an option, we can also use an open-source tool like **Putty** as an SSH connection tool. We must convert the key pair file from the **.pem** file to the **.ppk** file with the **Puttygen** tool. Then, we change the authentication mode by adding the **.ppk** file in Putty. You can check the details about connection with Putty tool in the following AWS official document:

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html?icmpid=docs_ec2_console

Paramiko module: One of the most popular SSH connection module in Python is the **paramiko** module. In *Example 11.6*, we use the **paramiko** module to connect an instance and execute a command. We looked at several examples of the **paramiko** module in the previous chapters; the only difference is the usage of the **connect** function in this example. We enter the hostname value as a public IP address or the DNS we get from the AWS console. Username is the default username **ec2-user**, and a new parameter named **key_filename** is used to set the key pair file to **ec2-keypair.pem**. Instead of entering passwords, we use the key pair authentication method between our PC and the instance in the AWS platform.

Example 11.6: Connect EC2 instance with paramiko

```
from paramiko import SSHClient, AutoAddPolicy
from time import sleep
client = SSHClient()
client.set_missing_host_key_policy(AutoAddPolicy())
client.connect(hostname="PUBLIC IP_OR_DNS", username="ec2-user",
key_filename= "ec2-keypair.pem")
commands = client.invoke_shell()
commands.send("uname -a \n")
sleep(1)
output = commands.recv(1000000)
```

```
output = output.decode("utf-8")
client.close()
print(output)print("---\nInstance type is changed")
instance.start()
print("---\nInstance is starting")
instance.wait_until_running()
print(f"---\nInstance started with the new instance type.")
print(f"Instance ID: {instance_id} \nCurrent Instance Type:
{instance.instance_type}")
```

Netmiko module: We used the netmiko module several times in the previous chapters. In *Example 11.7*, we use the **`netmiko`** module to log in to an instance. The only difference here is that instead of **`password`**, we use the **`key_file`** parameter with the **`ec2-keypair.pem`** value, which is the file name of our key pair. The code is more straightforward, and the output is more evident with the **`netmiko`** module.

Example 11.7: Connect EC2 instance with netmiko

```
from netmiko import Netmiko
device = {"host": "18.170.25.70", "username": "ec2-user",
"device_type": "linux", "key_file": "ec2-keypair.pem"}
net_connect = Netmiko(**device)
output = net_connect.send_command("uname -a", strip_command=False)
print(output)
```

In *Example 11.8*, we create an instance with **`boto3`** and execute a command in the new instance. After we create the instance, we get the instance ID with the **`instance_id`** parameter. We wait until the instance is in the **`running`** state with the **`wait_until_running`** function.

Then, we reload the instance to update the **`instance`** attributes to avoid getting the public IP address from the instance data. The remaining part is what we did in *Example 11.7* by connecting the instance with netmiko and executing the command inside.

Example 11.8: Create and log in an instance with executing commands

```
import boto3
from netmiko import Netmiko
ec2 = boto3.resource("ec2")
instances = ec2.create_instances(
    ImageId="ami-06672d07f62285d1d",
    MinCount=1,
```

```
    MaxCount=1,
    InstanceType="t2.micro",
    KeyName="ec2-keypair"
)
instance_id = instances[0].instance_id
print(f"{instance_id} Instance is created")
instances[0].wait_until_running()
print(f"Instance is started")
instances[0].reload()
public_ip = instances[0].public_ip_address
print(f"Public IP: {public_ip}")
device = {"host": public_ip, "username": "ec2-user",
"device_type": "linux", "key_file": "ec2-keypair.pem"}
net_connect = Netmiko(**device)
output = net_connect.send_command("uptime", strip_command=False)
print(output)
```

When we execute the script, it creates a new instance, gets the instance ID of the new instance, and starts the instance. After that, it collects the instance's public IP address and connects with the `netmiko` module. In the end, it executes the specific command and displays it in the output. All steps are shown in the following output with the `print` functions.

Output:

```
i-0156f7378bee20917 Instance is created
Instance is started
Public IP: ***.***.***.***
uptime
20:12:42 up 0 min,  1 user,  load average: 0.23, 0.05, 0.02
```

S3 bucket management

In this section, we continue with another popular service of the AWS platform: **S3** (**Simple Storage Service**). In EC2, objects are called instances, and in S3, objects are called buckets. So, we manage S3 buckets with Python scripts here. The AWS platform provides an S3 storage system to store files in the AWS cloud. We write simple scripts, such as creating or deleting buckets and managing the files in buckets.

