Hands-On Enterprise Automation with Python

Automate common administrative and security tasks with Python

Packt
www.packt.com

By Bassem Aly

Hands-On Enterprise Automation with Python

Automate common administrative and security tasks with Python

Bassem Aly

Packt>

BIRMINGHAM - MUMBAI

Hands-On Enterprise Automation with Python

Commissioning Editor: Vijin Boricha
Acquisition Editor: Rohit Rajkumar
Content Development Editor: Ron Kurien
Technical Editor: Manish D Shanbhag
Copy Editor: Safis Editing
Project Coordinator: Judie Jose
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tom Scaria
Production Coordinator: Aparna Bhagat

First published: June 2018

Production reference: 1270618

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-851-2

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.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 Packt books and eBooks.

Contributors

About the author

Bassem Aly is an experienced SDN/NFV solution consultant at Juniper Networks and has been working in the telco industry for the last 9 years. He has focused on designing and implementing next-generation solutions by leveraging different automation and DevOps frameworks. Also, he has extensive experience of architecting and deploying telco applications over OpenStack. He also conducts corporate training on network automation and network programmability using Python and Ansible.

I would like to thank my amazing wife, Sarah, and my fantastic daughter, Mariam. They've sacrificed many nights and meals for this dream. I hope Mariam will read this book one day and understand why I spent so much time on the computer instead of "chasing". Thanks to my parents for their encouragement, which made me who I am today. Finally, thanks to my mentor, Ashraf Albasti, who has helped me in countless ways in my career.

About the reviewer

Jere Julian is a senior network automation engineer with nearly two decades of automation experience currently focused on workflow simplification through automation. The past few years have found him on the speaker circuit at DevOps Days and Interop ITX, as well as regularly contributing to network computing. He lives in NC with his wife and two boys and fights fire as a community volunteer as opposed to the data center. He can be contacted on Twitter at `@julianje`.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Table of Contents

Preface

The book starts by covering the set up of a Python environment to perform automation tasks, as well as the modules, libraries, and tools you will be using.

We'll explore examples of network automation tasks using simple Python programs and Ansible. Next, we will walk you through automating administration tasks with Python Fabric, where you will learn to perform server configuration and administration along with system administration tasks such as user management, database management, and process management. As you progress through this book, you'll automate several testing services with Python scripts and perform automation tasks on virtual machines and the cloud infrastructure with Python. In the concluding chapters, you will cover Python-based offensive security tools and learn to automate your security tasks.

By the end of this book, you will have mastered the skills of automating several system administration tasks with Python.

You can visit the author's blog at the following link: `https://basimaly.wordpress.com/`.

Who this book is for

Hands-On Enterprise Automation with Python is for system administrators and DevOps engineers who are looking for an alternative to major automation frameworks such as Puppet and Chef. Basic programming knowledge with Python and Linux shell scripting is necessary.

What this book covers

`Chapter 1`, *Setting Up Python Environment*, explores how to download and install the Python interpreter along with the Python Integrated Development Environment, called *JetBrains PyCharm.* The IDE provides you with smart autocompletion, intelligent code analysis, powerful refactoring and integrates with Git, virtualenv, Vagrant, and Docker. This will help you to write professional and robust Python code.

`Chapter 2`, *Common Libraries Used in Automation*, covers the Python libraries that are available today and that are used for automation. We will categorize them based on their usage (system, network, and cloud) and provide a simple introduction. As you progress through the book, you will find yourself deep diving into each of them and understanding their usage.

`Chapter 3`, *Setting up Your Network Lab Environment*, discusses the merits of network automation and how network operators use it today to automate the current devices. We will explore popular libraries that are used today to automate network nodes from Cisco, Juniper, and Arista. This chapter covers how to build a networking lab to apply the Python script on. We will use an open source network emulation tool called EVE-NG.

`Chapter 4`, *Using Python to Manage Network Devices*, dives into managing networking devices through telnet and SSH connections using netmiko, paramiko, and telnetlib. We will learn how to write the Python code to access switches and routers and execute commands on the terminal and then return the output. We will also learn how to utilize different Python techniques to back up and push configuration. The chapter ends with some use cases used today in modern network environment.

`Chapter 5`, *Extracting Useful Data from Network Devices*, explains how to use different tools and techniques inside Python to extract useful data from returned output and act upon it. Also, we will use a special library called *CiscoConfParse* to audit the configuration. Then we will learn how to visualize data to generate appealing graphs and reports with matplotlib.

`Chapter 6`, *Configuration Generator with Python and Jinja2*, explains how to generate a common configuration for a site with hundreds of network nodes. We will learn how to write a template and use it to generate a golden configuration with a templating language called Jinja2.

`Chapter 7`, *Parallel Execution of the Python Script*, covers how to instantiate and execute your Python code in parallel. This will allow us to finish the automation workflow faster as long as it is not interdependent.

`Chapter 8`, *Preparing a Lab Environment*, covers the installation process and preparation for our lab environment. We will install our automation server either in CentOS or Ubuntu over different hypervisors. Then we will learn how to automate the operating system installation with *Cobbler*.

`Chapter 9`, *Using the Subprocess Module*, explains how to send a command from a Python script directly to the operating system shell and investigate the returning output.

`Chapter 10`, *Running System Administration Tasks with Fabric*, introduces Fabric, which is a Python library for executing system administration tasks through SSH. Also, it's used in large deployment of applications. We will learn how to utilize and leverage this library to execute tasks on remote servers.

`Chapter 11`, *Generating System Reports*, *Managing Users, and System Monitoring*, explains that collecting data and generating recurring reports from the system is an essential task for any system administrator, and automating this task will help you to discover issues early and provide a solutions for them. In this chapter, we will see some proven ways to automate data collection from servers and generate formal reports. We will learn how to manage new and existing users using Python and Ansible. Also, we will dive into monitoring the system KPI and logs analysis. You can also schedule the monitoring scripts to run on a regular basis and send the result to your mail inbox.

`Chapter 12`, *Interacting with the Database*, states that if you're a database administrator or database developer, then Python provides a wide range of libraries and modules that cover managing and working on popular DBMSes such as MySQL, Postgress, and Oracle. In this chapter, we will learn how to interact with DBMSes using Python connectors.

`Chapter 13`, *Ansible for System Administration*, explores one of the most powerful tools in configuration management software. Ansible is very powerful when it comes to system administration and can be used to make sure the configuration is replicated exactly across hundreds or even thousands of servers at the same time.

`Chapter 14`, *Creating and Managing VMWare Virtual Machines*, explains how to automate VM creation on a VMWare hypervisor. We will discover different ways to create and manage virtual machines over ESXi using VMWare's official binding library.

`Chapter 15`, *Interacting with Openstack API*, explains that OpenStack was very popular in creating private IaaS when it came to private cloud. We will use Python modules such as requests to create REST calls and interact with OpenStack services such as nova, cinder, and neutron, and create the required resources over OpenStack. Later in the chapter, we will use Ansible playbooks for the same workflow.

`Chapter 16`, *Automating AWS with Python and Boto3*, covers how to automate common AWS services such as EC2 and S3 using official Amazon binindgs (BOTO3), which provides an easy-to-use API for services access.

`Chapter 17`, *Using the SCAPY Framework*, introduces SCAPY, which is a powerful Python tool used to build and craft packets and then send them on the wire. You can build any type of network stream and send it on the wire. It can also help you to capture network packets and replay them to the wire.

`Chapter 18`, *Building Network Scanner Using Python*, provides a complete example of building a network scanner using Python. You can scan a complete subnet for different protocols and ports and generate a report for each scanned host. Then, we will learn how to share the code with the open source community (GitHub) by leveraging Git.

To get the most out of this book

The reader should be acquainted with the basic programming paradigm of Python programming language and should have basic knowledge of Linux and Linux shell scripting.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Enterprise-Automation-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/HandsOnEnterpriseAutomationwithPython_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Some large packages such as `matplotlib` or `django` have hundreds of modules inside them, and developers usually categorize the related modules into a sub-directories."

A block of code is set as follows:

```
from netmiko import ConnectHandler
from devices import R1,SW1,SW2,SW3,SW4

nodes = [R1,SW1,SW2,SW3,SW4]

for device in nodes:
    net_connect = ConnectHandler(**device)
    output = net_connect.send_command("show run")
    print output
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
hostname {{hostname}}
```

Any command-line input or output is written as follows:

```
pip install jinja2
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example:

"Choose your platform from the **Download** page, and either the x86 or x64 version."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 `copyright@packtpub.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 `authors.packtpub.com`.

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Setting Up Our Python Environment

In this chapter, we will provide a brief introduction to the Python programming language and the differences between the current versions. Python ships in two active versions, and making a decision on which one to use during development is important. In this chapter, we will download and install Python binaries into the operating system.

At the end of the chapter, we will install one of the most advanced **Integrated Development Editors** (**IDEs**) used by professional developers around the world: PyCharm. PyCharm provides smart code completion, code inspections, on-the-fly error highlighting and quick fixes, automated code refactoring, and rich navigation capabilities, which we will go over throughout this book, as we write and develop Python code.

The following topics will be covered in this chapter:

- An introduction to Python
- Installing the PyCharm IDE
- Exploring some nifty PyCharm features

An introduction to Python

Python is a high-level programming language that provides a friendly syntax; it is easy to learn and use, for both beginner and expert programmers.

Python was originally developed by Guido van Rossum in 1991; it depends on a mix of C, C++, and other Unix shell tools. Python is known as a language for general purpose programming, and today it's used in many fields, such as software development, web development, network automation, system administration, and scientific fields. Thanks to its large number of modules available for download, covering many fields, Python can cut development time down to a minimum.

The Python syntax was designed to be readable; it has some similarities to the English language, while the code construction itself is beautiful. Python core developers provide 20 informational rules, called the Zen of Python, that influenced the design of the Python language; most of them involve building clean, organized, and readable code. The following are some of the rules:

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

You can read more about the Zen of Python at `https://www.python.org/dev/peps/pep-0020/`.

Python versions

Python comes with two major versions: Python 2.x and Python 3.x. There are subtle differences between the two versions; the most obvious is the way their `print` functions treat multiple strings. Also, all new features will only be added to 3.x, while 2.x will receive security updates before full retirement. This won't be an easy migration, as many applications are built on Python 2.x.

Why are there two active versions?

I will quote the reason from the official Python website:

> *"Guido van Rossum (the original creator of the Python language) decided to clean up Python 2.x properly, with less regard for backwards compatibility than is the case for new releases in the 2.x range. The most drastic improvement is the better Unicode support (with all text strings being Unicode by default) as well as saner bytes/Unicode separation.*
>
> *"Besides, several aspects of the core language (such as print and exec being statements, integers using floor division) have been adjusted to be easier for newcomers to learn and to be more consistent with the rest of the language, and old cruft has been removed (for example, all classes are now new-style, "range()" returns a memory efficient iterable, not a list as in 2.x)."*

You can read more about this topic at `https://wiki.python.org/moin/Python2orPython3`.

Should you only learn Python 3?

It depends. Learning Python 3 will future-proof your code, and you will use up-to-date features from the developers. However, note that some third-party modules and frameworks lack support for Python 3 and will continue to do so for the near future, until they completely port their libraries to Python 3.

Also, note that some network vendors, such as Cisco, provide limited support for Python 3.x, as most of the required features are already covered in Python 2.x releases. For example, the following are the supported Python versions for Cisco devices; you will see that all devices support 2.x, not 3.x:

WHICH VERSION OF PYTHON DOES YOUR DEVICE SUPPORT?

	CAT 3650	CAT 3850	ISR 4K	Nexus 3K/9K	Nexus 5K/6K	Nexus 7K
Python 2.7	IOS-XE 16.5.1	IOS-XE 16.5.1	IOS-XE 16.5.1	N3K NX-OS 6.0 N9K NX-OS 7.0	N5K NX-OS 5.2 N6K NX-OS 6.0	NX-OS 6.1
Python 3.x	N/A	N/A	IOS-XE 16.5.1			

Source: `https://developer.cisco.com/site/python/`

Does this mean I can't write code that runs on both Python 2 and Python 3?

No, you can, of course, write your code in Python 2.x and make it compatible with both versions, but you will need to import a few libraries first, such as the `__future__` module, to make it backward compatible. This module contains some functions that tweak the Python 2.x behavior and make it exactly like Python 3.x. Take a look at the following examples to understand the differences between the two versions:

```
#python 2 only
print "Welcome to Enterprise Automation"
```

The following code is for Python 2 and 3:

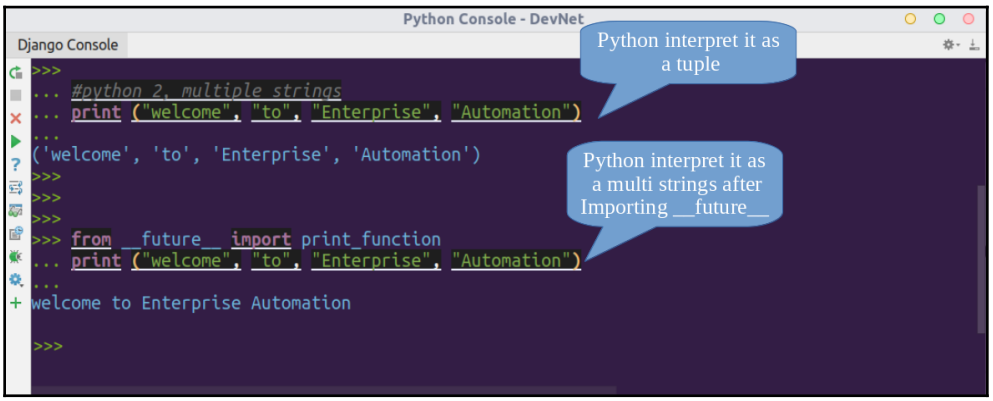
```
# python 2 and 3
print("Welcome to Enterprise Automation")
```

Now, if you need to print multiple strings, the Python 2 syntax will be as follows:

```
# python 2, multiple strings
print "welcome", "to", "Enterprise", "Automation"