You can search for S3 in the search bar on the AWS console to check the details of the S3 service and the bucket list. We are still in the same region as

eu-west-2, i.e., London.

In *Example 11.9*, we use the **create_bucket** function to create a bucket and the **delete** function to delete a bucket. S3 buckets are a global namespace, meaning that all bucket names must be unique in the AWS platform. If any user in AWS uses the bucket with a name, we cannot use the same bucket name again. If it's identical, the following code gives an error message. So, we must write a unique bucket name.

We call the resource function from the boto3 module with another service called **s3**. After that, we can use all functions inside the S3 service. To create a bucket, we must write two parameters: **Bucket** as a bucket name and **CreateBucketConfiguration** as a region name. We use the **eu-west-2** region as London. When we execute this function by writing the bucket name in the following code, AWS creates a bucket for us if the bucket name is unique.

```
create_bucket("test-storage-Python-1")
```

We use the **Bucket** function with the bucket name to delete a bucket. Then, we call it the **delete** function. If the bucket exists in our region, AWS deletes the bucket permanently.

```
  bucket = s3.Bucket(bucket_name)
  bucket.delete()
```

Example 11.9: Create and delete buckets

```
import boto3
s3 = boto3.resource("s3")
def create_bucket (bucket_name):
  s3.create_bucket(Bucket=bucket_name, CreateBucketConfiguration={
  "LocationConstraint": "eu-west-2"})
def delete_bucket (bucket_name):
  bucket = s3.Bucket(bucket_name)
  bucket.delete()
create_bucket("test-storage-Python-1")
```

After running the script, you can check the S3 service in AWS Console to see whether the buckets are created or deleted. Remember to delete a bucket that must be empty, without including any files or folders inside it.

In *Example 11.10*, we write a Python automation script to upload a file from a local PC to the AWS S3 bucket and download a file from the AWS S3 bucket to the local PC. We also have a function to delete a file from a bucket.

We use the **Object** function for the bucket name and the file on the S3 bucket. Then, we call the **upload_file** function with the local file name in the local

PC to upload a file to the S3 bucket. We call the **download_file** function with the local file name in the local PC to download a file from the S3 bucket to the local PC.

Finally, we call the **delete** function to delete the file on the bucket that we mentioned in the parameter of the **Object** function.

Example 11.10: Upload, download and delete files in buckets

```
import boto3
bucket_name = "test-storage-Python-1"
local_file = "test.txt"
file_on_bucket = "test.txt"
s3 = boto3.resource("s3")
s3_object = s3.Object(bucket_name, file_on_bucket)
def uploading(local_file):
  s3_object.upload_file(local_file)
def downloading(local_file):
  s3_object.download_file(local_file)
def deleting():
  s3_object.delete()
  print("S3 object deleted")
uploading(local_file)
```

In *Example 11.11*, we copy a file from one bucket to another. We create two variables called **source** and **destination**. In the **source** variable, we create a dictionary with two items, such as **Bucket** with the source bucket name and **Key** with the file to be transferred. We call the bucket function in the **destination** variable by writing the destination bucket name in parentheses.

In the end, we call the **copy** function for the destination variable by adding the source dictionary and the destination file name as **test.txt**.

When we execute the code, the **test.txt** file from the **test-storage-Python-1** bucket is copied to the **test-storage-Python-2** bucket with the name called **test.txt**.

Example 11.11: Copy files from a bucket to another

```
import boto3
s3 = boto3.resource("s3")
source= { "Bucket" : "test-storage-Python-1", "Key": "test.txt"}
destination = s3.Bucket("test-storage-Python-2")
destination.copy(source, "test.txt")
```

We can also list all S3 buckets in the default region. We use the `s3.buckets.all` code in a **for** loop. Then, we call the `name` variable from the `bucket` iterable. The following code lists all buckets when we execute it:

```
import boto3
s3 = boto3.resource("s3")
print("All Bucket Lists:")
for bucket in s3.buckets.all():
  print(f"- {bucket.name}")
```

Output:

```
All Bucket Lists:
- test-storage-Python-1
- test-storage-Python-2
```

In *Example 11.12*, we list all items or objects in an S3 bucket. We get the specific bucket name data to the `s3_bucket` variable with the `s3.Bucket(bucket_name)` function. After that, we check all objects in the `s3_bucket` variable with `s3_bucket.objects.all` in the `for` loop.

Inside the loop, we print the `key` value, which is the file name with its extension, and the `size` value, which is the file size in bytes.

Example 11.12: List all items in a bucket

```
import boto3
bucket_name = "test-storage-Python-1"
s3 = boto3.resource("s3")
s3_bucket = s3.Bucket(bucket_name)
for item in s3_bucket.objects.all():
  print(f"{item.key} - Size: {item.size} Bytes")
```

This code gets all the objects in the S3 bucket. Even if we have folders including files, it shows all objects or items in that bucket, as in the following output:

```
test1.txt - Size: 2112 Bytes
test2.txt - Size: 1200 Bytes
test_folder/ - Size: 0 Bytes
test_folder/test3.txt - Size: 1968 Bytes
```

EBS volume management

Elastic Block Store (**EBS**) is another storage service provided by the AWS platform. It's inside the EC2 service, and EC2 instances use it for block-level storage. It offers high performance, durability, and scalable volumes, as in **Tebibyte (TiBs).**

EBS volumes have an option for snapshots of volumes in which we can store all backup data in different availability zones. It protects our data and is highly reliable.

You can check the EBS section in the AWS console on the EC2 page which is listed in *Figure 11.12*. All EBS volumes in the specific region are listed in *Figure 11.12*.

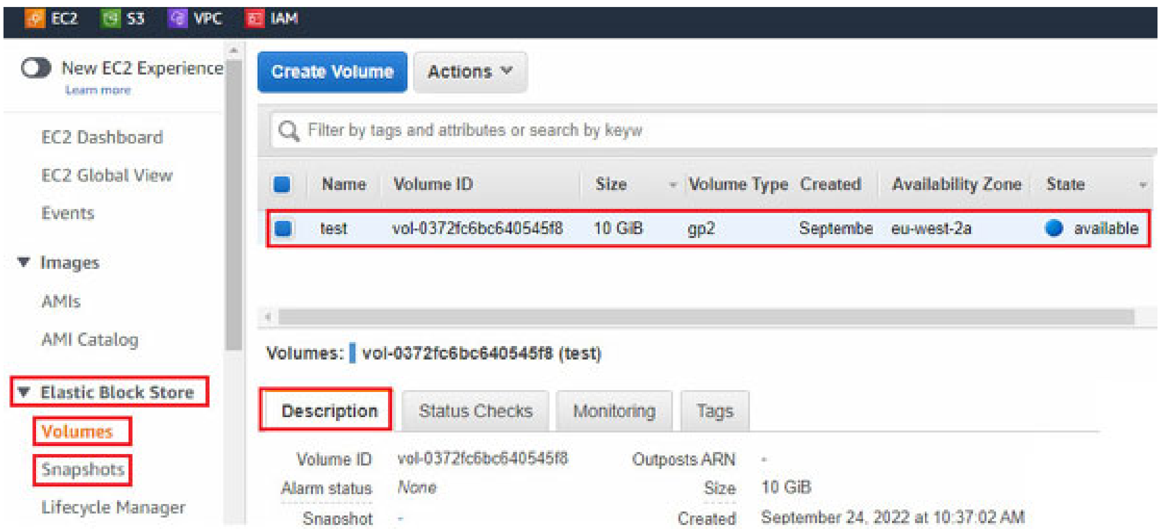


Figure 11.12: *EBS Section in AWS Console*

There are various volume types in EBS, such as **`gp2 (General Purpose SSD)`**, **`io1 (Provisioned IOPS SSD)`**, **`st1 (Throughput Optimized HDD)`**, and **`sc1 (Cold HDD)`**. You can check the details of each volume type in the AWS documentation.

Manage EBS volumes

We can manage EBS volumes with the boto3 module, as we did in EC2 and S3 services. In *Example 11.13*, we create an EBS volume using the Python script.