# python 3, multiple strings
print ("welcome", "to", "Enterprise", "Automation")
```

If you try to use parentheses to print multiple strings in Python 2, it will interpret it as a tuple, which is wrong. For that reason, we will import the `__future__` module at the beginning of our code, to prevent that behavior and instruct Python to print multiple strings.

The output will be as follows:

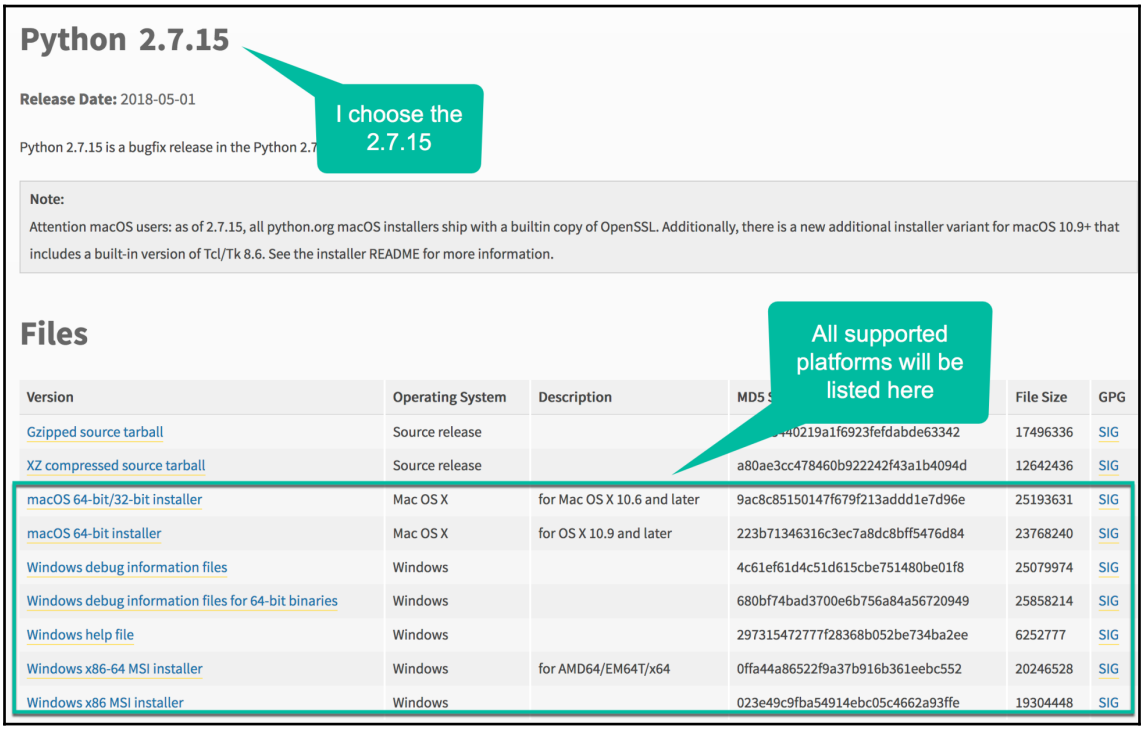


Python installation

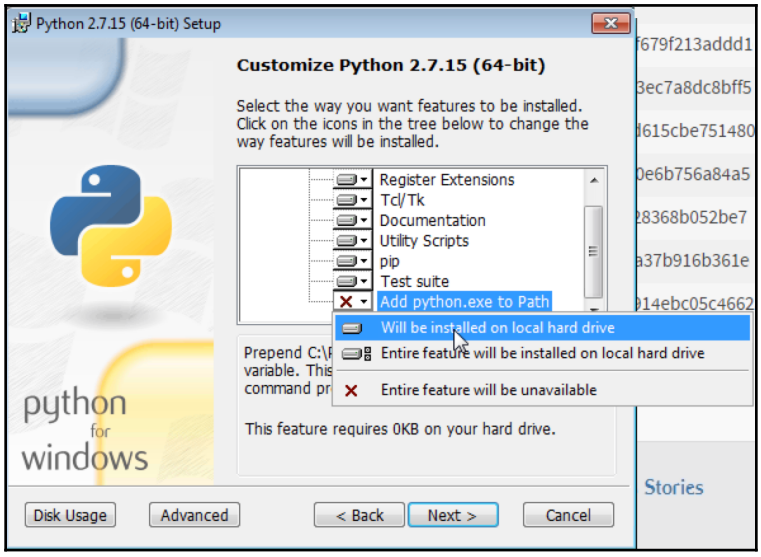
Whether you choose to go with a popular Python version (2.x) or build future-proof code with Python 3.x, you will need to download the Python binaries from the official website and install them in your operating system. Python provides support for different platforms (Windows, Mac, Linux, Raspberry PI, and so on):

1. Go to `https://www.python.org/downloads/` and choose the latest version of either 2.x or 3.x:

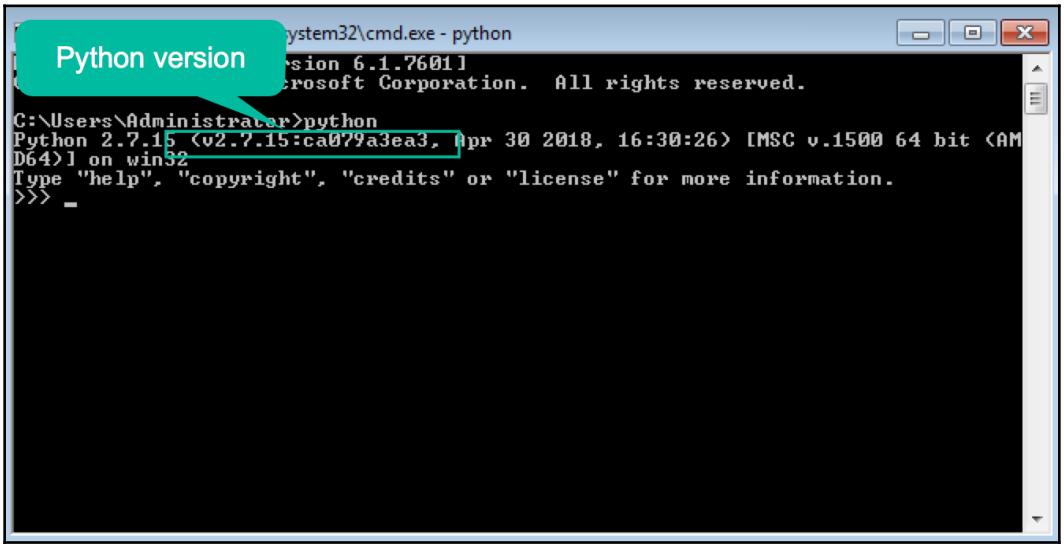
2. Choose your platform from the **Download** page, and either the x86 or x64 version:



3. Install the package as usual. It's important to select the **Add python to the path** option during the installation, in order to access Python from the command line (in the case of Windows). Otherwise, Windows won't recognize the Python commands and will throw an error:



4. Verify that the installation is complete by opening the command line or terminal in your operating system and typing `python`. This should access the Python console and provide a verification that Python has successfully installed on your system:



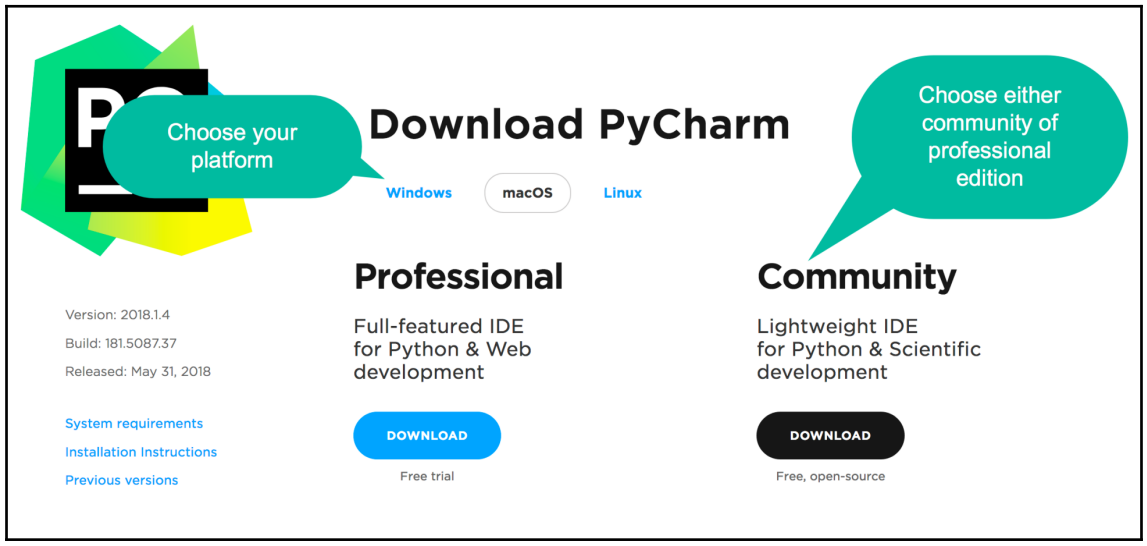
Installing the PyCharm IDE

PyCharm is a fully fledged IDE, used by many developers around the world to write and develop Python code. The IDE is developed by the Jetbrains company and provides rich code analysis and completion, syntax highlighting, unit testing, code coverage, error discovery, and other Python linting operations.

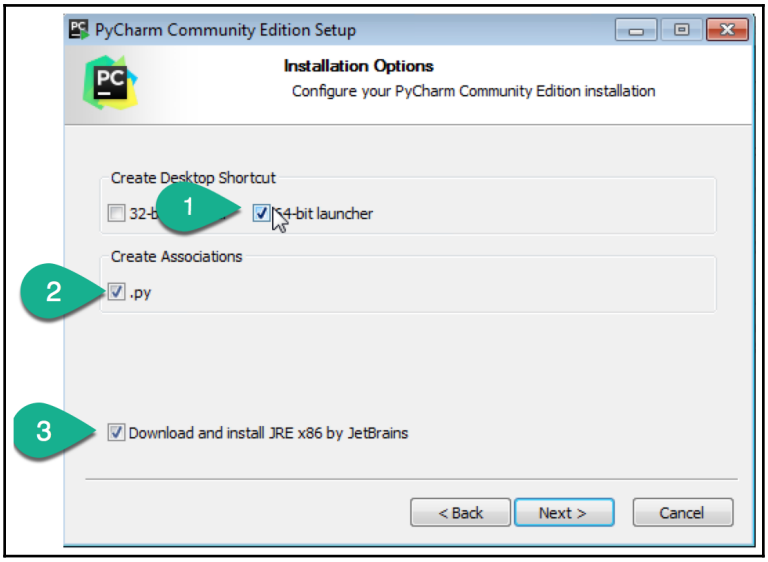
Also, PyCharm Professional Edition supports Python web frameworks, such as Django, web2py, and Flask, beside integrations with Docker and vagrant for running a code over them. It provides amazing integration with multiple version control systems, such as Git (and GitHub), CVS, and subversion.

In the next few steps, we will install PyCharm Community Edition:

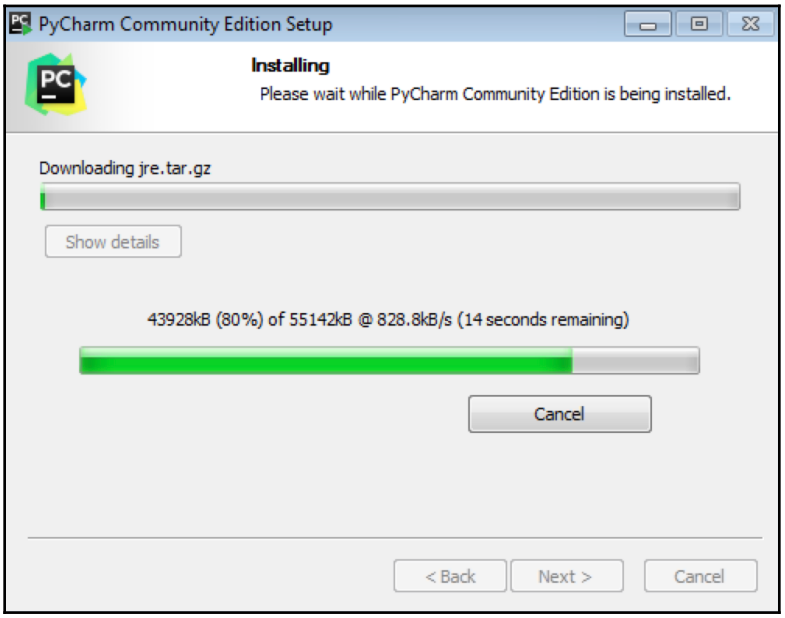
1. Go to the PyCharm download page (`https://www.jetbrains.com/pycharm/download/`) and choose your platform. Also, choose to download either the Community Edition (free forever) or the Professional Edition (the Community version is completely fine for running the codes in this book):



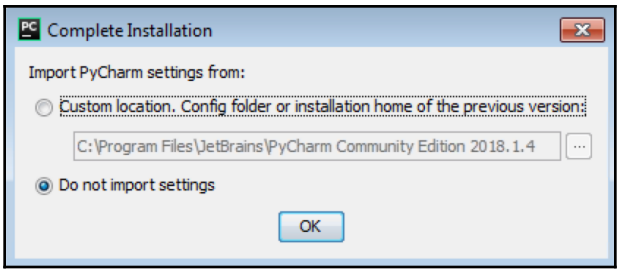
2. Install the software as usual, but make sure that you select the following options:
    - **32-** or **64-bit** launcher (depending on your operating system).
    - **Create Associations** (this will make PyCharm the default application for Python files).
    - **Download and install JRE x86 by JetBrains**:



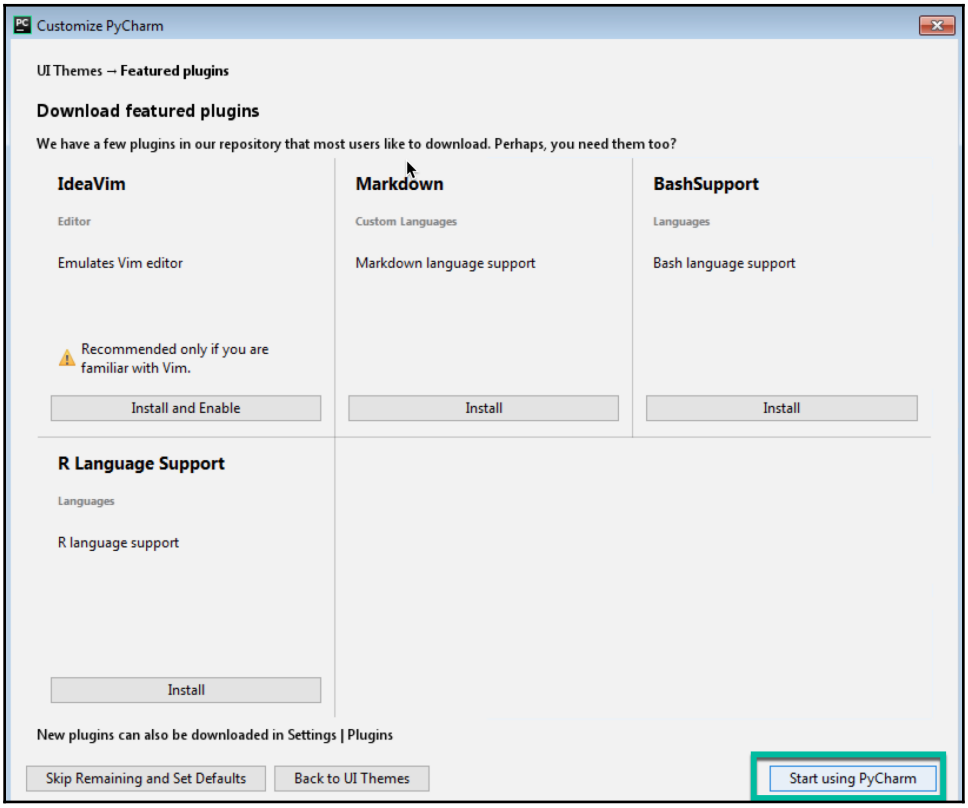
3. Wait until PyCharm downloads the additional packages from the internet, and installs it, then choose **Run PyCharm Community Edition**:



4. Since this is a new and fresh installation, we won't import any settings from



5. Select the desired UI theme (either the **default** or **darcula**, for dark mode). You can install some additional plugins, such as **Markdown** and **BashSupport,** which will make PyCharm recognize and support those languages. When you finish, click on **Start Using PyCharm**:

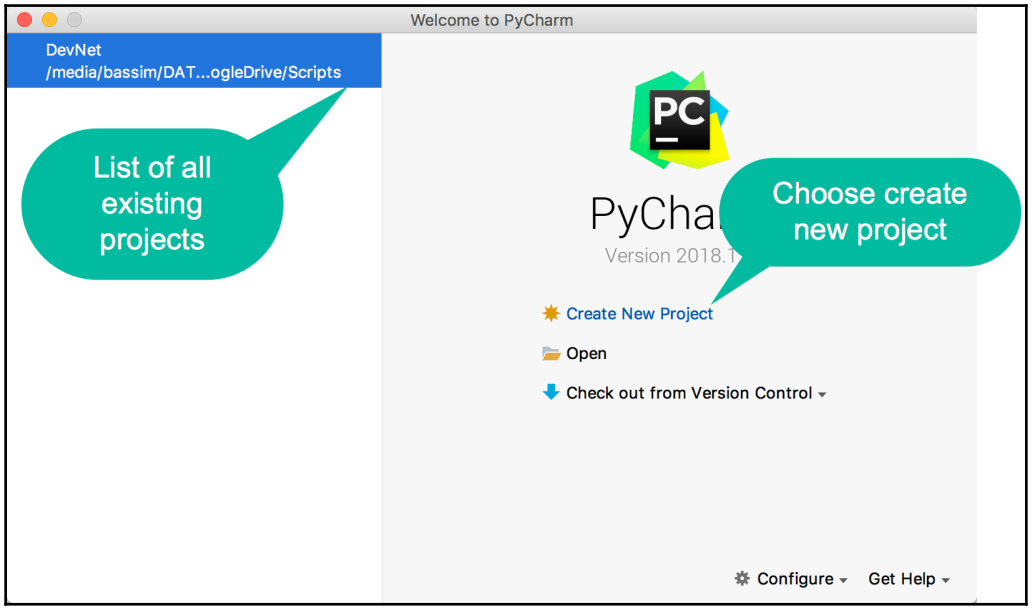


Setting up a Python project inside PyCharm

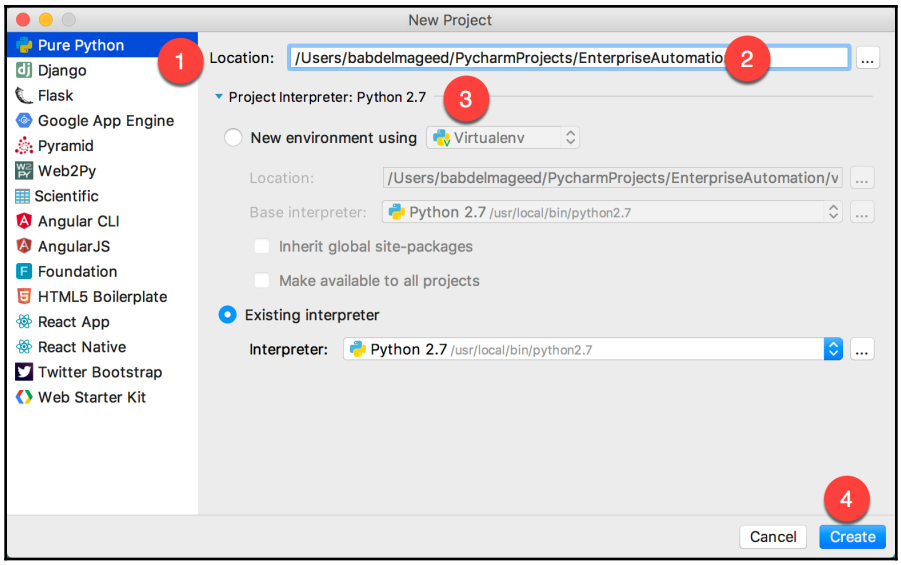
Inside PyCharm, a Python project is a collection of Python files that you have developed and Python modules that are either built in or were installed from a third party. You will need to create a new project and save it to a specific location inside your machine before starting to develop your code. Also, you will need to choose the default interpreter for this project. By default, PyCharm will scan the default location on the system and search for the Python interpreter. The other option is to create a completely isolated environment, using Python `virtualenv`. The basic problem with the `virtualenv` address is its package dependencies. Let's assume that you're working on multiple different Python projects, and one of them needs a specific version of *x* package. On the other hand, one of the other projects needs a completely different version from the same package. Notice that all installed Python packages go to `/usr/lib/python2.7/site-packages`, and you can't store different versions of the same package. The `virtualenv` will solve this problem by creating an environment that has its own installation directories and its own package; each time you work on either of the two projects, PyCharm (with the help of `virtualenv`) will activate the corresponding environment to avoid any conflict between packages.

Follow these steps to set up the project:

1. Choose **Create New Project**:

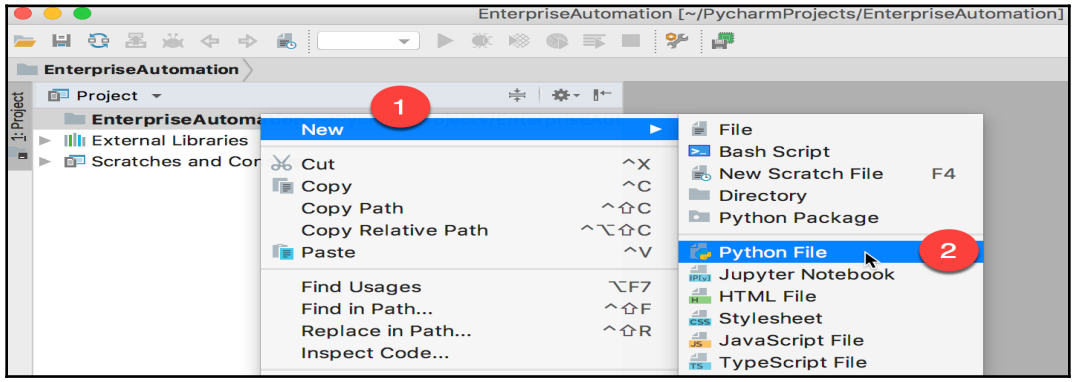


2. Choose the project settings:



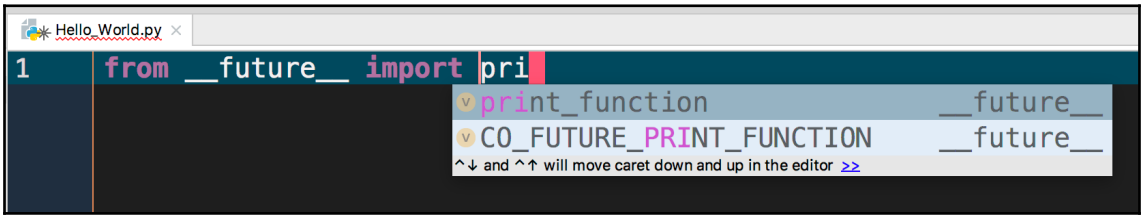
1. Select the type of project; in our case, it will be **Pure Python**.
2. Choose the project's location on the local hard drive.
3. Choose the Project Interpreter. Either use the existing Python installation in the default directory, or create a new virtual environment tied specifically to that project.
4. Click on **Create**.

3. Create a new **Python File** inside the project:

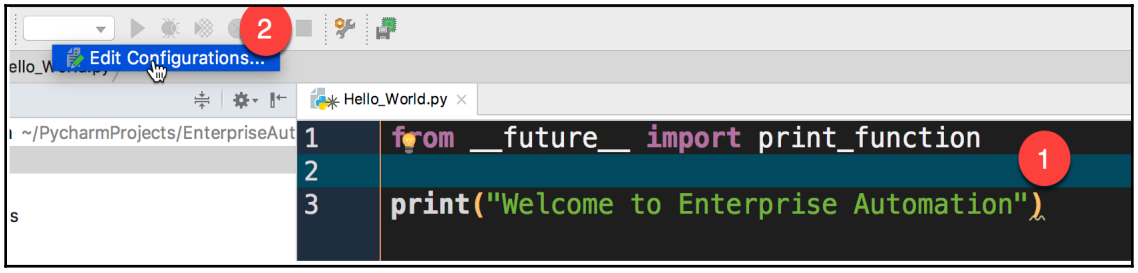


1. Right-click on the project name and select **New**.
2. Choose **Python File** from the menu, then choose a filename.

A new, blank file is opened, and you can write a Python code directly into it. Try to import the `__future__` module, for example, and PyCharm will automatically open a pop-up window with all possible completions available as shown in the following screenshot:

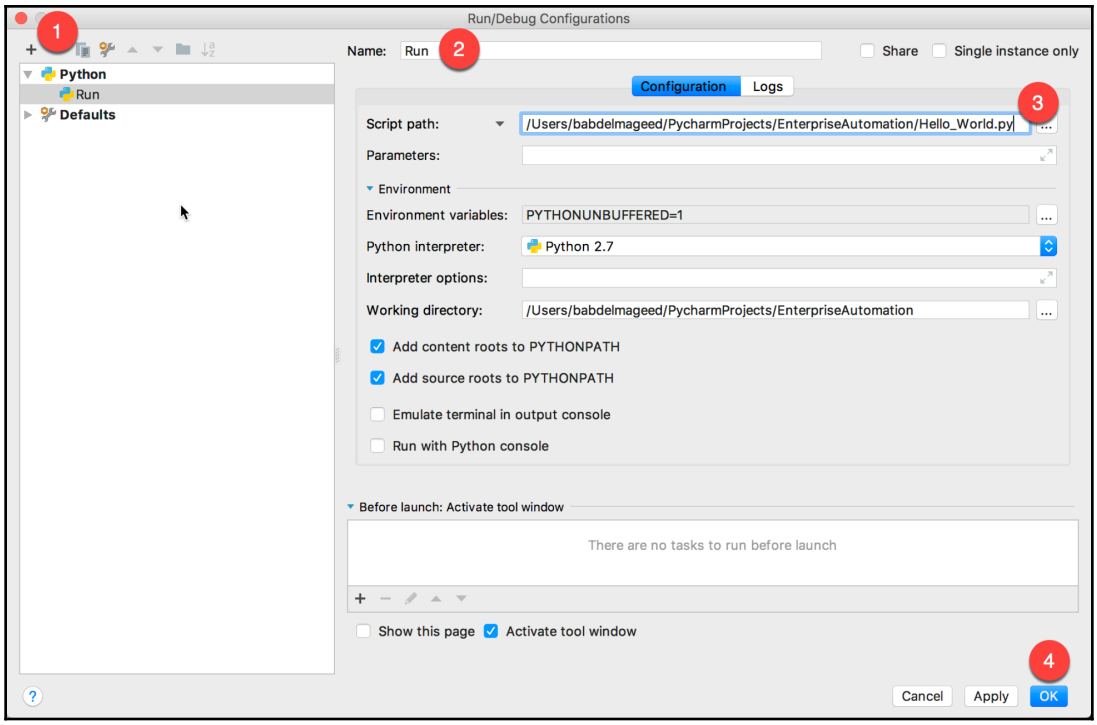


4. Run your code:



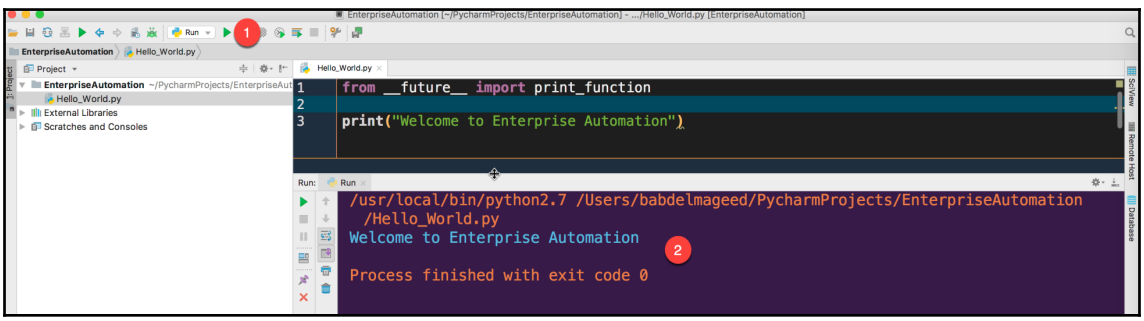
1. Enter the code that you wish to run.
2. Choose **Edit Configuration** to configure the runtime settings for the Python file.

5. Configure new Python settings for running your file:



1. Click on the + sign to add a new configuration, and choose **Python**.
2. Choose the configuration name.
3. Choose the script path inside your project.
4. Click on **OK**.

6. Run the code:



1. Click on the **play** button beside the configuration name.
2. PyCharm will execute the code inside the file specified in the configuration, and will return the output to the terminal.

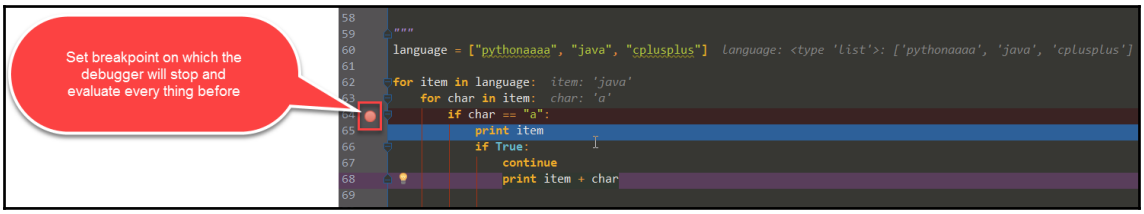
Exploring some nifty PyCharm features

In this section, we will explore some of PyCharm's features. PyCharm has a huge collection of tools out of the box, including an integrated debugger and test runner, Python profiler, a built-in Terminal, integration with major VCS and built-in database tools, remote development capabilities with remote interpreters, an integrated SSH Terminal, and integration with Docker and Vagrant. For a list of other features, please check the official site (`https://www.jetbrains.com/pycharm/features/`).

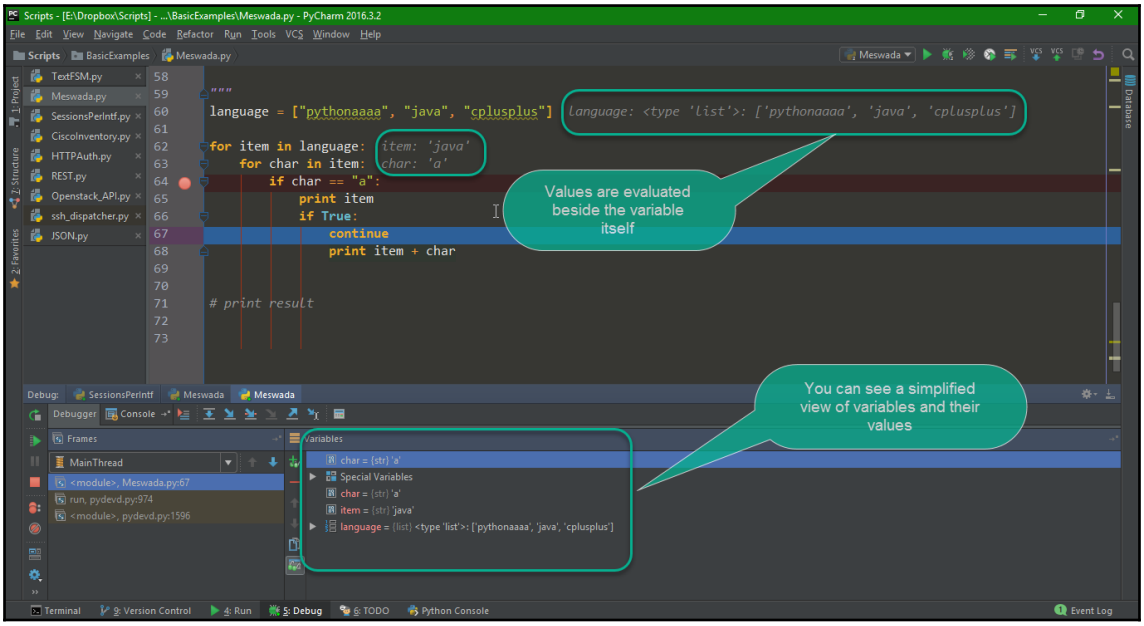
Code debugging

Code debugging is a process that can help you to understand the cause of an error, by providing an input to the code and walking through each line of the code and seeing how it evaluates at the end. The Python language contains some debugging tools to get insights from the code, starting with a simple `print` function, assert command till a complete unit testing for the code. PyCharm provides an easy way to debug the code and see the evaluated values.

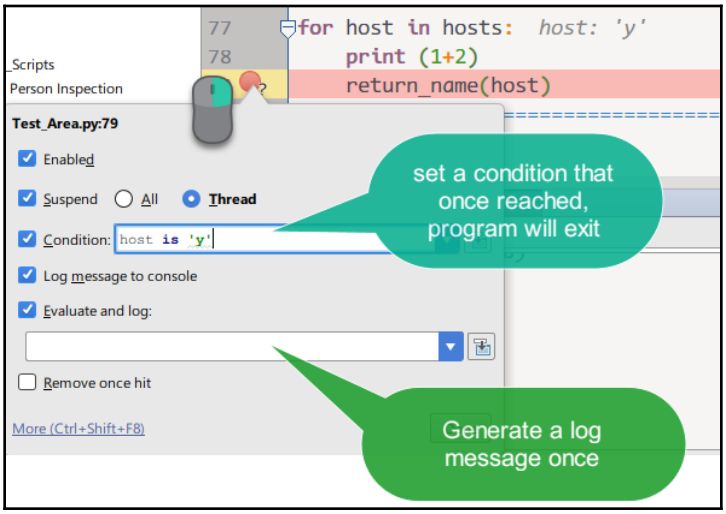
To debug code in PyCharm (say, a nested `for` loop with `if` clauses), you need to set a breakpoint on the line at which you want PyCharm to stop the program execution. When PyCharm hits this line, it will pause the program and dump the memory to see the contents of each variable:



Notice that the value of each variable is printed besides it, on the first iteration:



Also, you can right-click on the breakpoint and add a specific condition for any variable. If the variable is evaluated to a specific value, then a log message will be printed:



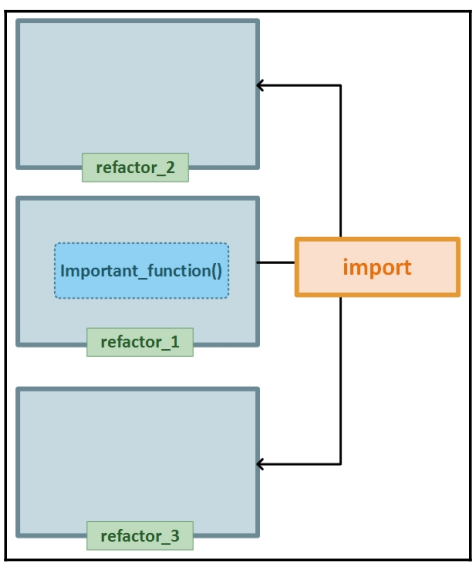
Code refactoring

Refactoring the code is the process of changing the structure of a specific variable name inside your code. For example, you may choose a name for your variable and use it for a project that consists of multiple source files, then later decide to rename the variable to something more descriptive. PyCharm provides many refactoring techniques, to make sure that the code can be updated without breaking the operation.

PyCharm does the following:

- The refactoring itself
- Scans every file inside the project and makes sure that the references to the variables are updated
- If something can't be updated automatically, it will give you a warning and open a menu, so you can decide what to do
- Saves the code before refactoring it, so you can revert it later

Let's look at an example. Assume that we have three Python files in our project, called `refactor_1.py`, `refactor_2.py`, and `refactor_3.py`. The first file contains `important_funtion(x)`, which is also used in both `refactor_2.py` and `refactor_3.py`.



Copy the following code in a `refactor_1.py` file:

```
def important_function(x):
    print(x)
```

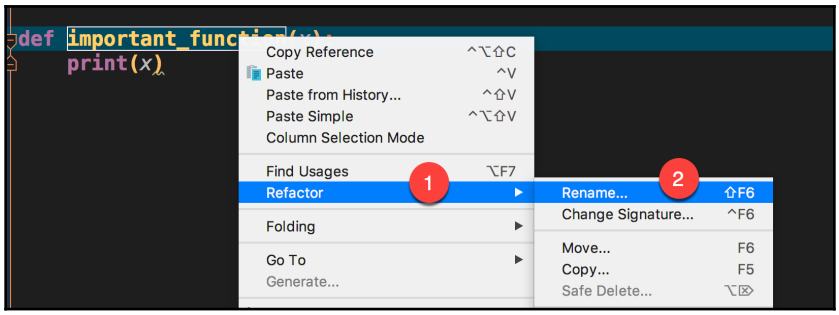
Copy the following code in a `refactor_2.py` file:

```
from refactor_1 import important_function
important_function(2)
```

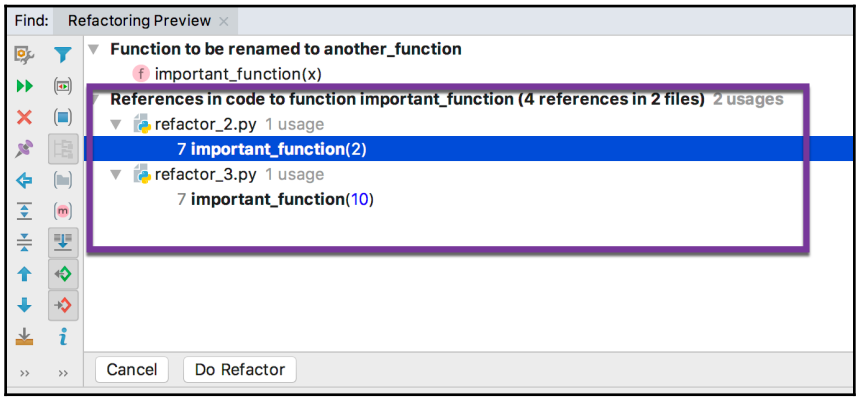
Copy the following code in a `refactor_3.py` file:

```
from refactor_1 import important_function
important_function(10)
```

To perform the refactoring, you need to right-click on the method itself, select **Refactor** | **Rename**, and enter the new name for the method:



Notice that a window opens at the bottom of the IDE, listing all references of this function, the current value for each one, and which file will be affected after the refactoring:

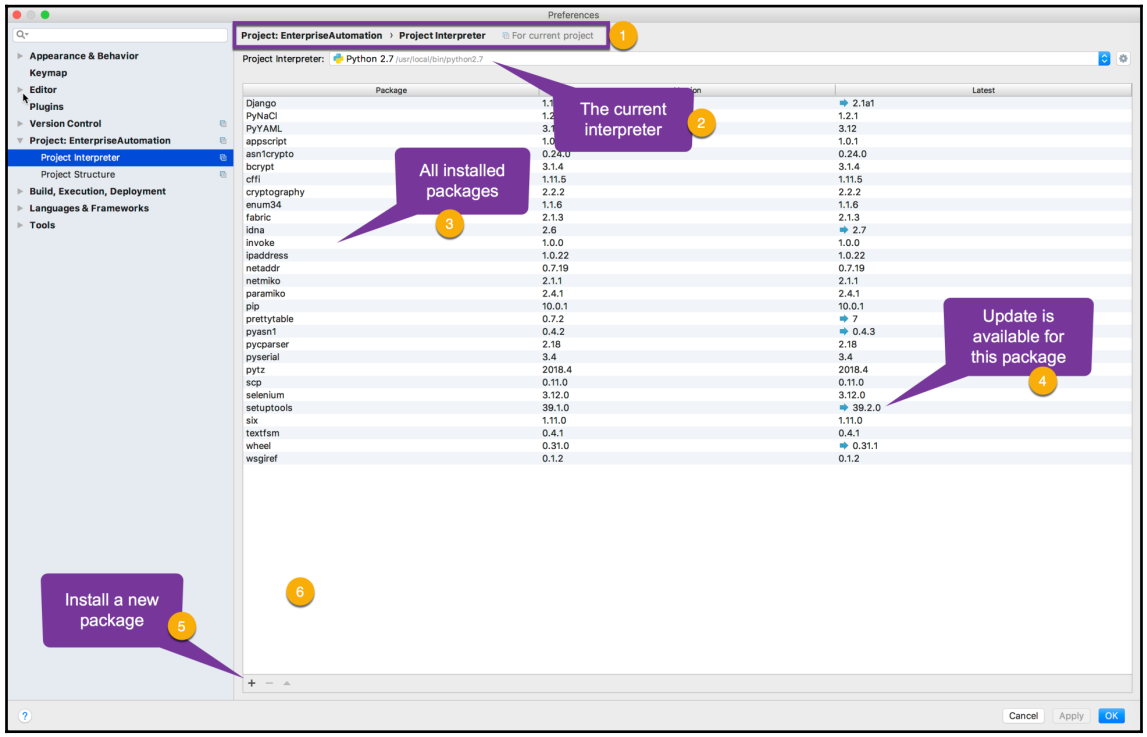


If you choose **Do Refactor**, all of the references will be updated with the new name, and your code will not be broken.

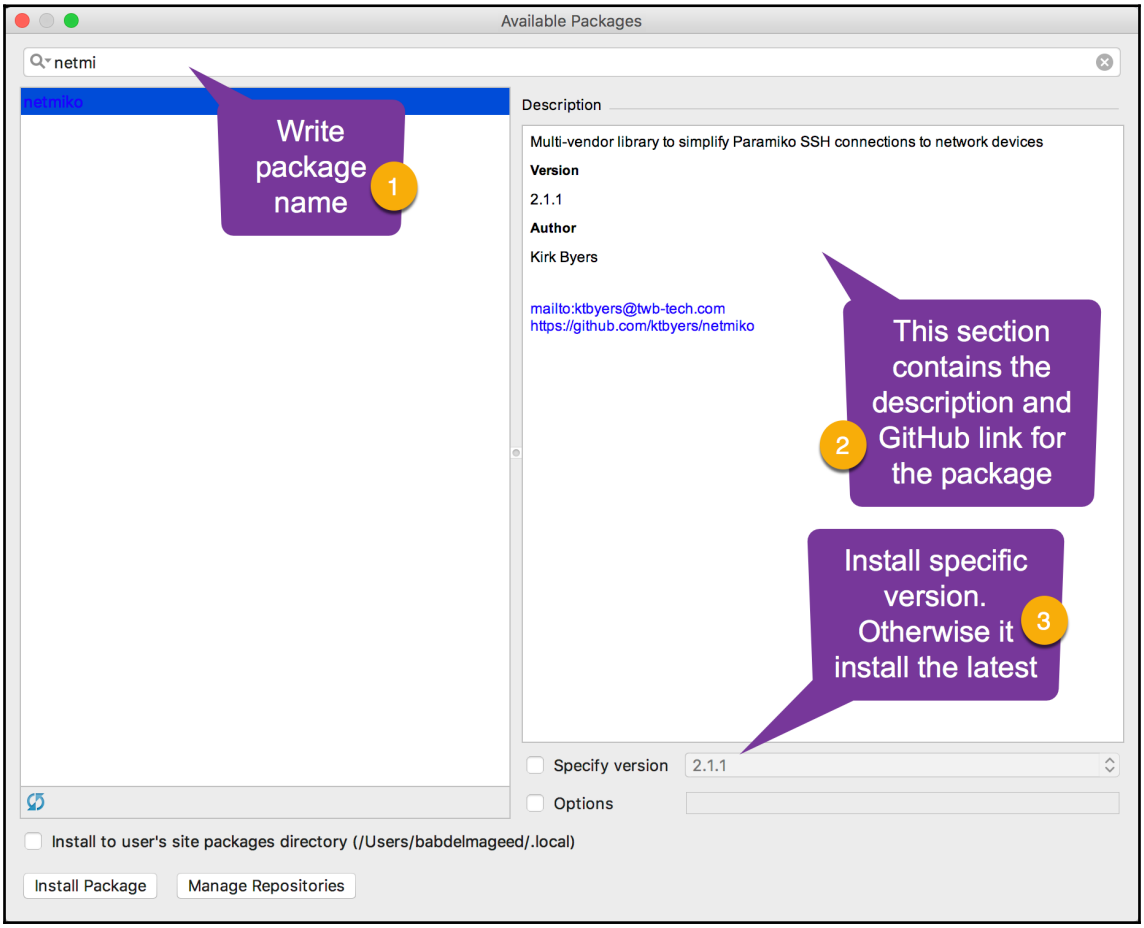
Installing packages from the GUI

PyCharm can be used to install packages for existing interpreters (or the `virtualenv`) using the GUI. Also, you can see a list of all installed packages, and whether upgrades are available for them.

First, you need to go to **File | Settings | Project | Project Interpreter:**



As shown in the preceding screenshot, PyCharm provides a list of installed packages and their current versions. You can click on the + sign to add a new package to the project interpreter, then enter the package initials into the search box:



You should see a list of available packages, containing a name and description for each one. Also, you can specify a specific version to be installed on your interpreter. Once you have clicked on **Install Package**, PyCharm will execute a `pip` command on your system (and may ask you for a permission); then, it will download the package onto the installation directory and execute the `setup.py` file.

Summary

In this chapter, you learned the differences between Python 2 and Python 3, and how to decide which one to use, based on your needs. Also, you learned how to install a Python interpreter and how to use PyCharm as an advanced editor to write and manage your code's life cycle.

In the next chapter, we will discuss the Python package structure and the common Python packages used in automation.

2
Common Libraries Used in Automation

This chapter will walk you through how Python packages are structured and the common libraries used today to automate the system and network infrastructure. There's a long growing list of Python packages that cover network automation, system administration, and managing public and private clouds.

Also, it's important to understand how to access the module source code and how the small pieces inside the Python package are related to each other so we can modify the code, add or remove features, and share the code again with the community.

The following topics will be covered in this chapter:

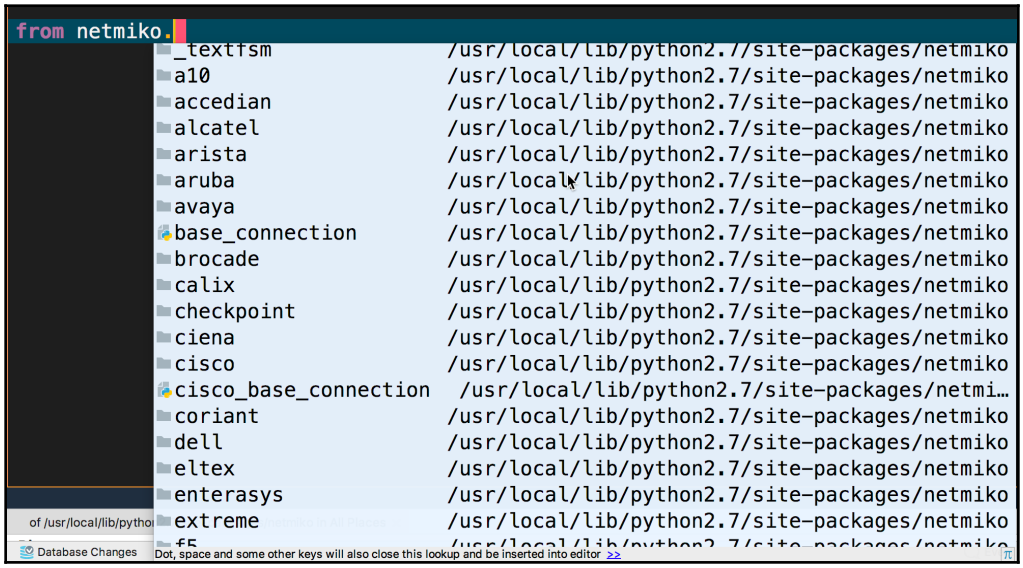
- Understanding Python packages
- Common Python libraries
- Accessing module source code

Understanding Python packages

Python core code is actually small by design to maintain simplicity. Most functionalities will be through adding third-party packages and modules.

Module is a Python file that contains functions, statements, and classes that will be used inside your code. The first thing to do is `import` the module then start to use its functions.

On other hand, a **package** collects related modules connected to each other and puts them in a single hierarchy. Some large packages such as `matplotlib` or `django` have hundreds of modules inside them, and developers usually categorize the related modules into a sub-directories. For example, the `netmiko` package contains multiple sub-directories and each one contains modules to connect to network devices from different vendors:



Doing that gives the package maintainer the flexibility to add or remove features from each module without breaking the global package operation.

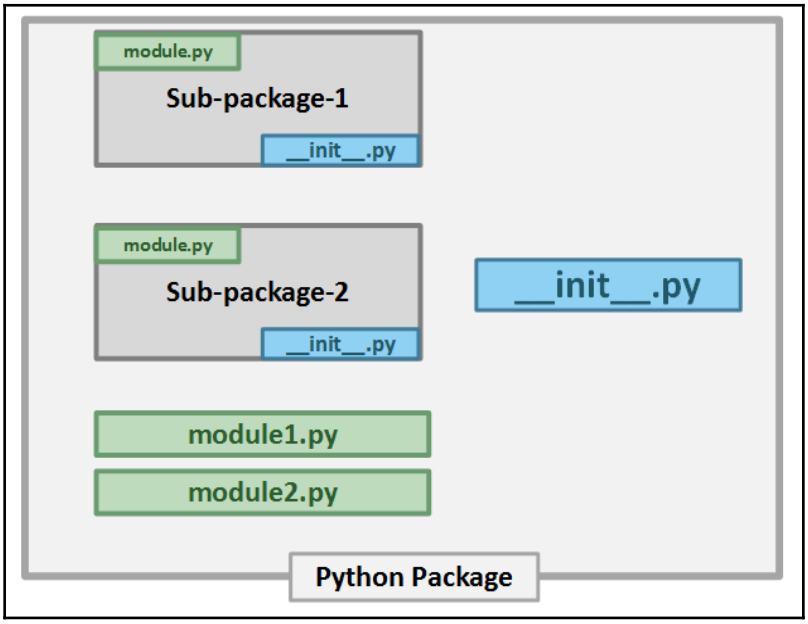
Package search paths

Typically, Python searches for modules in some specific system paths. You can print these paths by importing the `sys` module and printing the `sys.path`. This will actually return the strings inside the `PYTHONPATH` environment variable and inside the operating system; notice the result is just a normal Python list. You can add more paths to the search scope using a list function such as `insert()`.

However, it's better to install the packages in the default search paths so the code won't break when you share it with other developers:

```
bassim:~$ python
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu', '/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/
lib-old', '/usr/lib/python2.7/lib-dynload', '/home/bassim/.local/lib/python2.7/site-packages', '/usr/local/lib/python2.7
/dist-packages', '/usr/local/lib/python2.7/dist-packages/pycontrail-2.20b64-py2.7.egg', '/usr/lib/python2.7/dist-package
s']
>>>
```

A simple package structure with a single module will be something like this:



The `__init__` file inside each package (in the global directory or in the sub-directory) will tell the Python interpreter that this directory is a Python package, and each file ending with `.py` will be a module file, which could be imported inside your code. The second function of the `init` file is to execute any code inside it once the package is imported. However, most developers leave it empty and just use it to mark the directory as a Python package.

Common Python libraries

In the next sections, we will explore the common Python libraries used for network, system, and cloud automation.

Network Python Libraries

Network environments nowadays contain multiple devices from many vendors, and each device plays a different role. Design and automation frameworks for network devices are essential to network engineers in order to automate repeated tasks and enhance the way they usually do their job, while reducing human errors. Large enterprises and service providers usually tend to design a workflow that can automate different network tasks and improve network resiliency and agility. The workflow contains a series of related tasks that together form a process or a workflow that will be executed when there's a change needed on the network.

Some of the tasks that could be performed by a network automation framework without human intervention are:

- Root cause analysis for the problem
- Checking and updating the device operating system
- Discovering the topology and relationships between nodes
- Security audits and compliance reporting
- Installing and withdrawing routes from the network device based on the application needs
- Managing device configuration and rollback

Here are some Python libraries that are used to automate network devices:

Network Library	Description	Link
Netmiko	A multi-vendor library that supports SSHing and Telnet for network devices and executes commands on it. Support includes Cisco, Arista, Juniper, HP, Ciena, and many other vendors.	`https://github.com/ktbyers/netmiko`
NAPALM	A Python library that works as a wrapper for the official Vendor API. It provides abstraction methods that connect to devices from multiple vendors and extract information from it while returning the output in an structured format. This can be easily processed by software.	`https://github.com/napalm-automation/napalm`

PyEZ	A Python library used to manage and automate Juniper devices. It can perform CRUD operation on the device from the Python client. Also, it can retrieve facts about the device such as the management IP, serial number, and version. The returned output will be in JSON or XML format.	`https://github.com/Juniper/py-junos-eznc`
infoblox-client	A Python client used to interact with infoblox NIOS over the interface, based on a REST called WAPI.	`https://github.com/infobloxopen/infoblox-client`
NX-API	A Cisco Nexus (some platforms only) series API that exposes the CLI through HTTP and HTTPS. You can enter a show command in the provided sandbox portal and it will be converted to an API call to the device and will return the output in JSON and XML format.	`https://developer.cisco.com/docs/nx-os/#!working-with-nx-api-cli`
pyeapi	A Python library that acts as a wrapper around the Arista EOS eAPI and is used to configure Arista EOS devices. The library supports eAPI calls over HTTP and HTTPs.	`https://github.com/arista-eosplus/pyeapi`
netaddr	A Python library for working with network addresses such as IPv4, IPv6, and layer 2 addresses (MAC addresses). It can iterate, slice, sort, and summarize the IP block.	`https://github.com/drkjam/netaddr`
ciscoconfparse	A Python library that is able to parse a Cisco IOS-style configuration and returns the output in a structured format. The library also provides support for device configuration based on brace-delimited configurations such as Juniper and F5.	`https://github.com/mpenning/ciscoconfparse`
NSoT	A database for tracking the inventory and metadata of network devices. It provides a frontend GUI based on Python Django. The backend is based on SQLite database where the data is stored. Also, it provides the API interface for the inventory using pynsot bindings.	`https://github.com/dropbox/nsot`
Nornir	A new automation framework based on Python and consumed directly from Python code without a need to have custom **DSL** (**Domain Specific Language**). The Python code is called runbook and contains a set of tasks that can run against the devices stored in the inventory (supports also Ansible inventory format). The tasks can utilize other libraries (such as NAPALM) to get information or configure the devices.	`https://github.com/nornir-automation/nornir`

System and cloud Python libraries

Here are some of the python packages that can be used for both system and cloud administration. Public cloud providers such as **Amazon Web Services** (**AWS**) and Google tend to provide open and standard access to their resources in order to be easily integrated with the organization DevOps model. Phases like continuous integration, testing, and deployment require *continuous* access to infrastructure (either virtualized or bare metal servers) in order to complete the code life cycle. This can't be done manually and needs to be automated:

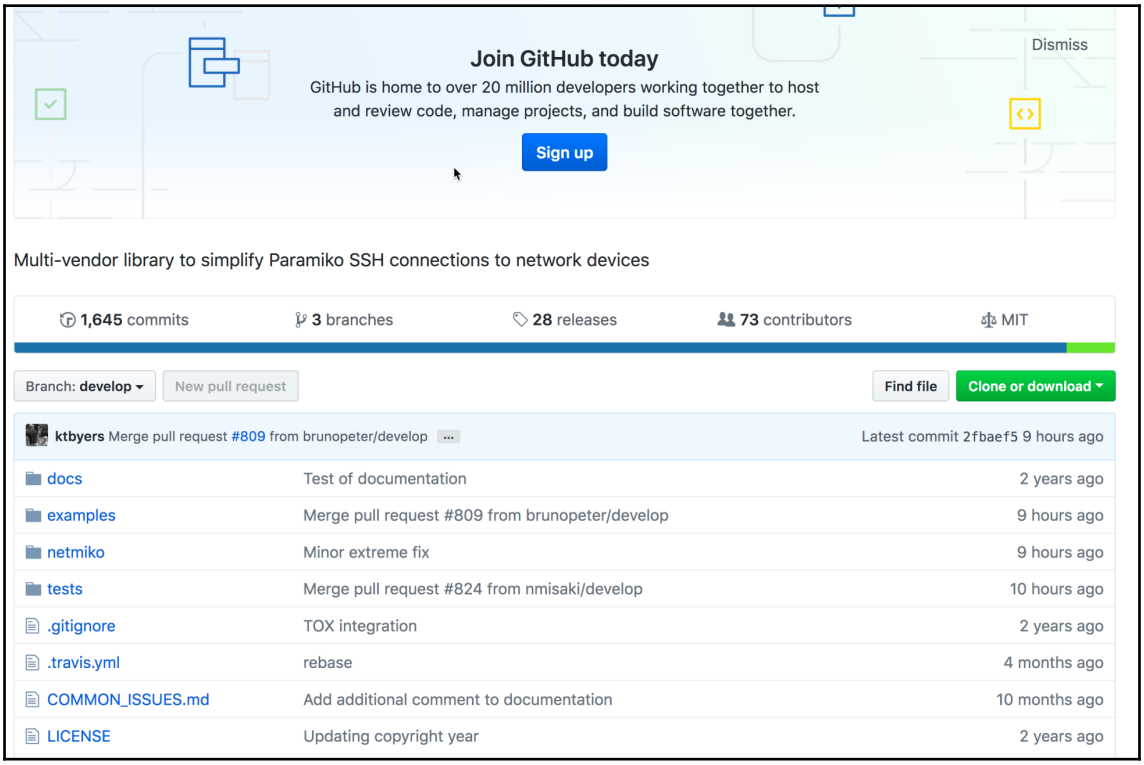
Library	Description	Link
ConfigParser	Python standard library to parse and work with the INI files.	`https://github.com/python/cpython/blob/master/Lib/configparser.py`
Paramiko	Paramiko is a Python (2.7, 3.4+) implementation of the SSHv2 protocol, providing both client and server functionality.	`https://github.com/paramiko/paramiko`
Pandas	A library providing high-performance, easy-to-use data structures and data analysis tools.	`https://github.com/pandas-dev/pandas`
`boto3`	Offifical Python interface that manages different AWS operations, such as creating EC2 instances and S3 storage.	`https://github.com/boto/boto3`
`google-api-python-client`	Google official API client library for Google Cloud Platform.	`https://github.com/google/google-api-python-client`
`pyVmomi`	The official Python SDK from VMWare that manages ESXi and vCenter.	`https://github.com/vmware/pyvmomi`
PyMYSQL	A pure python MySQL driver to work with MySQL DBMS.	`https://github.com/PyMySQL/PyMySQL`

Psycopg	The PostgresSQL adapter for python which conforms to DP-API 2.0 standard.	`http://initd.org/psycopg/`
Django	A high-level open source web framework based on Python. The framework follows the **MVT** (**Model, View, and Template**) architecture design for building web applications without the hassle of web development and common security mistakes.	`https://www.djangoproject.com/`
Fabric	A simple Python tool for executing commands and software deployments on remote devices based on SSH.	`https://github.com/fabric/fabric`
SCAPY	A brilliant Python-based packet manipulation that is able to handle a wide range of protocols and can build packets with any combination of network layers; it can also send them on the wire.	`https://github.com/secdev/scapy`
Selenium	A python library used to automate web-browser tasks and web-acceptance testing. The library works with Selenium webdrivers for Firefox, Chrome, and Internet Explorer to run tests on web browsers.	`https://pypi.org/project/selenium/`

You can find more of the python packages categorized into different areas at the following link: `https://github.com/vinta/awesome-python`.

Accessing module source code

You can access the source code of any module that you use in two ways. First, go to the `module` page at `github.com` and view all the files, releases, commits, and issues in one place, as in the following screenshot. I have read access to all shared code via the `netmiko` module maintainer and can see a full list of commits and file contents:

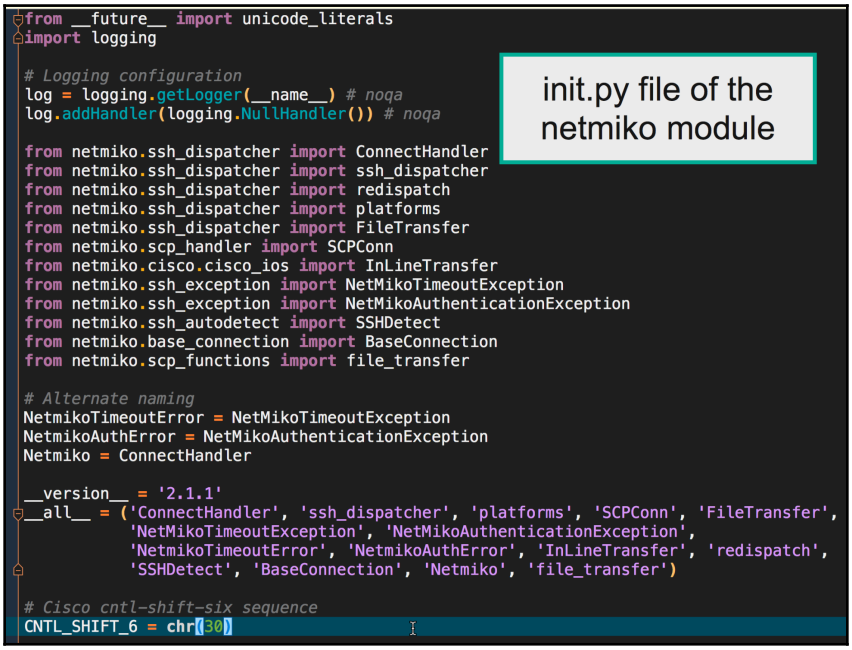


The second method is to install the package itself in the Python site-package directory using `pip` or PyCharm GUI. What `pip` actually does is it goes to GitHub and downloads the module content and runs `setup.py` to install and register the module. You can see the module files, but this time you have full read/write access on all files and you can change the original code. For example, the following code leverages the `netmiko` library to connect to a Cisco device and execute the `show arp` command on it:

```
from netmiko import ConnectHandler

device = {"device_type": "cisco_ios",
```

```
            "ip": "10.10.88.110",
            "username": "admin",
            "password": "access123"}

net_connect = ConnectHandler(**device)
output = net_connect.send_command("show arp")
```

If I want to see the netmiko source code, I can go either to site-packages where the netmiko library installed and list all files *or* I can use *Ctrl* and left-click on the module name in PyCharm. This will open the source code in a new tab:




```
from __future__ import unicode_literals
import logging

# Logging configuration
log = logging.getLogger(__name__) # noqa
log.addHandler(logging.NullHandler()) # noqa

from netmiko.ssh_dispatcher import ConnectHandler
from netmiko.ssh_dispatcher import ssh_dispatcher
from netmiko.ssh_dispatcher import redispatch
from netmiko.ssh_dispatcher import platforms
from netmiko.ssh_dispatcher import FileTransfer
from netmiko.scp_handler import SCPConn
from netmiko.cisco.cisco_ios import InLineTransfer
from netmiko.ssh_exception import NetMikoTimeoutException
from netmiko.ssh_exception import NetMikoAuthenticationException
from netmiko.ssh_autodetect import SSHDetect
from netmiko.base_connection import BaseConnection
from netmiko.scp_functions import file_transfer

# Alternate naming
NetmikoTimeoutError = NetMikoTimeoutException
NetmikoAuthError = NetMikoAuthenticationException
Netmiko = ConnectHandler

__version__ = '2.1.1'
__all__ = ('ConnectHandler', 'ssh_dispatcher', 'platforms', 'SCPConn', 'FileTransfer',
           'NetMikoTimeoutException', 'NetMikoAuthenticationException',
           'NetmikoTimeoutError', 'NetmikoAuthError', 'InLineTransfer', 'redispatch',
           'SSHDetect', 'BaseConnection', 'Netmiko', 'file_transfer')