We use the **`ec2`** service with the resource function from the **`boto3`** module. After that, we call the **`create_volume`** function to create an EBS volume. There are two mandatory parameters to create an EBS volume: **`AvailabilityZone`** and **`Size`**. We must define the availability zone and the volume size. In the following example, the availability zone is **`eu-west-2a`**. These zones are inside the **eu-west** region. We add **`a,b,c`** to the end of the region name to write an availability zone. We set the volume size to 20 GB, and we can change it to any value. In the end, we show the new volume ID by calling the **new_volume.id**.

Example 11.13: Create an EBS volume

```
import boto3
ec2 = boto3.resource("ec2")
new_volume = ec2.create_volume(AvailabilityZone="eu-west-2a",
Size=20)
print(f"Created volume ID: {new_volume.id}")
```

We can also add optional parameters inside the **create_volume** function. With the **VolumeType** parameter, we can configure the volume type when we create a volume, and it's **gp2** by default.

```
volume = ec2.create_volume(AvailabilityZone="eu-west-2a", Size=20,
VolumeType="gp2")
```

We can also add a tag to a volume. To do that, we must use the **TagSpecifications** parameter. We set **ResourceType** and **Tags** parameters. In the **Tags**, we select the key as **Name** and the value as **test-tag-name**, which will be our new volumes tag.

```
volume = ec2.create_volume(AvailabilityZone="eu-west-2a", Size=20,
TagSpecifications=
[
    {
    "ResourceType": "volume",  "Tags": [
     {"Key": "Name", "Value": "test-tag-name"}]}])
```

When we execute the script, we can see an output with the generated volume's ID. You can also check the volume in the AWS Console.

Output: **Created volume ID: vol-00bda8ef88e6f21ad**

It's also recommended to delete all values after your study to prevent any chances of them being used.

Create snapshots of EBS volumes

EBS volume has a data protection feature that creates a copy of the volume, which is called a snapshot. These snapshots can be stored in different regions or availability zones to protect the data. You can check the created snapshots on the EC2 service page, under the `Elastic Block Store` tab as `Snapshots`, as shown in *Figure 11.12*.

In *Example 11.14*, we create a snapshot of an existing volume with the **`create_shapshot`** function. Inside parentheses, we write the **`VolumeId`** parameter to choose the target volume to create a backup or snapshot.

Example 11.14: Create a snapshot of an EBS volume

```
import boto3
ec2 = boto3.resource("ec2")
volume_id = "vol-0ae620def39c1c379"
snapshot_volume = ec2.create_snapshot(VolumeId=volume_id)
print(f"Original Volume: {volume_id} \nSnapshot Volume:
{snapshot_volume.id}")
```

When we execute the script, we display the original volume and the snapshot volume ID. Volume IDs start with **`vol-`**, and snapshot volume IDs begin with **`snap-`**.

Output:

```
Original Volume: vol-0ae620def39c1c379
Snapshot Volume: snap-0206bfa631894b75f
```

Attach EBS volume to EC2 instance

In *Example 11.15*, we attach an EBS volume to an EC2 instance. We must have at least one available EBS volume not connected to an instance and an active instance to attach the volume. We check the volume state at the beginning of the code, and we check the status at the end. If the EBS volume is not attached to an instance, it's **`available`**. If it's attached to an instance, it's in the **`in-use`** state.

We use the **`attach_to_instance`** function to attach the volume to an instance; we must add two mandatory parameters: **`Device`** and **`InstanceID`**. The **`Device`** parameter is used to expose to the instance (for example, **`/dev/sdh`** or **`xvdh`**).

If the instance is an AWS marketplace instance, the instance must be stopped before the attachment of the EBS volume. Otherwise, as we did in this example, we can attach an EBS volume to an instance in the **`running`** state.

Example 11.15: Attach EBS Volume to an EC2 Instance

```
import boto3
ec2_resource = boto3.resource("ec2")
volume = ec2_resource.Volume("vol-0ae620def39c1c379")
print(f"Volume: {volume.id}   Status: {volume.state}")
```

```
volume.attach_to_instance(Device="/dev/sdh", InstanceId="i-0d1d8dd7bc1887539")
print(f"Volume: {volume.id}   Status: {volume.state}")
```

When we execute the script, in the output, we can see the volume's status at the beginning and the end of the code. After it attaches to an instance, the status of the EBS volume changes from `available` to `in-use`.

Output:

```
Volume vol-0ae620def39c1c379 status -> available
Volume vol-0ae620def39c1c379 status -> in-use
```

We can also check the instance by connecting via SSH. In Linux machines, we can check all volumes and their size information with the `lsblk` command. Before executing the script, we have only one drive: `xvda`, 8 GB. After running the script, a new drive is added: `xvdh`, 20 GB. This is the volume that we attach to this instance.

```
Before script:
[ec2-user@IP_ADDRESS /]$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvda 202:0 0 8G 0 disk
└─xvda1 202:1 0 8G 0 part /
After script:
[ec2-user@IP_ADDRESS /]$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvda 202:0 0 8G 0 disk
└─xvda1 202:1 0 8G 0 part /
xvdh 202:112 0 20G 0 disk
```

We can also detach EBS volumes from the EC2 instances using the `detach_from_instance` function. We write the same code, only changing the function name. This function also has two mandatory parameters: `Device` and `InstanceID`.

```
volume.detach_from_instance(Device="/dev/sdh", InstanceId="i-09b0e33f1e8a88c37")
```

IAM user management

AWS has another essential service called **Identity and Access Management** (**IAM**). It's a user management service related to user permissions and access control, allowing us to manage all users and access levels in the AWS console.

We can create multiple users in a single AWS account, so we can divide users into groups to give access to the specific services according to the related teams. We can even manage the billing section for the particular user.

In *Example 11.16*, we create a user with the IAM service. So, we call the **iam** value in the **resource** function and assign it to a variable. After that, we create the user with the **create_user** function. Inside parentheses, we write the **UserName** parameter with its value as **abcd**. In the end, it displays the output. When we execute this script, it creates a new user called **abcd**.

Example 11.16: Create a new user in IAM service

```
import boto3
iam = boto3.resource("iam")
result = iam.create_user(UserName="abcd")
print(result)
```

In the following example, we list all the users in the IAM service. We get user data from **users.all** and assign it to a variable in a list type. After that, we use the **for** loop to get each item's name value to get the username. The script displays all active users in the IAM service when we execute the script.

```
import boto3
iam = boto3.resource("iam")
users = list(iam.users.all())
for user in users:
  print(user.name)
```

In *Example 11.17*, we add new roles and remove current roles from users in the IAM service. We use the **attach_policy** function with the policy name to add a new role to a user and the **detach_policy** function to remove a current role from a user. We add policy **Amazon Resource Name** (**ARN**) with the role name at the end in the following example.

When we call the customized **add_role** function with the username and the role name, the script adds the particular role to the specified user. You can check the user permissions from the IAM service in the AWS Console.

Example 11.17: Add and remove roles from users

```
import boto3
def add_role(username,role_name):
  policy_arn = f"arn:aws:iam::aws:policy/{role_name}"
  iam = boto3.resource("iam")
  iam.User(username).attach_policy(PolicyArn=policy_arn)
def remove_role(username,role_name):
```

```
  policy_arn = f"arn:aws:iam::aws:policy/{role_name}"
  iam = boto3.resource("iam")
  iam.User(username).detach_policy(PolicyArn=policy_arn)
add_role("abcd","AmazonEC2FullAccess")
```

Conclusion

This chapter taught us about creating automation scripts in the AWS cloud platform with the Python script. We made an environment in AWS Console to connect it with the scripts we created. We used the **`boto3`** module to handle tasks like managing EC2 instances, S3 buckets, EBS volumes, and user access and permissions in the IAM Service.

Multiple choice questions

1. Which of the following services are managed through the AWS user access management?
    a. EC2
    b. VPC
    c. S3
    d. IAM
2. Which of the following functions create an EC2 instance?
    a. `create_ec2_instance()`
    b. `instance()`
    c. `create_instances()`
    d. `add_instances()`
3. Which of the following feature is wrong about EBS volumes to compare the S3 bucket?
    a. High performance
    b. Cheaper
    c. Easy data backup via snapshots
    d. Highly scalable

Answers

1. d
2. c
3. b

Questions

1. Write a script to create 3 EC2 instances and attach 5 GB of EBS volumes for each.
2. Write a script to detach an EBS volume from an EC2 instance and attach it to another EC2 instance.