# Cisco cntl-shift-six sequence
CNTL_SHIFT_6 = chr(30)
```

Visualizing Python code

Ever wondered how a Python custom module or class is manufactured? How does the developer write the Python code and glue it together to create this nice and amazing *x* module? What's going on under the hood?

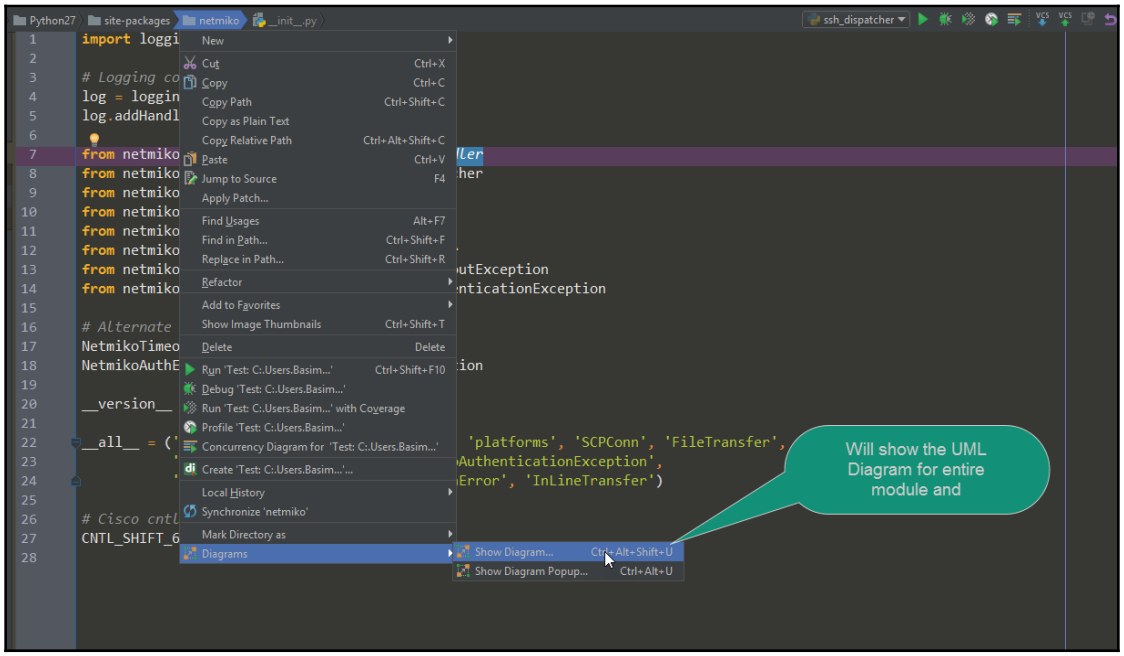
Documentation is a good start, of course, but we all know that it's not usually updated with every new step or detail that the developer added.

For example, we all know the powerful netmiko library created and maintained by Kirk Byers (`https://github.com/ktbyers/netmiko`) that leverages another popular SSH library called Paramiko (`http://www.paramiko.org/`). But we don't understand the details and how the classes are related to each other. If you need to understand the magic behind netmiko (or any other library) in order to process the request and return the result, please follow the next steps (requires PyCharm professional edition).

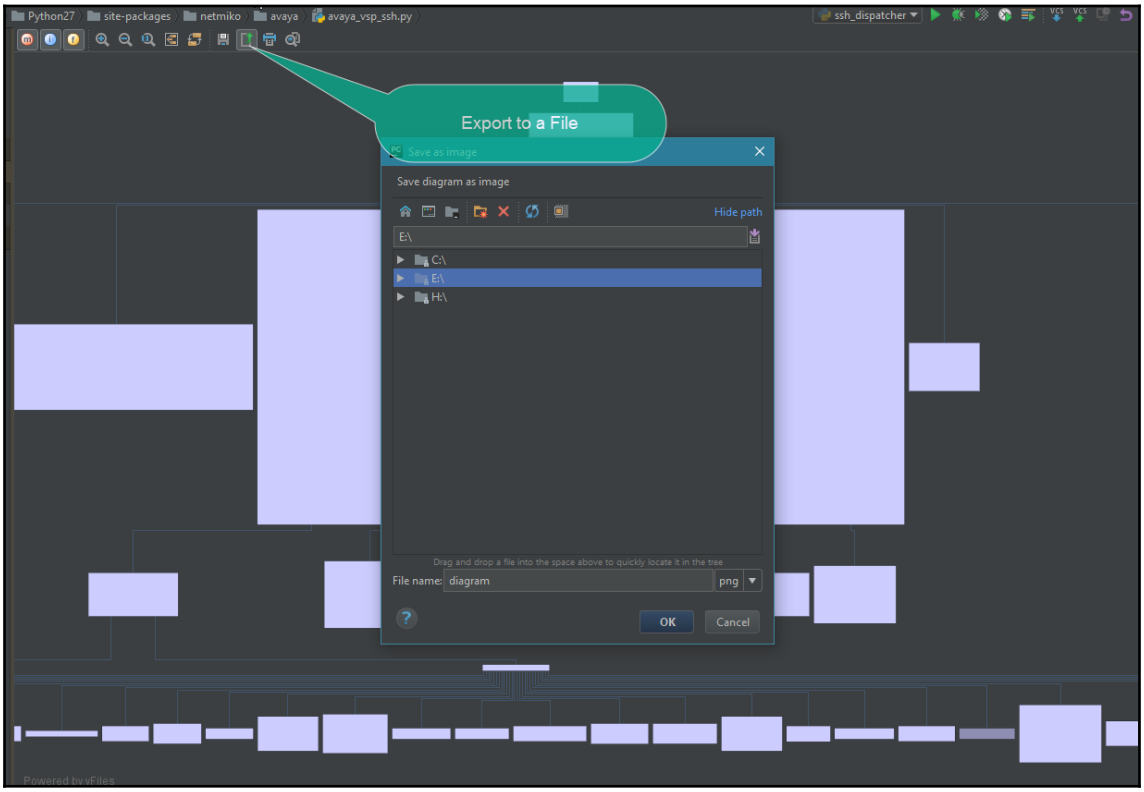
Code visualization and inspection in PyCharm is not supported in PyCharm community edition and is only supported in the professional version.

Following are the steps you need to follow:

1. Go to the netmiko module source code inside the Python library location folder (usually `C:\Python27\Lib\site-packages` on Windows or `/usr/local/lib/pyhon2.7/dist-packages` on Linux) and open the file from PyCharm.
2. Right-click on the module name that appears in the address bar and choose **Diagrams** | **Show Diagram**. Select Python class diagram from the pop-up window:



3. PyCharm will start to build the dependency tree between all classes and files in the `netmiko` module and then will show it in the same window. Note this process may require some time depending on your computer memory. Also, it's better to save the graph as an external image to view it:

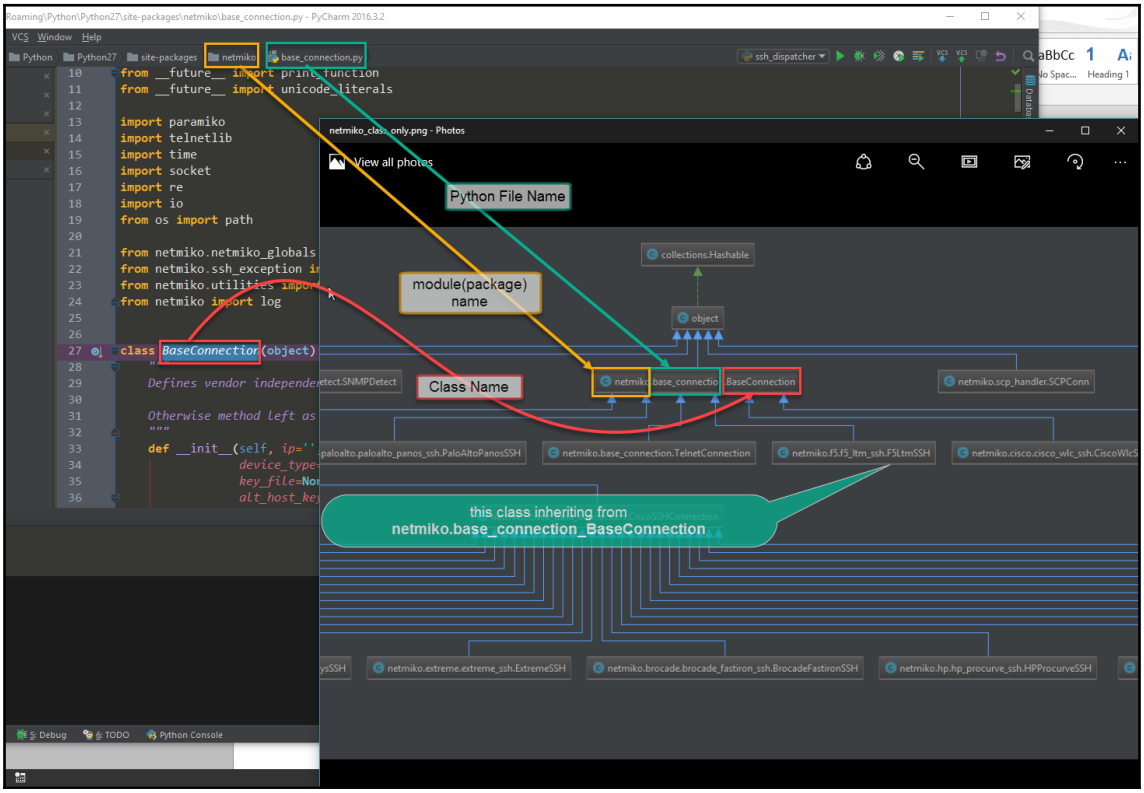


Based on the resulting graph, you can see that Netmiko is supporting a lot of vendors such as HP Comware, entrasys, Cisco ASA, Force10, Arista, Avaya, and so on, and all of these classes are are inheriting from the `netmiko.cisco_base_connection.CicsoSSHConnection` parent class (I think this is because they use the same SSH style as Cisco). This in turn inherits from another big parent class called `netmiko.cisco_base_connection.BaseConnection`.

Also, you can see that Juniper has its own class (`netmiko.juniper.juniper_ssh.JuniperSSH`) that connects directly to the big parent. Finally, we connect to the parent of all parents in python: the `Object` class (remember everything in Python is an object in the end).

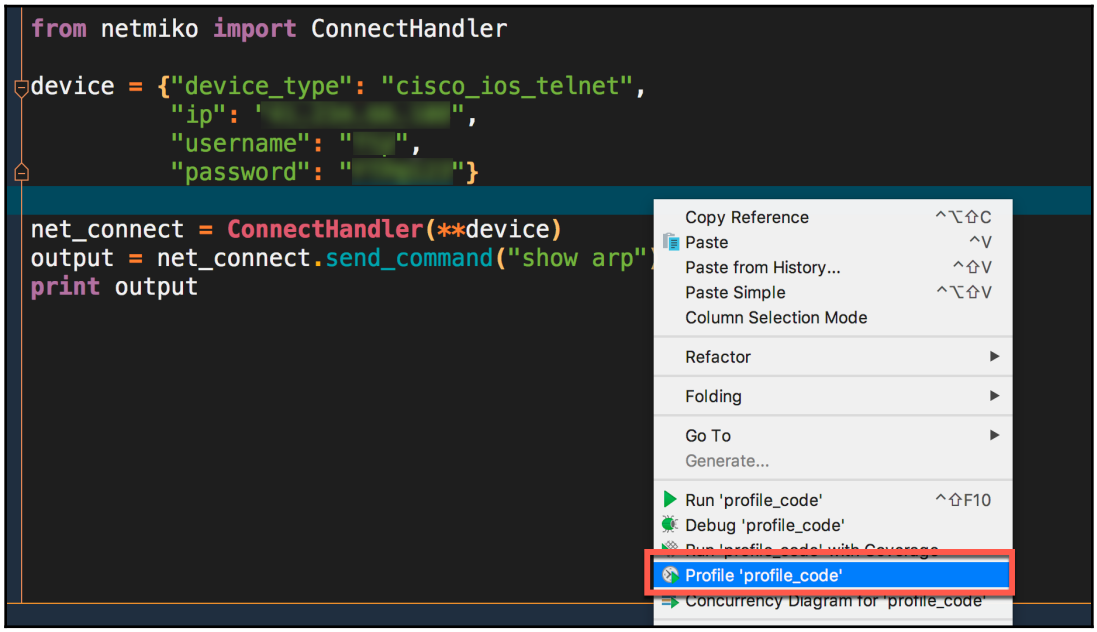
You can find a lot of interesting things such as an *SCP transfer* class and *SNMP* class, and with each one you will find the methods and parameters used to initialize the class.

So the `ConnectHandler` method is primarily used to check the `device_type` availability in the vendor classes and, based on returned data, it will use the corresponding SSH class:

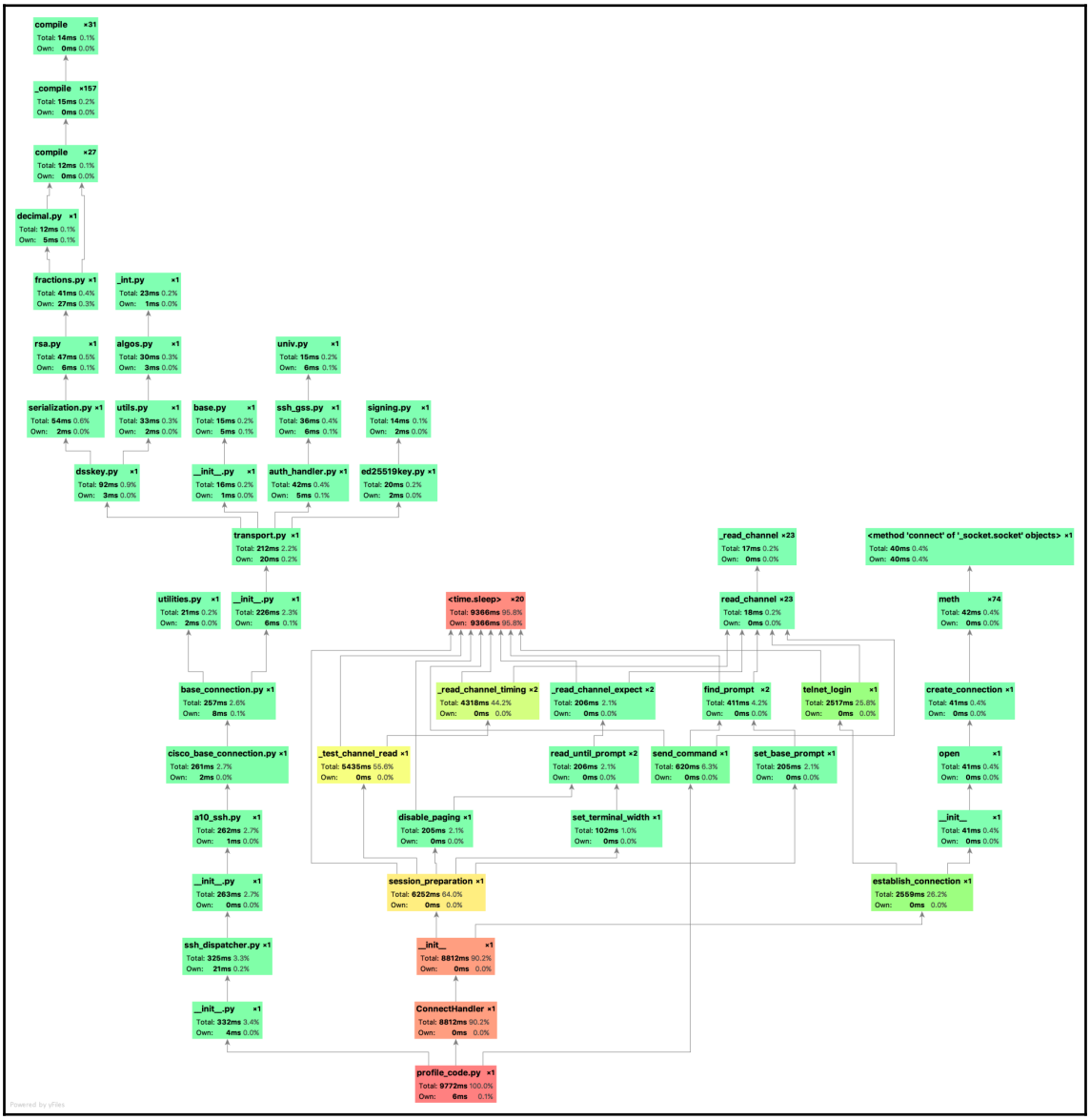


Another way to visualize your code is to see exactly which modules and functions are being hit during code execution. This is called profiling and it allows you to examine the functions during runtime.

First, you need to write your code as usual and then right-click on an empty space and select **profile** instead of running the code as normal:



Wait for the code to be executed. This time PyCharm will inspect each file that is called from your code and generate the *call graph* for the execution so you can easily know which files and functions are used and count the execution time for each one:



As you can see in the previous graph, our code in `profile_code.py` (bottom of the graph) will call the `ConnectHandler()` function which in turn will execute `__init__.py`, and execution will continue. On the graph's left side, you can see all files that it touched during your code execution.

Summary

In this chapter, we explored some of most popular network, system, and cloud packages provided in Python. Also, we learned how to access the module source code and to visualize it for better understanding of the internal code. We looked at the call flow for code while running. In the next chapter, we will start building a lab environment and apply our code to it.

3
Setting Up the Network Lab Environment

We now have a fair idea of how to write and develop Python scripts, the building blocks to creating programs. We will now move on to understanding why automation is an important topic in today's network, and then we will build our network automation lab using one of the popular pieces of software, called EVE-NG, which helps us to virtualize network devices.

We will cover the following topics in this chapter:

- When and why to automate the network
- Screen scraping versus API automation
- Why to use Python for network automation
- The future of network automation
- Lab setup
- Getting ready: installing EVE-NG
- Building an enterprise network topology

Technical requirements

In this chapter, we will cover the EVE-NG installation steps and how to create our lab environment. The installation will be done over VMware Workstation, VMware ESXi, and finally Red Hat KVM, so you should be familiar with the virtualization concept and have one of the hypervisors up and running prior to lab setup.

When and why to automate the network

Network automation is increasing all over the network world. However, it's really important to understand when and why to automate your network. For example, if you're an administrator of a few network devices (three or four switches) and you don't execute so many tasks on them regularly, then you might not need full automation for them. Actually, the time needed to write and develop a script and test and troubleshoot it might be greater than the time to do a simple task manually. On the other hand, if you're responsible for a big enterprise network that contains multi-vendor platforms and you always execute repetitive tasks, then it's highly recommended to have a script to automate it.

Why do we need automation?

There are several reasons for why automation is important for networks today:

- **Lower costs**: Using automation solutions (either developed in-house or purchased from vendors) will reduce network operation complexity and the time required to provision, configure, and operate network devices
- **Business continuity**: Automation will reduce human error during service creation over current infrastructure, and hence, allow businesses to reduce the service **time to market** (**TTM**)
- **Business agility**: Most network tasks are repeated and by automating them, you will increase productivity and drive business innovation
- **Correlation**: Building a solid automation workflow allows the network and systems administrators to perform root cause analysis faster and increases the possibility of solving the problem by correlating multiple events together

Screen scraping versus API automation

For a long period of time, the CLI was the only access method available to manage and operate network devices. Operators and administrators used to have SSH and Telnet to access the network terminal for configuration and troubleshooting. Python, or any programming language, has two approaches to communicating with devices. The first one is to use SSH or telnet the same as before and get the information, then process it. This method is called **screen scraping** and requires libraries that will be able to establish a connection to the device and execute a command directly on the terminal, and other libraries to process the returned information to extract useful data from it. This method often requires knowledge of additional parsing languages, such as regular expressions, to match the data pattern from the output and extract useful data from it.

The second method is called an **Application Programmable Interface** (**API**) and this method depends entirely on sending a structured request using REST or SOAP protocols to the device and returning the output, also in structured format, encoded in JSON or XML. The time needed for processing the returned data in this method is quite small compared to the first method; however, the API requires additional configuration on network devices to support it.

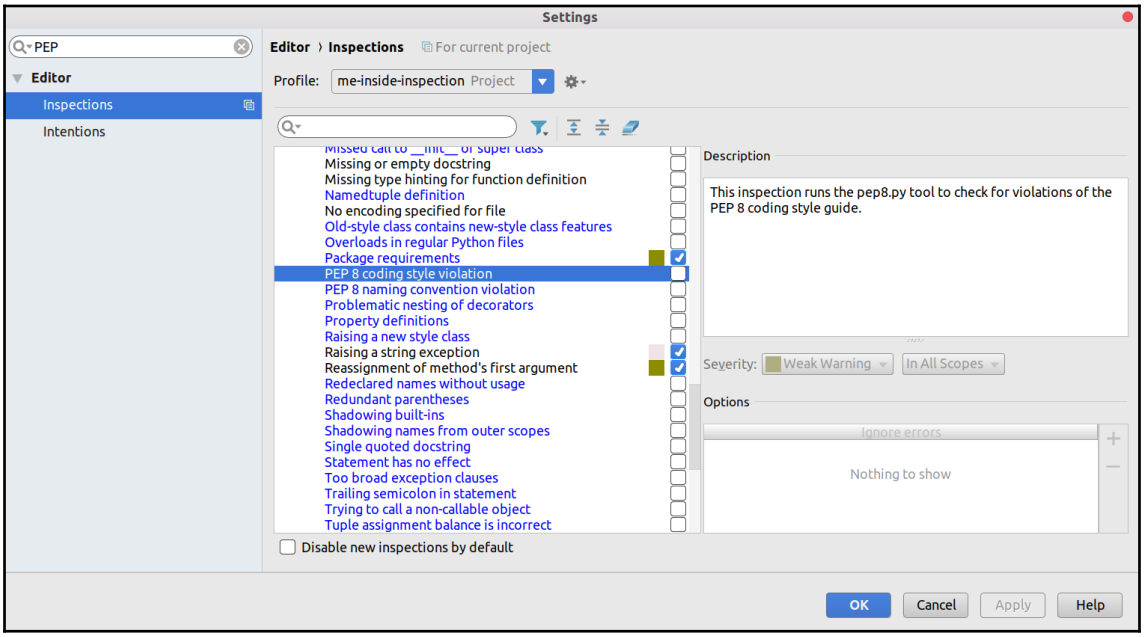
Why use Python for network automation?

Python is a pretty well-structured and easy programming language available today and targets many areas in technology, web and internet development, data mining and visualization, desktop GUI, analysis, game building, and automation testing; that's why it's called a *general purpose language.*

So, there are three reasons to choose Python:

- **Readability and ease of use**: When you develop using Python, you actually find yourself writing in English. Many keywords and program flows inside Python are structured to have readable statements. Also, Python doesn't require `;` or curly braces to start and end blocks, which gives Python a shallow learning curve. Finally, Python has some optional rules, called PEP 8, that tell Python developers how to format their program to have readable code.

You can configure PyCharm to take care of these rules and check whether your code violates them or not by going to **Settings | Inspections | PEP 8 coding style violation**:



- **Libraries**: This is the real power of Python: libraries and packages. Python has a wide range of libraries in many areas. Any Python developer can easily develop a Python library and upload it online to make it available to other developers. Libraries are uploaded to a website called PyPI (`https://pypi.Python.org/pypi`) and linked to a GitHub repository. When you want to download the library to your PC, then you use a tool called `pip` to connect to PyPI and download it locally. Network vendors such as Cisco, Juniper, and Arista developed libraries to facilitate access to their platforms. Most vendors are pushing to make their libraries easy to use and require minimum installation and configuration steps to retrieve useful information from devices.

- **Powerful**: Python tries to minimize the number of steps required to reach the end result. For example, to print hello world using Java, you will need this block of code:

```
JAVA

public class Main {
  public static void main(String[] args) {
    System.out.println("hello world");
  }
}
```

However, in Python, the whole block is written in one line to print it, as shown in the following screenshot:

```
PYTHON

print('hello world')
```

Combining all these reasons together leads to making Python the de facto standard for automation and the first choice for vendors when it comes to automating network devices.

The future of network automation

For a long period of time, network automation only meant developing a script using a programming language such as Perl, TcL, or Python in order to execute tasks on different network platforms. This approach is known as **script-driven network automation**. But as the network becomes more complex and more service-oriented, new types of automation were required and started to appear, such as the following:

- **Software-defined network automation**: Network devices will have only a forwarding plane, while the control plane is implemented and created using an external software called an **SDN controller**. The benefit of this approach is there will be a single point of contact for any network changes and the SDN controller can accept those change requests from other software, such as an external portal, through well-implemented northbound interfaces.

- **High-level orchestration**: This approach requires software called an orchestrator that integrates with SDN controllers and enables the creation of network service models using languages, such as YANG, that abstract the service from the underlying devices that will run over it. Also, an orchestrator can integrate with a **Virtual Infrastructure Manager** (**VIM**) such as OpenStack and vCenter, in order to manage virtual machines as a part of network service modeling.
- **Policy-based networking**: In this type of automation, you describe what you want to have in the network and the system has all the details to figure out how to implement it in the underlying devices. This allows software engineers and developers to implement changes in the network and describe their application's needs in declarative policies.

Network lab setup

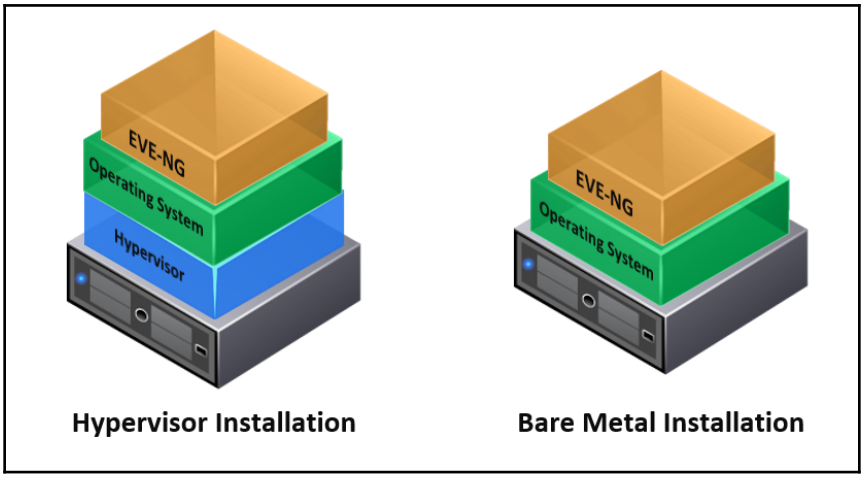
Now, we will start building our networking lab on a popular platform called EVE-NG. You could, of course, use a physical node to implement the topology, but a virtualized environment gives us an isolated and sandboxed environment to test many different configurations, plus the flexibility to add/remove nodes to/from the topology with a few clicks. Also, we can have multiple snapshots to our configuration so we can revert back to any scenario at any time.

EVE-NG (formerly known as UNetLab) is one of the most popular choices in network emulation. It supports a wide range of virtualized nodes from different vendors. There's another option, which is GNS3, but, as we will see during this chapter and the next one, EVE-NG provides many features that make it a solid choice for network modeling.

EVE-NG comes in three editions: Community, Pro, and Learning Center. We will use the Community edition as it contains all the features that we will need during this book.

Getting ready – installing EVE-NG

EVE-NG Community edition came with two options, OVA and ISO. The first option is to use OVA, which gives you the minimum installation steps required, given that you already have VMware Player/Workstation/Fusion, or VMware ESXi, or Red Hat KVM. The second option is to install it directly over a bare metal server without a hypervisor, this time using Ubuntu 16.06 LTS OS:



The ISO option, however, requires some advanced skills in Linux to prepare the machine itself and import the installation repositories into the operating system.

Oracle VirtualBox doesn't support the hardware acceleration needed by EVE-NG, so it's better to install it either in VMware or KVM.

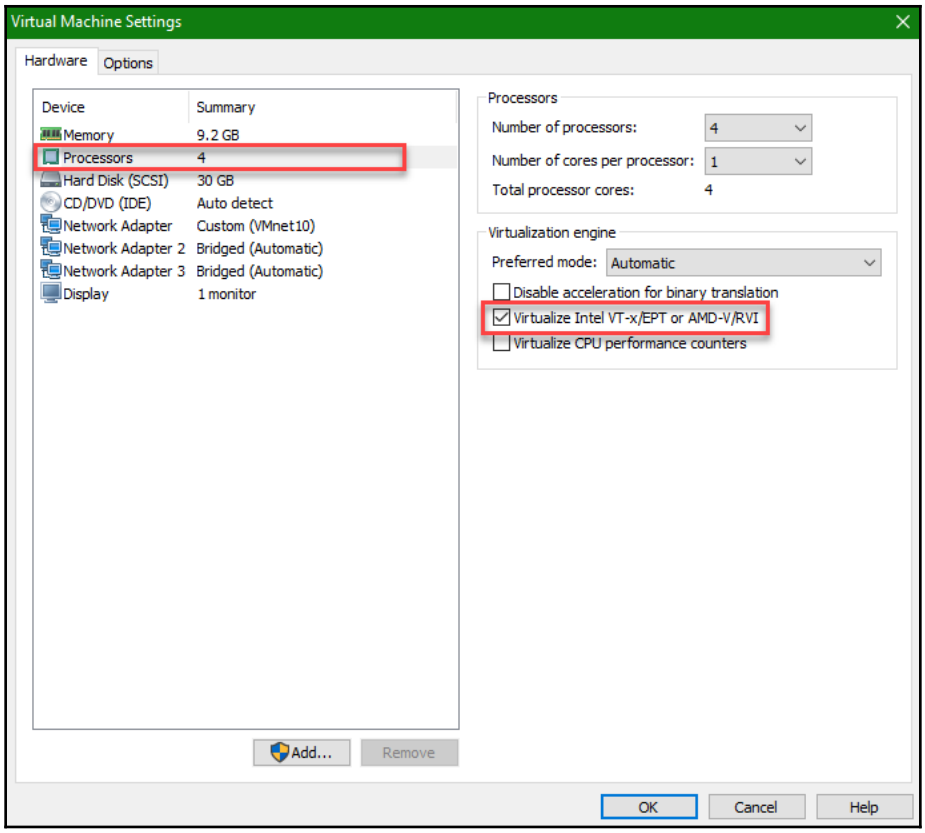
First, head to `http://www.eve-ng.net/index.php/downloads/eve-ng` to download the latest version of EVE-NG, then import it into your hypervisor. I dedicated 8 GB of memory and four vCPUs to the created machine, but you can add additional resources to it. In the next section, we will see how to import the downloaded image to hypervisors and configure each one.

Installation on VMware Workstation

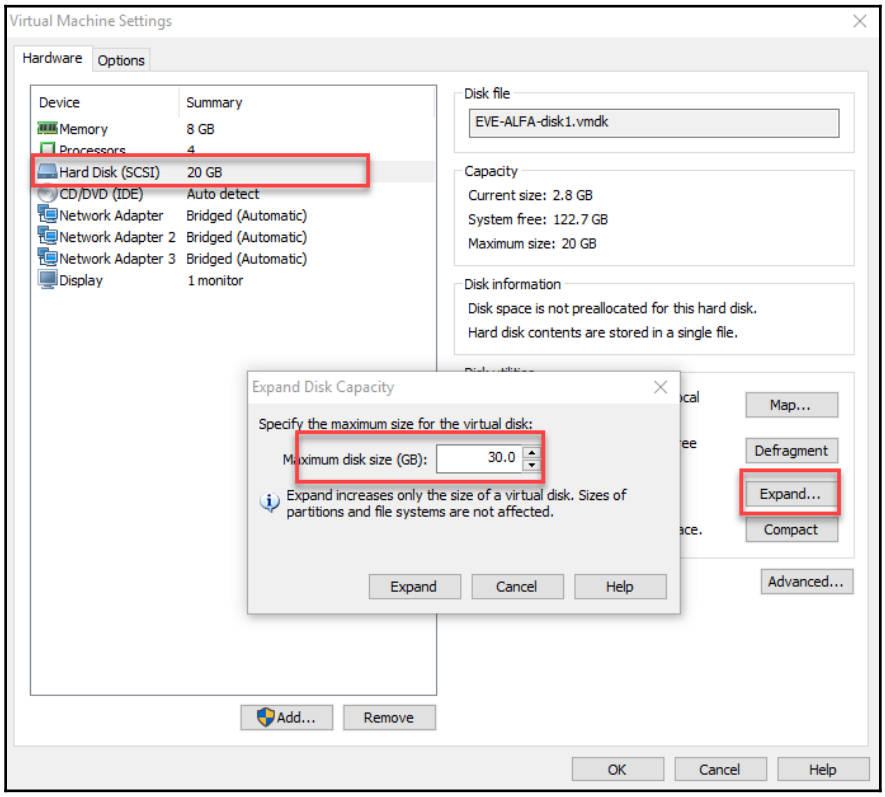
In the following steps, we will import the downloaded EVE-NG OVA image into VMware Workstation. OVA-based images contain files that describe the virtual machine in terms of hard disk, CPU, and RAM values. You can later modify these numbers after importing them:

1. Open VMware workstation and from **File**, choose **Open** to import the OVA.
2. After completing the import process, right-click on the newly created machine and choose **Edit Settings**.

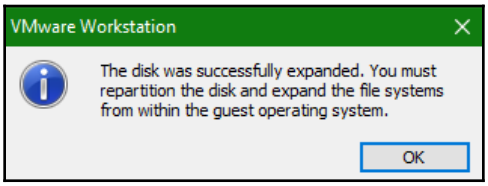
3. Increase the number of processors to **4** and the memory allocated to 8 GB (again, you could add more if you have the resources but this setting will be enough for our lab).
4. Make sure the **Virtualize Intel VT-x/EPT or AMD-V/RVI** checkbox is enabled. This option instructs VMware workstation to pass the virtualization flags to the guest OS (nested virtualization):



Also, it's recommended to expand the hard disk by adding additional space to the existing hard disk in order to have enough space to host multiple images from vendors:



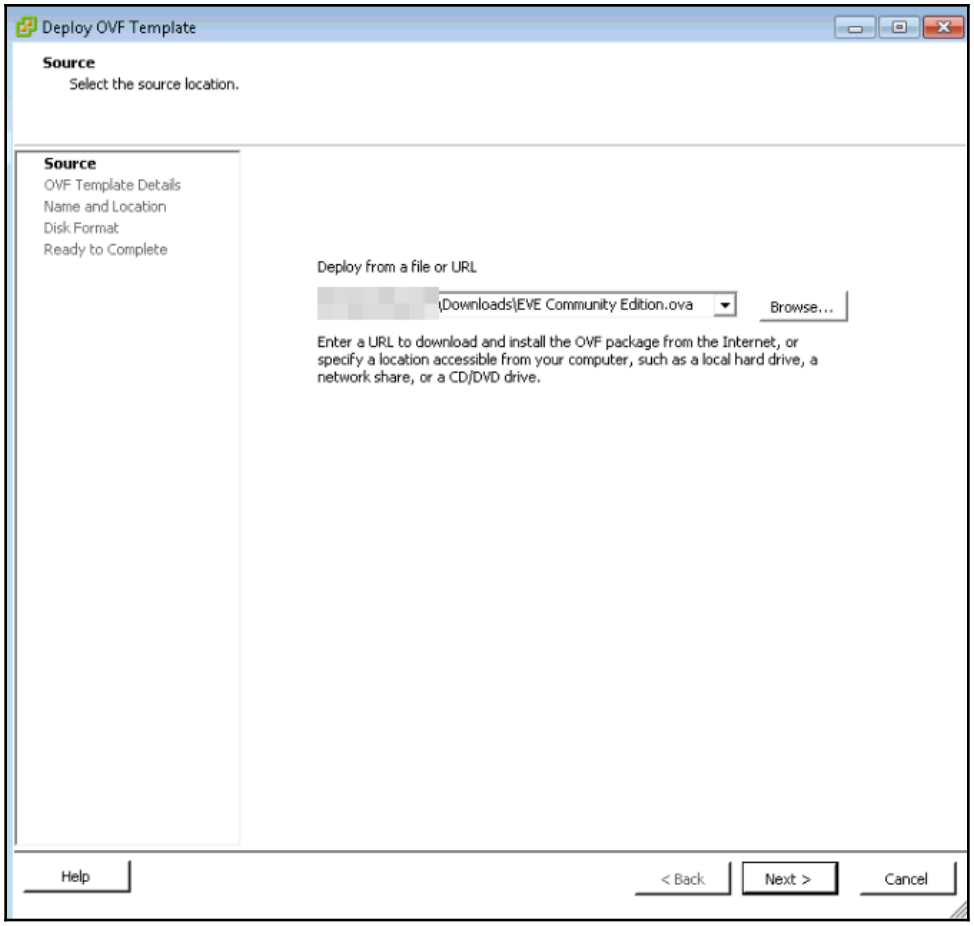
A message will appear after expanding the disk, indicating that the operation was done successfully and you need to follow some procedures in the guest operating system to merge the new space with the old one. Luckily for us, we don't need to do that as EVE-NG will merge any new space found in the hard disk with the old one during system boot:



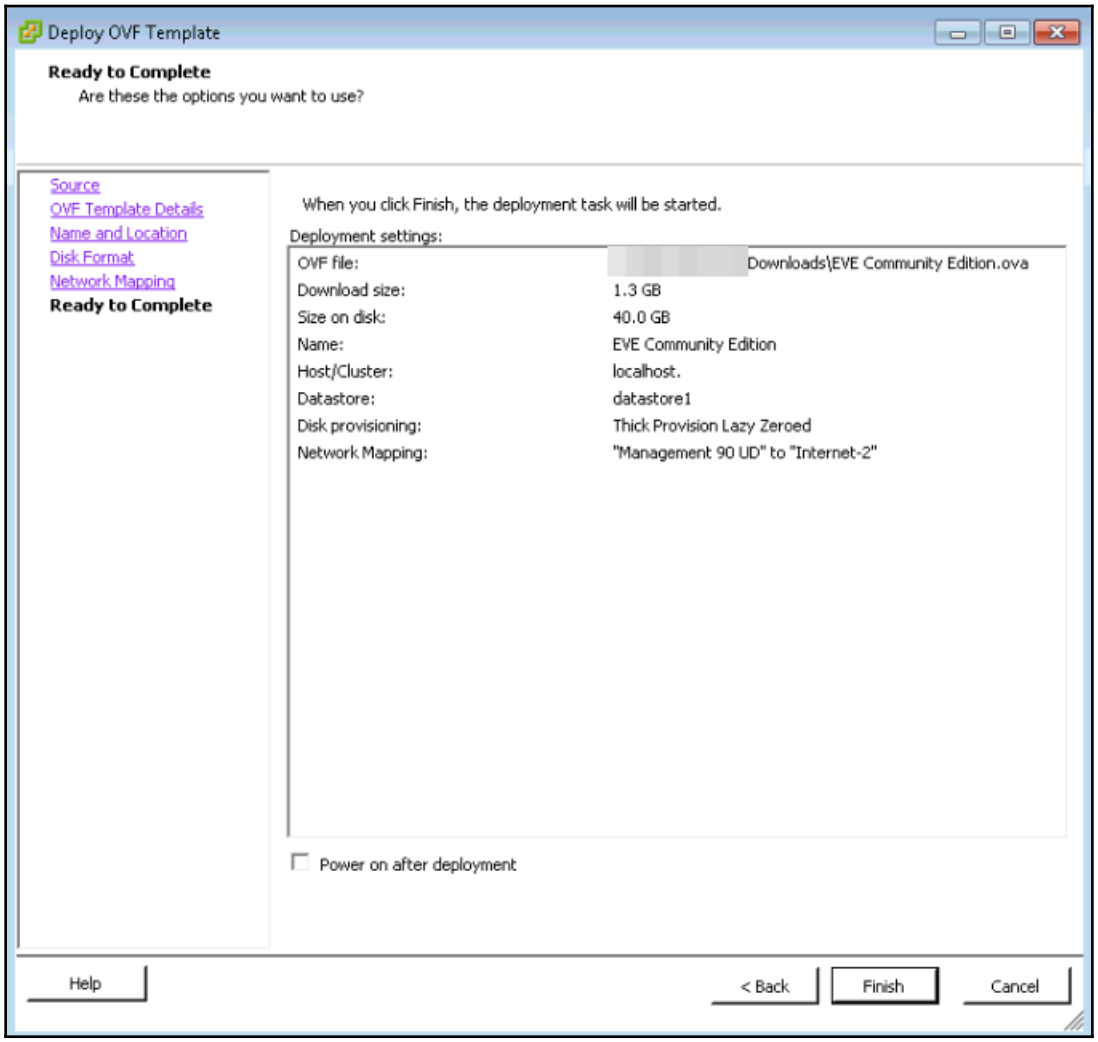
Installation over VMware ESXi

VMware ESXi is a good example of a type 1 hypervisor that runs directly on the system. Sometimes they're called bare-metal hypervisors, and they provide many features compared to type 2 hypervisors, such as VMware workstation/Fusion or VirtualBox:

1. Open the vSphere client and connect to your ESXi server
2. From the **File** menu, choose **Deploy OVF Template**
3. Enter the path for the downloaded OVA image and click **Next**:



4. Accept all the default settings suggested by the hypervisor till you land on the final page, **Ready to Complete**, and click on **Finish**:



ESXi will start to deploy the image on the hypervisor, and later you can change its settings and add more resources to it, as we did before in VMware workstation.

Installation over Red Hat KVM

You need to convert the downloaded OVA image to QCOW2 format, which is supported by KVM. Follow these steps to convert one format into another. We will need a special utility called `qemu-img` available inside the `qemu-utils` package:

1. Untar the downloaded OVA to extract the VMDK file (the HDD of the image):

```
tar -xvf EVE\ Community\ Edition.ova
EVE Community Edition.ovf
EVE Community Edition.vmdk
```

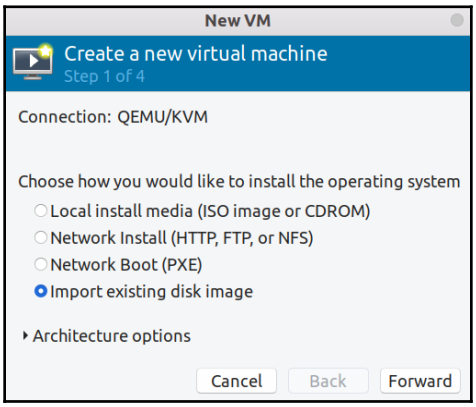
2. Install the `qemu-utils` tools:

```
sudo apt-get install qemu-utils
```

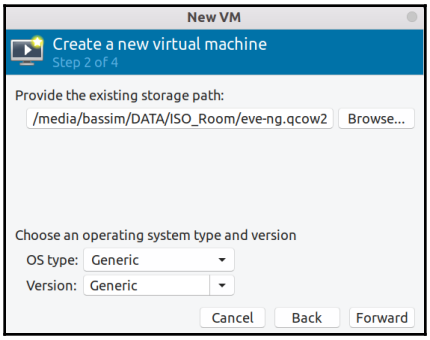
3. Now, convert the VMDK to QCOW2. It may take a few minutes for the conversion to be complete:

```
qemu-img convert -O qcow2 EVE\ Community\ Edition.vmdk eve-ng.qcow
```

Finally, we have our own `qcow2` file ready to be hosted inside the Red Hat KVM. Open the KVM console and choose the **Import existing disk image** option from the menu:



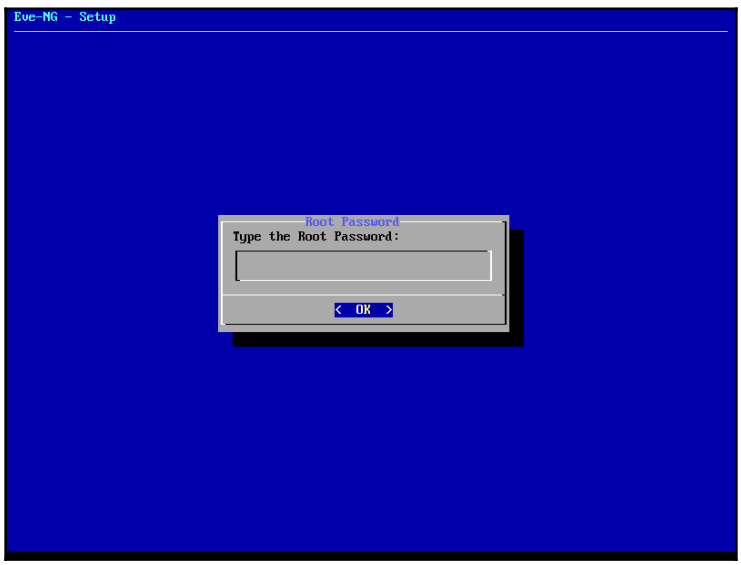
Then, choose the path of the converted image and click on **Forward**:



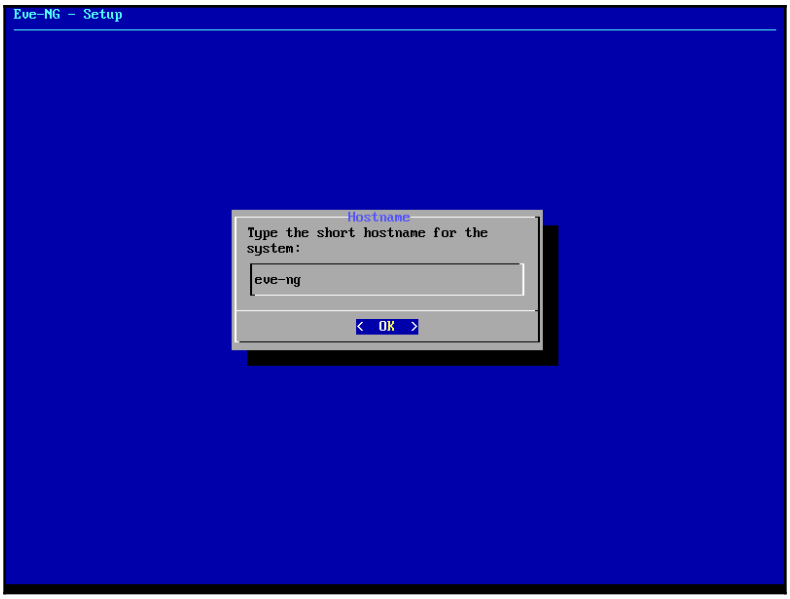
Accessing EVE-NG

After you import the image to the hypervisor and start it, you will be asked to provide some information to complete the installation. First, you will be greeted with the EVE logo as an indication that the machine has been successfully imported over the hypervisor and it is ready to start the boot phase:

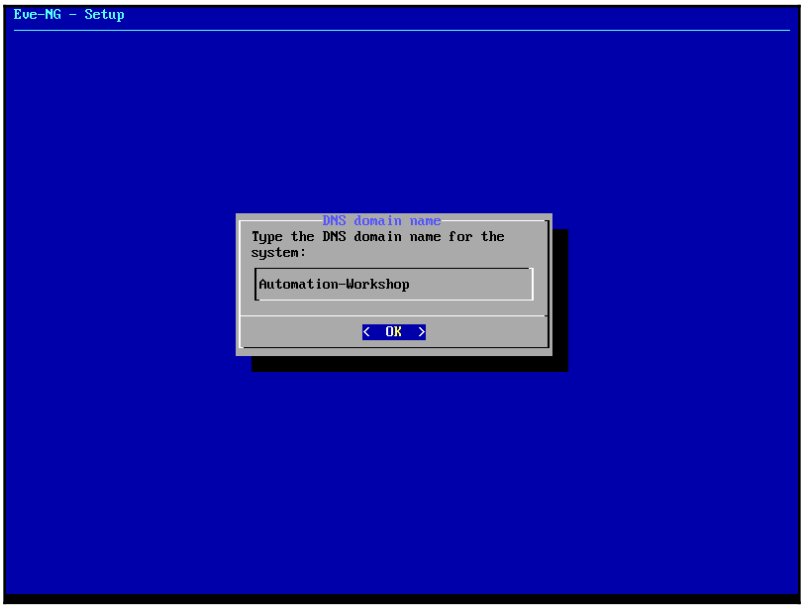
1. Provide the root password that will be used for SSHing to the EVE machine. By default, it will be `eve`:



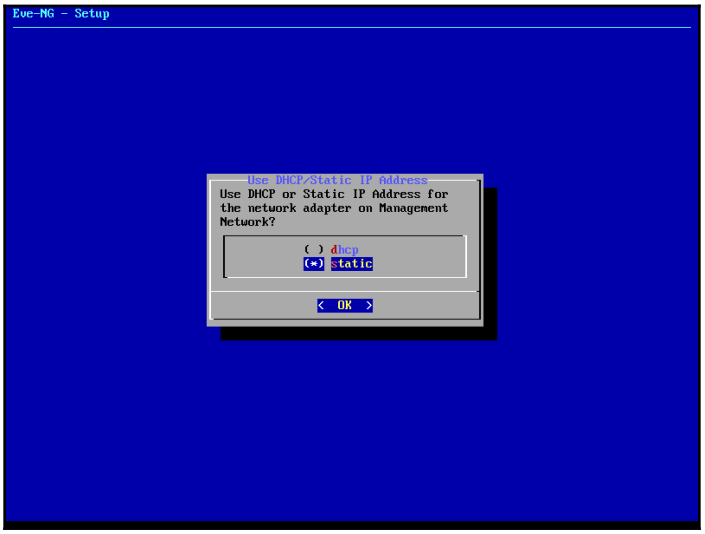
2. Provide the hostname that will be used as a name inside Linux:



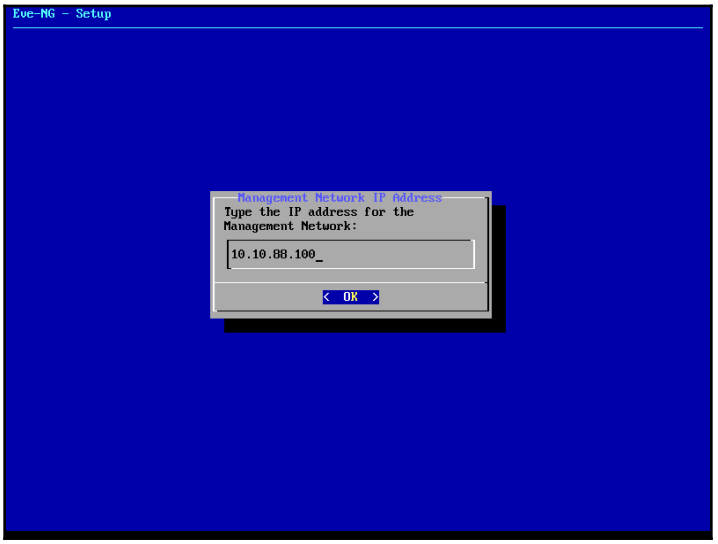
3. Provide a domain name for the machine:



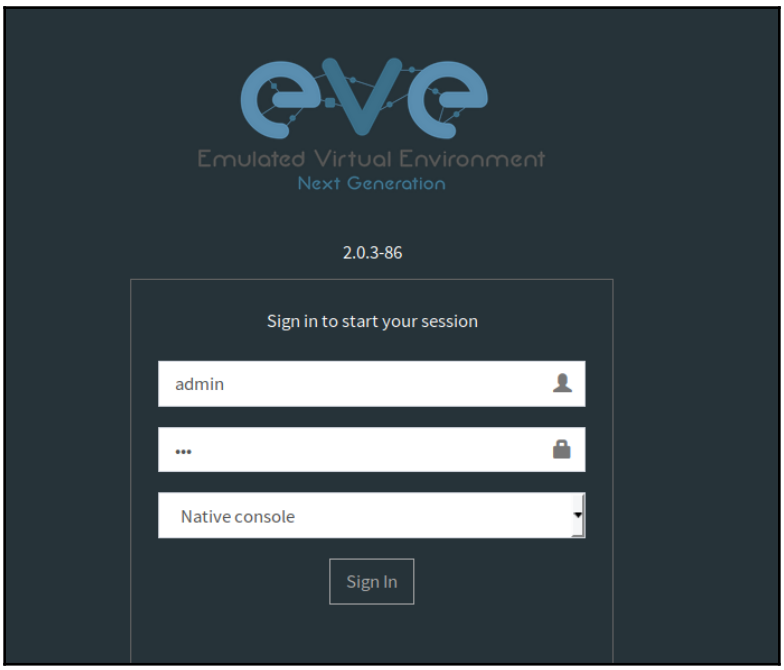
4. Choose to configure networking with the static method. This will ensure the IP address given will be persistent even after machine reboot:



5. Finally, provide the static IP address from a range that is reachable from your network. This IP will be used to SSH to EVE and upload vendor images to the repositories:



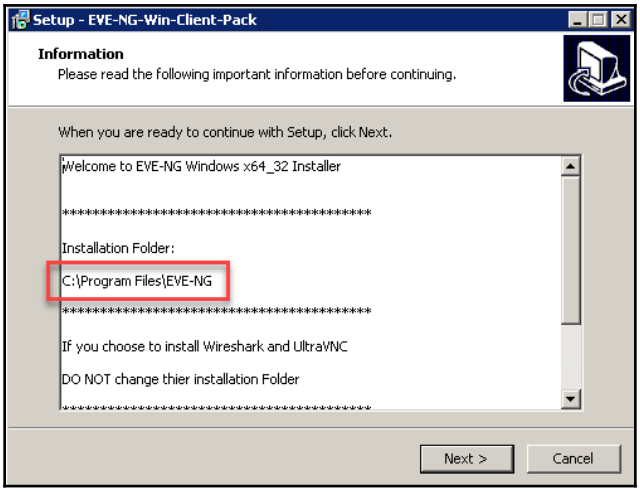
In order to access the EVE-NG GUI, you need to open a browser and go to `http://<server_ip>`. Please note `server_IP` is what we used during the installation steps:



The default username for the GUI is `admin` and the password is `eve`, while the default username for SSH is `root` and the password is what was provided during the installation steps.

Installing EVE-NG client pack

The client pack that comes with EVE-NG allows us to choose which application is used when you telnet or SSH to the device (either PuTTY or SecureCRT) and set up Wireshark for remote packet captures between links. Also, it facilitates work on RDP- and VNC-based images. First, you need to download the client pack to your PC from `http://eve-ng.net/index.php/downloads/windows-client-side-pack`, then extract the file to `C:\Program Files\EVE-NG`:



The extracted files contain many scripts written in Windows batch scripting (`.bat`) to configure the machine that will be used to access EVE-NG. You will find scripts that configure the default Telnet/SSH client and another one for Wireshark and the VNC. The software sources are also available inside the folder:

If you are using a Linux desktop such as Ubuntu or Fedora, then you could use this excellent project from GitHub to get the client pack: `https://github.com/SmartFinn/eve-ng-integration`.

Loading network images into EVE-NG

All network images obtained from vendors should be uploaded to `/opt/unetlab/addons/qemu`. EVE-NG support QEMU-based images and Dynamics images, and also iOL (iOS On Linux).

When you get an image from a vendor, you should create a directory inside `/opt/unetlab/addons/qemu` and upload the image to that directory; then, you should execute this script to fix the permission of the uploaded image:

```
/opt/unetlab/wrappers/unl_wrapper -a fixpermission
```

Building an enterprise network topology

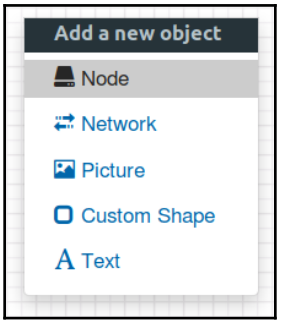
In our base lab setup, we will simulate an enterprise network that has four switches and one router that act as a gateway to outside networks. Here is the IP schema that will be used for each node:

Node name	**IP**
GW	`10.10.88.110`
Switch1	`10.10.88.111`
Switch2	`10.10.88.112`
Switch3	`10.10.88.113`
Switch4	`10.10.88.114`

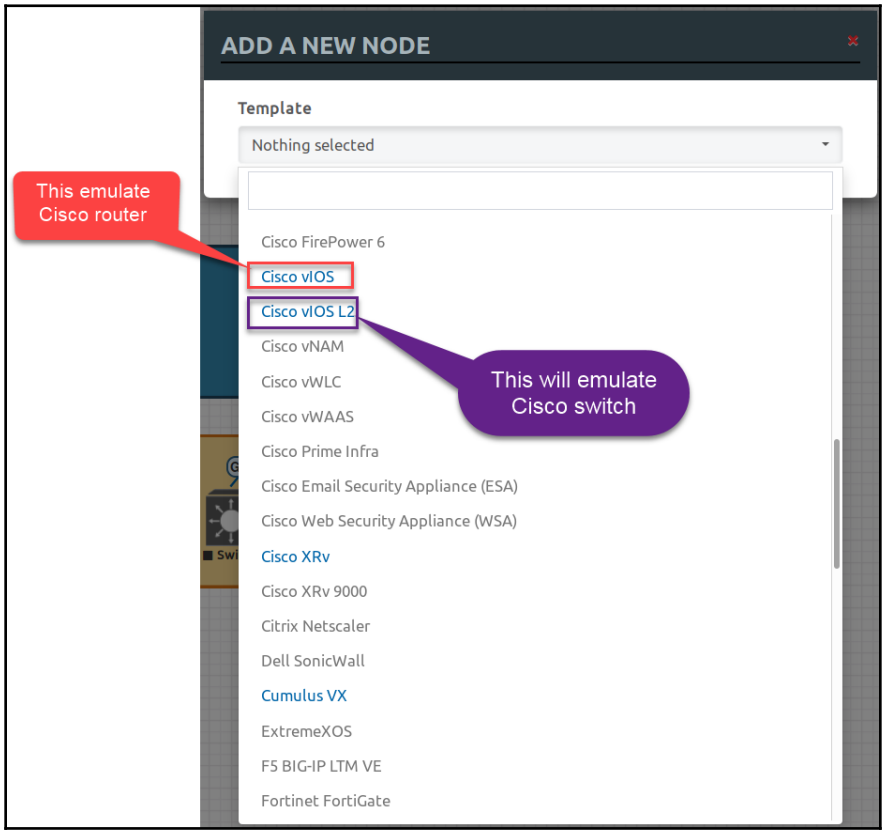
Our Python script (or Ansible playbook) will be hosted on an external Windows PC that connects to the management of each device.

Adding new nodes

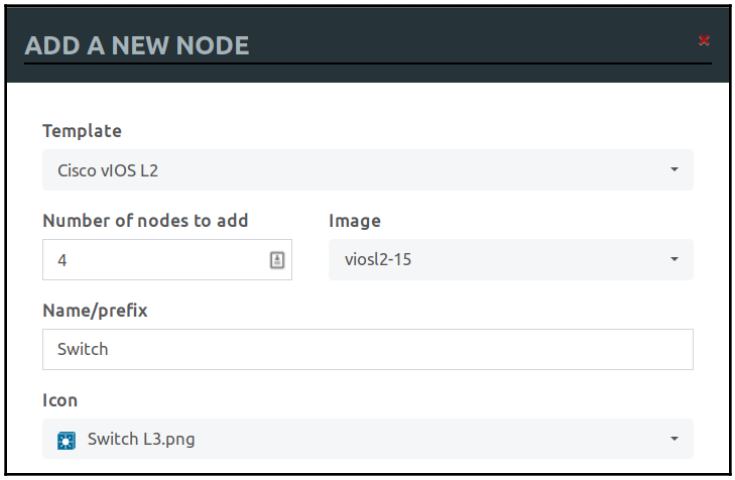
We will start by choosing the IOSv image that was already uploaded to EVE and add four switches to the topology. Right-click on any empty space in the topology and from the drop-down menu named **Add a new object**, choose to add a **Node**:



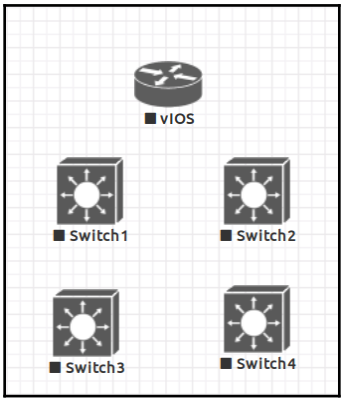
You should see two Cisco images colored blue as indication that they were successfully added to the available images inside the EVE-NG library and mapped to the corresponding template. Choose **Cisco vIOS L2** to add Cisco switches:



Increase the **Number of nodes to add** to **4** and click **OK**:



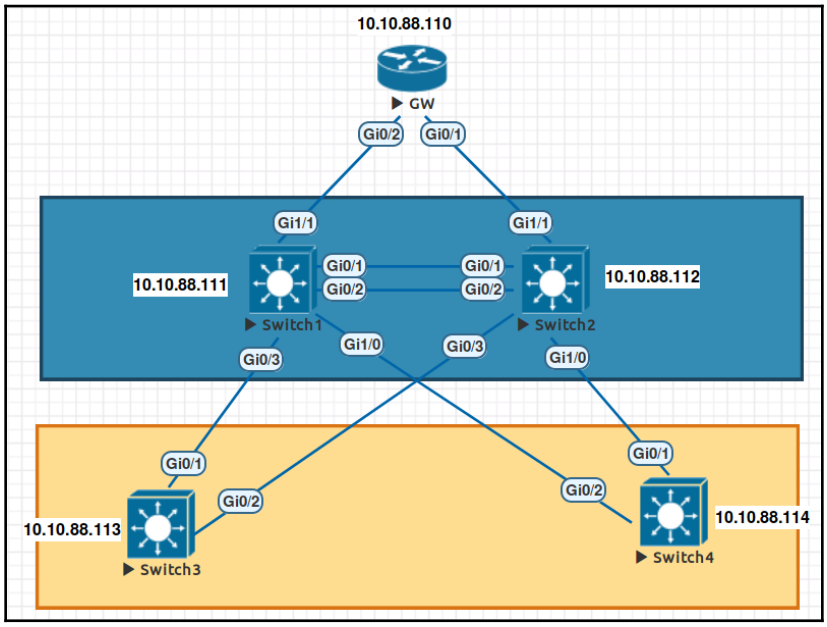
Now, you will see four switches added to the topology; repeat this again and add the router, but this time choose **Cisco vIOS**:



Connecting nodes together

Now, start to connect the nodes with each other while the nodes are offline, and repeat for each node till you finish connecting all of them inside the topology; then, start the lab:

The final view after adding IP addresses and some custom shapes to the topology will be as follows:



Now, our topology is ready and should be loaded with basic configuration. I used the following snippet as a configuration base for any Cisco-IOS device that enabled SSH and telnet and configured the username for access. Notice that there are some parameters surrounded with `{{ }}`. We will discuss them in the next chapter when we generate a golden configuration using a Jinja2 template but, for now, replace them with `hostname` and the management IP address for each device respectively:

```
hostname {{hostname}}
int gig0/0
  no shutdown
  ip address {{mgmt_ip}} 255.255.255.0

aaa new-model
aaa session-id unique
```

```
aaa authentication login default local
aaa authorization exec default local none

enable password access123
username admin password access123
no ip domain-lookup

lldp run

ip domain-name EnterpriseAutomation.net
ip ssh version 2
ip scp server enable
crypto key generate rsa general-keys modulus 1024
```

Summary

In this chapter, we learned about the different types of network automation available today and why we chose Python to be our primary tool in network automation. Also, we learned how to install EVE-NG over different hypervisors and platforms, how to provide the initial configuration, and how to add our network images to the images catalog. Then, we added different nodes and connected them together to create our network enterprise lab.

In the next chapter, we will start building our Python scripts that automate different tasks in the topology using different Python libraries, such as telnetlib, Netmiko, Paramiko, and Pexpect.

4
Using Python to Manage Network Devices

Now we have a fair knowledge about how to use and install Python in different operating systems and also how to build the network topology using the EVE-NG. In this chapter, we will discover how to leverage many network automation libraries, used today to automate various network tasks. Python can interact with network devices on many layers.

First, it can handle low-level layers with socket programming and `socket` modules, which serve as low-level networking interfaces between operating systems that run Python and the network device. Also, Python modules provide higher-level interaction through telnet, SSH, and API. In this chapter, we will dive deep into how to use Python to establish remote connections and execute commands on remote devices using telnet and SSH modules.

The following topics will be covered:

- Using Python to telnet to devices
- Python and SSH
- Handling IP addresses and networks with `netaddr`
- Network automation sample use cases

Technical requirements

The following tools should be installed and available in your environment:

- Python 2.7.1x
- PyCharm Community or Pro Edition
- EVE-NG topology; please refer to `Chapter 3`, *Setting up the Network Lab Environment*, for how to install and configure the emulator

You can find the full scripts developed in this chapter at the following GitHub URL: `https://github.com/TheNetworker/EnterpriseAutomation.git`.

Python and SSH

Unlike telnet, SSH provides a secure channel to exchange data between client and server. The tunnel created between the client and the device is encrypted with different security mechanisms that make it hard for anyone to decrypt the communication. The SSH protocol is the first choice for network engineers who need to securely administrate network nodes.

Python can communicate with network devices using the SSH protocol by utilizing a popular library called **Paramiko** that supports authentication, key handling (DSA, RSA, ECDSA, and ED25519), and other SSH features such as the `proxy` command and SFTP.

Paramiko module

The most widely used module for SSH in Python is called `Paramiko` and, as the GitHub official page says, the name Paramiko is a combination of the Esperanto words for "paranoid" and "friend." The module itself is written and developed using Python, though some core functions like crypto depend on the C language. You can find out more about the contributors and module history at the official GitHub link here: `https://github.com/paramiko/paramiko`.

Module installation

Open Windows cmd or Linux shell and execute the following command to download the latest `paramiko` module from PyPI. It will download additional dependency packages such as `cyrptography`, `ipaddress`, and `six` and install them on your machine:

```
pip install paramiko
```

```
bassim@me-inside:~$ pip install paramiko
Collecting paramiko
  Using cached paramiko-2.4.0-py2.py3-none-any.whl
Collecting cryptography>=1.5 (from paramiko)
  Using cached cryptography-2.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pynacl>=1.0.1 (from paramiko)
  Using cached PyNaCl-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pyasn1>=0.1.7 (from paramiko)
  Using cached pyasn1-0.4.2-py2.py3-none-any.whl
Collecting bcrypt>=3.1.3 (from paramiko)
  Using cached bcrypt-3.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting cffi>=1.7; platform_python_implementation != "PyPy" (from cryptography>=1.5->paramiko)
  Downloading cffi-1.11.4-cp27-cp27mu-manylinux1_x86_64.whl (406kB)
    100% |████████████████████████████████| 409kB 1.2MB/s
Collecting enum34; python_version < "3" (from cryptography>=1.5->paramiko)
  Using cached enum34-1.1.6-py2-none-any.whl
Collecting idna>=2.1 (from cryptography>=1.5->paramiko)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting asn1crypto>=0.21.0 (from cryptography>=1.5->paramiko)
  Using cached asn1crypto-0.24.0-py2.py3-none-any.whl
Collecting six>=1.4.1 (from cryptography>=1.5->paramiko)
  Using cached six-1.11.0-py2.py3-none-any.whl
Collecting ipaddress; python_version < "3" (from cryptography>=1.5->paramiko)
Collecting pycparser (from cffi>=1.7; platform_python_implementation != "PyPy"->cryptography>=1.5->pa
ramiko)
Installing collected packages: pycparser, cffi, enum34, idna, asn1crypto, six, ipaddress, cryptograph
y, pynacl, pyasn1, bcrypt, paramiko
```

You can verify that the installation is done successfully by entering the Python shell and importing the `paramiko` module as shown in the following screenshot. Python should import it successfully without printing any errors:

```
bassim@me-inside:~$ python
Python 2.7.14 (default, Sep 23 2017, 22:06:14)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>>
```

SSH to the network device

As usual, in every Python module, we first need to import it into our Python script, then we will create an SSH client by inheriting from `SSHClient()`. After that, we will configure the Paramiko to automatically add any unknown host-key and trust the connection between you and the server. Then, we will use the `connect` function and provide the remote host credentials:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
```

```
__EMAIL__ = "basim.alyy@gmail.com"

import paramiko
import time
Channel = paramiko.SSHClient()
Channel.set_missing_host_key_policy(paramiko.AutoAddPolicy())
Channel.connect(hostname="10.10.88.112", username='admin',
password='access123', look_for_keys=False,allow_agent=False)

shell = Channel.invoke_shell()
```

`AutoAddPolicy()` is one of the policies that can be used inside the `set_missing_host_key_policy()` function. It's preferred and acceptable in a lab environment. However, we should use a more restrictive policy in a production environment, such as `WarningPolicy()` or `RejectPolicy()`.

Finally, the `invoke_shell()` will start the interactive shell session towards our SSH server. You can provide additional parameters to it such as the terminal type, width, and height.

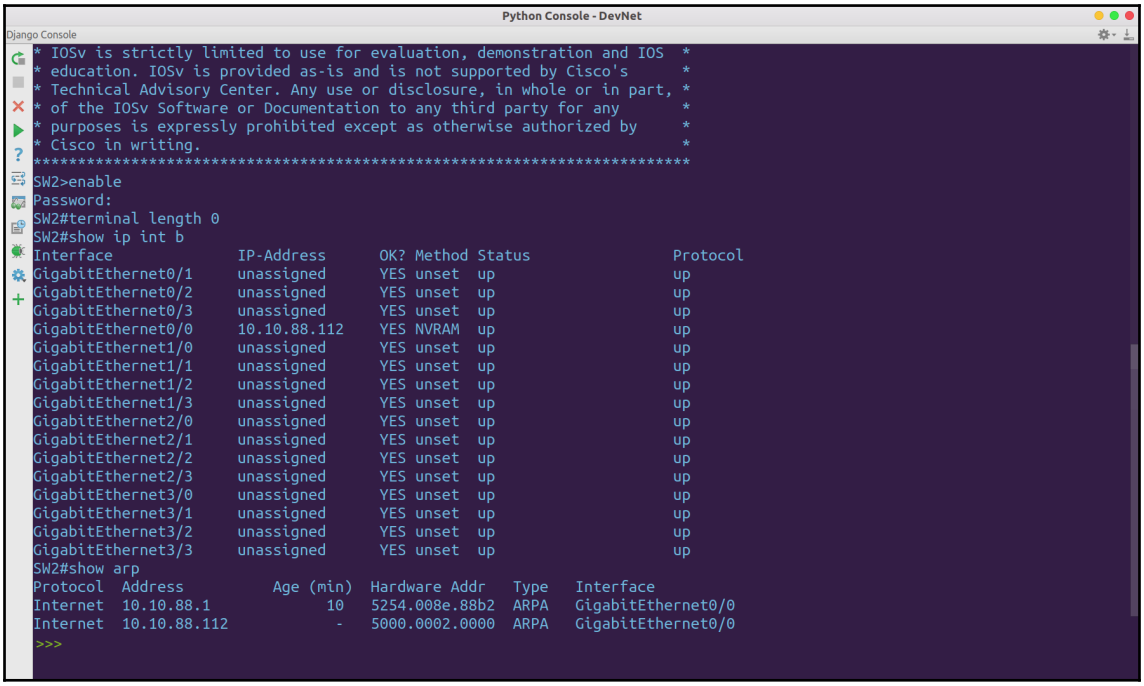
Paramiko connect parameters:

- `Look_For_Keys`: By default, it's `True`, and it will force the Paramiko to use the key-pair authentication where the user is using both private and public keys to authenticate against the network device. In our case, we will set it to `False` as we will use password authentication.
- `allow_agent paramiko`: It can connect to a local SSH agent OS. This is necessary when working with keys; in this case, since authentication is performed using a login/password, we will disable it.

The final step is to send a series of commands such as `show ip int b` and `show arp` to the device terminal and get the output back to our Python shell:

```
shell.send("enable\n")
shell.send("access123\n")
shell.send("terminal length 0\n")
shell.send("show ip int b\n")
shell.send("show arp \n")
time.sleep(2)
print shell.recv(5000)
Channel.close()
```

The script output is:



It's preferable to use `time.sleep()` when you need to execute commands that will take a long time on a remote device to force Python to wait some time till the device generates output and sends it back to python. Otherwise, python may return blank output to the user.

Netmiko module

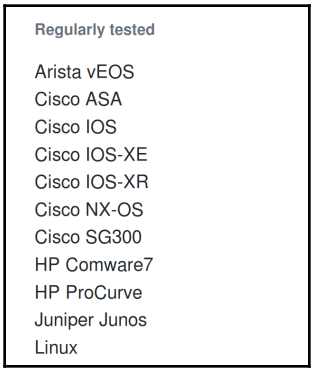
The `netmiko` module is an enhanced version of paramiko and targets network devices specifically. While paramiko is designed to handle SSH connections to a device and to check whether the device is a server, printer, or network device, Netmiko is designed with network devices in mind and handles SSH connections more efficiently. Also, Netmiko supports a wide range of vendors and platforms.

Netmiko is considered a wrapper around paramiko and extends its features with many additional enhancements, such as access to vendor-enabled modes directly given the enable password, reading configuration from a file and pushing it to devices, disabling paging during login, and sending the carriage return `"\n"` by default after each command.