Index

A

Address Resolution Protocol (ARP) packets 343
Amazon Web Services (AWS) 396
  EBS (Elastic Block Store) 396
  EC2 (Elastic Compute Cloud) 396
  IAM (Identity and Access Management) 397
  S3 (Simple Storage Service) 396
Ansible 96
AWS account
  creating 397
  user, creating 397-399
AWS CLI
  for Linux 400, 401
  for Windows 400
  installing 399
  usage 401, 402

B

boto3 module 397
  installing 397
break statement 53, 54

C

classes 91
close function 68
cloud environment deployment 396
  AWS 396, 397
collecting logs 118
  CPU levels, collecting 122-126
  duplicated IP address, finding 126-128
  version and device information, collecting 118-122
  with multithreading 129-132
  with SNMP 257-262
Command Line Interface (CLI) 2
concurrent module
  using 314
connection modules 96
  SSH connection 99
  telnet connections 112
continue statement 54, 55
CPU levels

plotting 240-243
crontab service 348

D

data plotting 235-238
CPU levels, plotting 240-243
interface bandwidth, plotting 243-245
data types 28, 29
dictionary 30, 41, 42
float 29
integer 31
list 29, 35-37
set 30
string 29, 30
tuple 29
devices. *See* network devices
dictionary 41, 42
for loop statements 44
dictionary methods
clear method 43
copy method 43
del method 43
pop method 43

E

EBS volumes
attaching, to EC2 instance 429, 430
managing 426
managing, with boto3 module 427, 428
snapshots, creating 428
EC2 instances
connecting to 417, 418
connecting, with Netmiko 420, 422
connecting, with paramiko 419
managing 402
managing, with Python 403-417
Elastic Block Store (EBS) 426
Elastic Compute Cloud (EC2) 402
elif statement 48
email
logs, sending via 262-267
ether function 343

F

file handling 66
Excel files 73-75
open function, using for 66-69

OS module 69
Word files 70, 71
file transfer 200
backup configuration file, with SCP 232-234
backup configuration file, with SSH 201-203
Netmiko SCP connection, with concurrent module 224-227
with FTP connection 204-213
with Netmiko SCP connection 216-223
with Nornir SCP connection 228-231
with SFTP connection 213-216
File Transfer Protocol (FTP) 99, 200
firewalld service 323
activating 323, 324
configuring, on servers 331-339
drop zone 331
installing 323
public zone 331
trusted zone 331
using 324-329
for…else statement 57
for loops 48-50
ftplib module 204
ftpretty module 209
functions 85, 86
variables, calling from 88, 89
with default parameters 87
with parameters 86, 87

G

GNS3
downloading 96
installing 97
GNS3 VM
downloading 97
installing 97-99
Google Cloud Platform (GCP) 396

I

IAM user management 430, 431
Identity and Access Management (IAM) 430
if condition 45-48
input function 22, 26, 27
InquirerPy module 362, 363
confirm prompt 365
secret prompt 366
select prompt 364
text prompt 364
integer data type 31

interface bandwidth
plotting 243-245
Internet Control Message Protocol (ICMP) packet 344
internet service provider (ISP) networks 322
IP address validator 134, 135

J

JavaScript Object Notation (JSON) 179
Jinja2 template 157, 158
devices, configuring with 157, 162-173
if statement 174-178
rendering, with YAML file 160, 161

K

key pairs
creating 405-407

L

Linux
Python installation 9
Linux servers maintenance 283-287
CPU and memory levels, collecting 294-298
file type and permission, collecting 301-303
interface information, collecting 298-301
logs, collecting via syslog 288, 289
server login, with secure password 291-294
list 35-37
list methods
append method 37, 38
clear method 39
copy method 39, 40
count method 40, 41
del method 39
insert method 38
len method 40
pop method 38, 39
remove method 38
sum method 40
logs
sending via email 262-267

M

MacOS
Python installation 10, 11
main tool script
creating 369-377

Management Information Base (MIB) 257
matplotlib module 235
  advanced usage 239
  graphic, drawing with 238
  sample, drawing 237
modules
  creating 90
multithreading 118

N

Napalm module 178
  devices, configuring with 178, 184, 185
  logs, collecting from devices 179-183
nested loops 58, 59
Netmiko module
  for SSH 107, 108
  for telnet 116, 117
network automation 2
  benefits 3, 4
  future of networking 4
  tool design 366-368
network devices
  alert alarms 251-257
  configurations, replacing on files 154-156
  configuring 146-149
  configuring, by Nornir and Jinja template 194-196
  configuring, with Jinja 162-173
  configuring, with Napalm module 178-185
  configuring, with Nornir module 185, 186
  connecting, with Nornir-NAPALM 193, 194
  connecting, with Nornir-Netmiko 189-192
  interfaces, configuring 150-153
  reachability test 269
  upgrading 248-250
Network Management Systems (NMS) 3
  tools 235
network orchestration 2
network packets
  manipulating, with scapy 343-348
network security
  configurations, checking 348
  CPU levels, checking periodically with crontab service 348-353
  logs, checking 348
  network packets, manipulating with scapy 343-348
  packet collecting from ports, with Pyshark 357, 358
  port security configuration, checking in routers 356, 357
  router configuration, checking for insecure password 353-355
  security services, activating 322
Nornir 185, 186

inventory, configuring 186-189
Nornir and Jinja template
devices, configuring with 194-196
Nornir-NAPALM
devices, connecting with 193, 194
Nornir-Netmiko
devices, connecting with 189-192

O

Object Identifier (OID) 257
Object-Oriented Programming (OOPs) 91
open function
append mode 67, 68
create mode 69
for file handling 66
read by characters 68
read mode 66, 67
write mode 68
openpyxl module 73
Open Shortest Path First (OSPF) 257
OS module 69
getcwd function 70
listdir function 70
mkdir function 69
remove function 69
rmdir function 69

P

paramiko module 96
for SSH 99-101
print function 22-26
Pycharm 13
installation, on Windows 13, 14
pyplot function
functions and parameters 236
Pyshark
for collecting packets, from ports 357, 358
Python 5
characteristics 5
codes, running 11, 12
functions 85
installation 6
installation, for Linux 9
installation, for Mac 10, 11
installation, for Windows 6-8
tools and calculators 133
URL 5
usage area 5

Python modules
importing 14-17
installing 14

Q

Quality of Service (QoS) 322

R

range statement 55, 56
reachability test, to network devices
script, creating 269-272
traceroute test script, creating 272, 273
RE module 76, 96
findall function 77
search function 78
sets 83, 84
special sequences 80-83
split function 79
sub function 80

S

S3 bucket management 422-426
scapy module
network packets, manipulating with 343-348
Secure Copy Protocol (SCP) 200, 201
Secure File Transfer Protocol (SFTP) 99, 200
security services
activating 322
firewalld service 323-331
network devices access lists, creating 339-342
server configurations 305
file transfer, with paramiko 312-314
packages, installing 309-312
processes, stopping by script 315-319
server reboot concurrently 314, 315
users, creating 305-308
server environment
implementing 278
SSH connection, activating 281, 282
Ubuntu installation, on VMware 279-281
VMware player, downloading 278, 279
show function 344
Simple Network Management Protocol (SNMP) 257
logs, collecting with 257-262
Software Defined Networks (SDN) 3
srp function 343
SSH connection 99

activating 281, 282
configuration commands, running with paramiko 103, 104
multiple devices, connecting with Netmiko 111
multiple devices, connecting with paramiko 105, 106
Netmiko module, importing 107, 108
one device, connecting with Netmiko 108, 109
one device, connecting with paramiko 101-103
paramiko module, importing 99-101
string 30
using 30, 31
string methods
len method 32
lower method 33
replace() function 33, 34
split method 34
strip method 33
upper method 32
subnet calculator 136-140
subtask scripts
creating 377
network device scripts 378-385
others.py file 389-392
server scripts 386-389
summary function 344

T

telnet connection 112
multiple devices, connecting with telnetlib 115
Netmiko module for 116, 117
telnetlib module for 112-115
threading module 96
try except statement 59-62

U

Ubuntu 278
downloading 279
installing, on VMware 279-281
Long Time Support (LTS) version 279

V

VMware player
downloading 279
VMware Workstation Player
downloading 278, 279

W

while loop 51-53
Windows
  Python installation 6-8
Wireshark tool 346

Y

Yet Another Markup Language (YAML) 158, 159