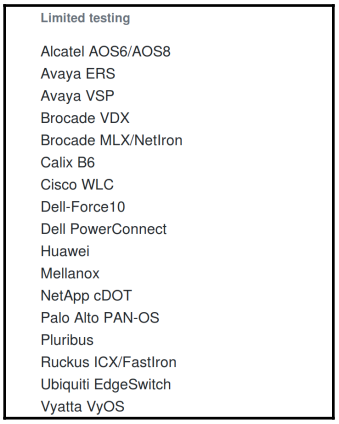
Vendor support

Netmiko supports many vendors and regularly adds new vendors to the supported list. Following is a list of supported vendors categorized into three groups: Regularly tested, Limited testing, and Experimental. You can find the list on the module GitHub page at `https://github.com/ktbyers/netmiko#supports`.

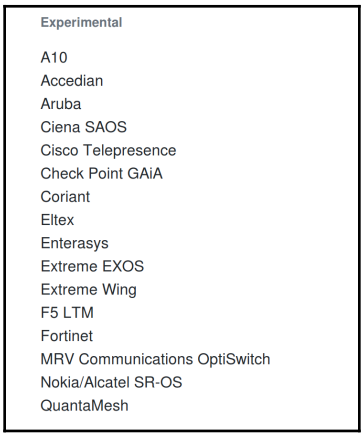
The following screenshot shows the number of supported vendors under the **Regularly tested** category:



The following screenshot shows the number of supported vendors under the **Limited testing** category:



The following screenshot shows the number of supported vendors under the **Experimental** category:



Installation and verification

To install `netmiko`, open the Windows cmd or Linux shell and execute the following command to get the latest package from PyPI:

```
pip install netmiko
```

```
bassim@me-inside:~$ pip install netmiko
Collecting netmiko
  Downloading netmiko-2.0.1.tar.gz (68kB)
    100% |████████████████████████████████| 71kB 450kB/s
Collecting paramiko>=2.0.0 (from netmiko)
  Using cached paramiko-2.4.0-py2.py3-none-any.whl
Collecting scp>=0.10.0 (from netmiko)
  Using cached scp-0.10.2-py2.py3-none-any.whl
Collecting pyyaml (from netmiko)
Collecting pyserial (from netmiko)
  Using cached pyserial-3.4-py2.py3-none-any.whl
Collecting textfsm (from netmiko)
  Downloading textfsm-0.3.2.tar.gz
Collecting cryptography>=1.5 (from paramiko>=2.0.0->netmiko)
  Using cached cryptography-2.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pynacl>=1.0.1 (from paramiko>=2.0.0->netmiko)
  Using cached PyNaCl-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pyasn1>=0.1.7 (from paramiko>=2.0.0->netmiko)
  Using cached pyasn1-0.4.2-py2.py3-none-any.whl
Collecting bcrypt>=3.1.3 (from paramiko>=2.0.0->netmiko)
  Using cached bcrypt-3.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting cffi>=1.7; platform_python_implementation != "PyPy" (from cryptography>=1.5->param
iko>=2.0.0->netmiko)
  Using cached cffi-1.11.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting enum34; python_version < "3" (from cryptography>=1.5->paramiko>=2.0.0->netmiko)
  Using cached enum34-1.1.6-py2-none-any.whl
Collecting idna>=2.1 (from cryptography>=1.5->paramiko>=2.0.0->netmiko)
```

Then import netmiko from the Python shell to make sure the module is correctly installed into Python site-packages:

```
$python
>>>import netmiko
```

Using netmiko for SSH

Now it's time to utilize netmiko and see its power for SSHing to network devices and executing commands. By default, netmiko handles many operations in the background during session establishment, such as adding unknown SSH key hosts, setting the terminal type, width, and height, and accessing enable mode when required, then disabling paging by running a vendor-specific command. You will need to define the devices first in dictionary format and provide five mandatory keys:

```
R1 = {
    'device_type': 'cisco_ios',
    'ip': '10.10.88.110',
    'username': 'admin',
    'password': 'access123',
    'secret': 'access123',
}
```

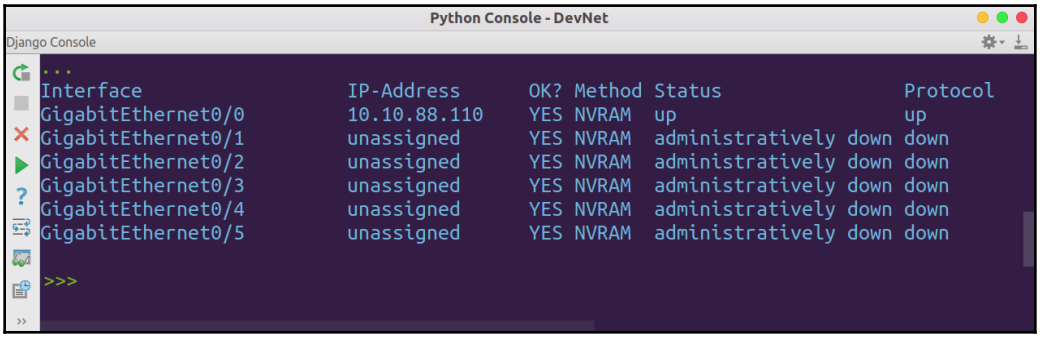
The first parameter is `device_type`, and it is used to define the platform vendor in order to execute the correct commands. Then, we need the `ip` address for SSH. This parameter could be the device hostname if it's already been resolved by your DNS, or just the IP address. Then we provide the `username`, `password`, and enable-mode password in `secret`. Notice you can use the `getpass()` module to hide the passwords and only prompt them during the script execution.

While the keys order inside the variable is not important, the key's name should be exactly the same as provided in the previous example in order for netmiko to correctly parse the dictionary and to start to establish a connection to the device.

Next, we will import the `ConnectHandler` function from the netmiko module and give it the defined dictionary to start the connection. Since all our devices are configured with an enable-mode password, we need to access the enable mode by providing `.enable()` to the created connection. We will execute the command on the router terminal by using `.send_command()`, which will execute the command and return the device output to the variable:

```
from netmiko import ConnectHandler
connection = ConnectHandler(**R1)
connection.enable()
output = connection.send_command("show ip int b")
print output
```

The script output is:

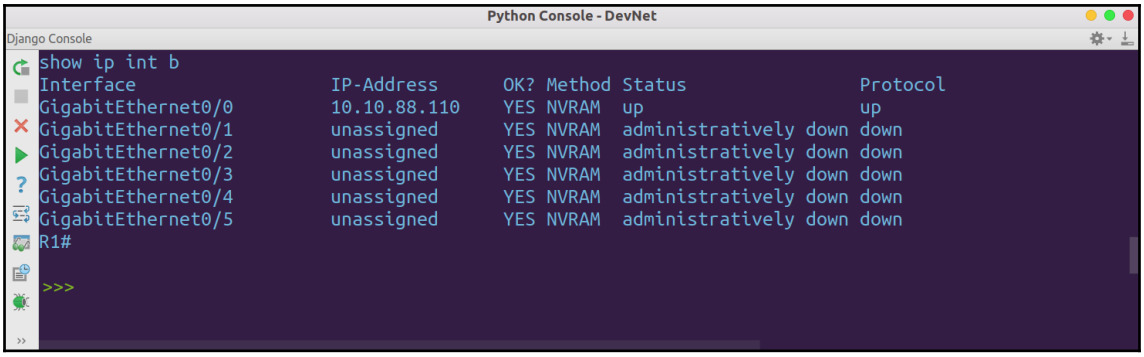


Notice how the output is already cleaned from the device prompt and the command that we executed on the device. By default, Netmiko replaces them and generates a cleaned output, which could be processed by regular expressions, as we will see in the next chapter.

If you need to disable this behavior and want to see the device prompt and executed command in the returned output, then you need to provide additional flags to `.send_command()` functions:

```
output = connection.send_command("show ip int
b",strip_command=False,strip_prompt=False)
```

The `strip_command=False` and `strip_prompt=False` flags tell netmiko to keep both the prompt and command and not to replace them. They're `True` by default and you can toggle them if you want:



Configuring devices using netmiko

Netmiko can be used to configure remote devices over SSH. It does that by accessing config mode using the `.config` method and then applies the configuration given in `list` format. The list itself can be provided inside the Python script or read from the file, then converted to a list using the `readlines()` method:

```
from netmiko import ConnectHandler

SW2 = {
    'device_type': 'cisco_ios',
    'ip': '10.10.88.112',
    'username': 'admin',
    'password': 'access123',
    'secret': 'access123',
}

core_sw_config = ["int range gig0/1 - 2","switchport trunk encapsulation
dot1q",
                  "switchport mode trunk","switchport trunk allowed vlan
1,2"]


print "########## Connecting to Device {0} ############".format(SW2['ip'])
net_connect = ConnectHandler(**SW2)
net_connect.enable()
```

```
print "***** Sending Configuration to Device *****"
net_connect.send_config_set(core_sw_config)
```

In the previous script, we did the same thing that we did before to connect to SW2 and enter enable mode, but this time we leveraged another netmiko method called `send_config_set()`, which takes the configuration in list format and accesses device configuration mode and starts to apply it. We have a simple configuration that modifies the `gig0/1` and `gig0/2` and applies trunk configuration on them. You can check if the command executed successfully by running `show run` command on the device; you should get output similar to the following:

```
interface GigabitEthernet0/1
 switchport trunk allowed vlan 1,2
 switchport trunk encapsulation dot1q
 switchport mode trunk
 media-type rj45
 negotiation auto
!
interface GigabitEthernet0/2
 switchport trunk allowed vlan 1,2
 switchport trunk encapsulation dot1q
 switchport mode trunk
 media-type rj45
 negotiation auto
!
```

Exception handling in netmiko

When we design our Python script, we assume that the device is up and running and also that the user has provided the correct credentials, which is not always the case. Sometimes there's a network connectivity issue between Python and the remote device or the user enters the wrong credentials. Usually, python will throw an exception if this happens and will exit, which is not the optimum solution.

The exception handling module in netmiko, `netmiko.ssh_exception`, provides some exception classes that can handle such situations. The first one is `AuthenticationException`, and will catch the authentication errors in the remote device. The second class is `NetMikoTimeoutException`, which will catch timeouts or any connectivity issues between netmiko and the device. What we will need to do is wrap our ConnectHandler() method with the try-except clause and catch timeout and authentication exceptions:

```
from netmiko import ConnectHandler
from netmiko.ssh_exception import AuthenticationException,
NetMikoTimeoutException
```

```
device = {
    'device_type': 'cisco_ios',
    'ip': '10.10.88.112',
    'username': 'admin',
    'password': 'access123',
    'secret': 'access123',
}


print "########## Connecting to Device {0}
############".format(device['ip'])
try:
    net_connect = ConnectHandler(**device)
    net_connect.enable()

    print "***** show ip configuration of Device *****"
    output = net_connect.send_command("show ip int b")
    print output

    net_connect.disconnect()

except NetMikoTimeoutException:
    print "=========== SOMETHING WRONG HAPPEN WITH {0}
============".format(device['ip'])

except AuthenticationException:
    print "========= Authentication Failed with {0}
============".format(device['ip'])

except Exception as unknown_error:
    print "============ SOMETHING UNKNOWN HAPPEN WITH {0} ============"
```

Device auto detect

Netmiko provides a mechanism that can guess the device type and detect it. It uses a combination of SNMP discovery OIDS and executes several show commands on the remote console to detect the router operating system and type, based on the output string. Then netmiko will load the appropriate driver into the `ConnectHandler()` class:

```
#!/usr/local/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"


from netmiko import SSHDetect, Netmiko
```

```
device = {
    'device_type': 'autodetect',
    'host': '10.10.88.110',
    'username': 'admin',
    'password': "access123",
}

detect_device = SSHDetect(**device)
device_type = detect_device.autodetect()
print(device_type)
print(detect_device.potential_matches)

device['device_type'] = device_type
connection = Netmiko(**device)
```

In the previous script:

- The `device_type` inside the device dictionary will be `autodetect`, which will tell `netmiko` to wait and not load the driver till the netmiko guesses it.
- Then we instruct the netmiko to perform device detection using the `SSHDetect()` class. The class will connect to the device using SSH and will execute some discovery commands to define the operating system type. The returned result will be a dictionary, and the best match will be assigned to the `device_type` variable using the `autodetect()` function.
- You can see all the matching results by printing the `potential_matches`.
- Now we can update the device dictionary and assign the new `device_type` to it.

Using the telnet protocol in Python

Telnet is one of the oldest protocols available in the TCP/IP stack. It is used primarily to exchange data over an established connection between a server and client. It uses TCP port `23` in the server for listening to the incoming connection from the client.

In our case, we will create a Python script that acts as a telnet client, and other routers and switches in the topology will act as the telnet server. Python comes with a native support for telnet via a library called `telnetlib` so we don't need to install it.

After creating the client object by instantiating it from the `Telnet()` class, available from the `telnetlib` module, we can use the two important functions available inside `telnetlib`, which are `read_until()` (used to read the output) and `write()` (used to write on the remote device). Both functions are used to interact with the created channel, either by writing or reading the output returned from it.

Also, it's important to note that reading the channel using `read_until()` will clear the buffer and data won't be available for any further reading. So, if you read important data and you will process and work on it later, then you need to save it as a variable before you continue with your script.

Telnet data is sent in clear text format, so your credentials and password may be captured and viewed by anyone performing a man-in-the-middle attack. Some service providers and enterprises still use it and integrate it with VPNs and radius/tacacs protocols to provide lightweight and secure access.

Follow the steps to understand the whole script:

1. We will import the `telnetlib` module inside our Python script and define the username and passwords in variables, as in the following code snippet:

```
import telnetlib
username = "admin"
password = "access123"
enable_password = "access123"
```

2. We will define a variable that establishes the connection with the remote host. Note that we won't provide the username or password during connection establishment; we will only provide the IP address of the remote host:

```
cnx = telnetlib.Telnet(host="10.10.88.110") #here we're telnet to
Gateway
```

3. Now we will provide the username for the telnet connection by reading the returned output from the channel and searching for the `Username:` keyword. Then we write our admin username. The same process is used when we need to enter the telnet password and enable password:

```
cnx.read_until("Username:")
cnx.write(username + "\n")
cnx.read_until("Password:")
cnx.write(password + "\n")
cnx.read_until(">")
```

```
cnx.write("en" + "\n")
cnx.read_until("Password:")
cnx.write(enable_password + "\n")
```

It's important to provide the exact keywords that appear in the console when you establish the telnet connection or the connection, will enter an infinite loop. Then Python script will be timed out with an error.

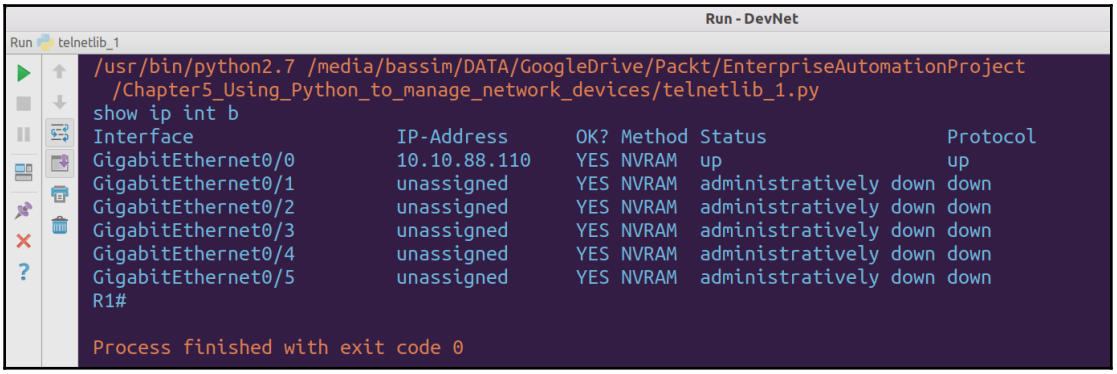
4. Finally, we will write the `show ip interface brief` command on the channel and read till the router prompt `#` to get the output. This should get us the interface configuration in the router:

```
cnx.read_until("#")
cnx.write("show ip int b" + "\n")
output = cnx.read_until("#")
print output
```

The full script is:

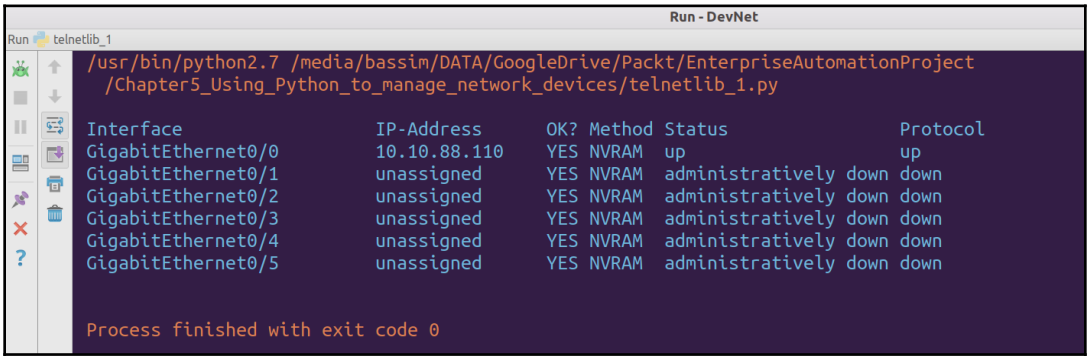
```
1  __author__ = "Bassim Aly"
2  __EMAIL__ = "basim.alyy@gmail.com"
3
4  import telnetlib
5  username = "admin"
6  password = "access123"
7  enable_password = "access123"
8  cnx = telnetlib.Telnet(host="10.10.88.110")
9  cnx.read_until("Username:")
10 cnx.write(username + "\n")
11 cnx.read_until("Password:")
12 cnx.write(password + "\n")
13 cnx.read_until(">")
14 cnx.write("en" + "\n")
15 cnx.read_until("Password:")
16 cnx.write(enable_password + "\n")
17 cnx.read_until("#")
18 cnx.write("show ip int b" + "\n")
19 output = cnx.read_until("#")
20 print output
```

The script output is:



Notice that the output contains the executed command `show ip int b`, and the router prompt "R1#" is returned and printed in the `stdout`. We could use built-in string functions like `replace()` to clean them from the output:

```
cleaned_output = output.replace("show ip int b","").replace("R1#","")
print cleaned_output
```



As you noticed, we provided both the password and enable password as clear text inside our script, which is considered a security issue. It's also not good practice to hardcode the values inside your Python script. Later, in the next section, we will hide the password and design a mechanism to provide credentials during script runtime only.

Also, if you want to execute commands that span multiple pages in output like `show running config` then you will need to disable paging first by sending `terminal length 0` after connecting to the device and before sending the command to it.

Push configuration using telnetlib

In previous section, we looked at a simplified operation of `telnetlib` by executing the `show ip int brief`. Now we need to utilize it to push VLAN configuration to the four switches in our topology. We could create a VLAN list using the python `range()` function and iterate over it to push the VLAN ID to the current switch. Notice we defined the switch IP addresses as an item inside the list, and this list will be our outer `for` loop. Also, I will use another built-in module called `getpass` to hide the password from the console and only provide it when the script is running:

```
#!/usr/bin/python
import telnetlib
import getpass
import time

switch_ips = ["10.10.88.111", "10.10.88.112", "10.10.88.113",
"10.10.88.114"]
username = raw_input("Please Enter your username:")
password = getpass.getpass("Please Enter your Password:")
enable_password = getpass.getpass("Please Enter your Enable Password:")

for sw_ip in switch_ips:
    print "\n#################### Working on Device " + sw_ip + "
####################"
    connection = telnetlib.Telnet(host=sw_ip.strip())
    connection.read_until("Username:")
    connection.write(username + "\n")
    connection.read_until("Password:")
    connection.write(password + "\n")
    connection.read_until(">")
    connection.write("enable" + "\n")
    connection.read_until("Password:")
    connection.write(enable_password + "\n")
    connection.read_until("#")
    connection.write("config terminal" + "\n")  # now i'm in config mode
    vlans = range(300,400)
    for vlan_id in vlans:
        print "\n********* Adding VLAN " + str(vlan_id) + "**********"
        connection.read_until("#")
        connection.write("vlan " + str(vlan_id) + "\n")
        time.sleep(1)
        connection.write("exit" + "\n")
        connection.read_until("#")
    connection.close()
```

In our outermost `for` loop, we are iterating over the devices and then, inside each iteration (each device), we're generating a vlan range from 300 to 400 and pushing them to the current device.

The script output is:

```
bassim@me-inside:~$ /usr/bin/python2.7 /media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProje
ct/Chapter5_Using_Python_to_manage_network_devices/telnetlib_push_vlans.py
Please Enter your username:admin
Please Enter your Password:
Please Enter your Enable Password:

################### Working on Device 10.10.88.111 ####################

********* Adding VLAN 300**********

********* Adding VLAN 301**********

********* Adding VLAN 302**********

********* Adding VLAN 303**********

********* Adding VLAN 304**********

********* Adding VLAN 305**********

********* Adding VLAN 306**********

********* Adding VLAN 307**********

********* Adding VLAN 308**********
```

Also, you can check the output from the switch console itself (output is omitted):

```
SW1#show vlan

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/1, Gi0/2, Gi0/3, Gi1/0
                                                Gi1/1, Gi1/2, Gi1/3, Gi2/0
                                                Gi2/1, Gi2/2, Gi2/3, Gi3/0
                                                Gi3/1, Gi3/2, Gi3/3
300  VLAN0300                         active
301  VLAN0301                         active
302  VLAN0302                         active
303  VLAN0303                         active
304  VLAN0304                         active
305  VLAN0305                         active
306  VLAN0306                         active
307  VLAN0307                         active
308  VLAN0308                         active
309  VLAN0309                         active
310  VLAN0310                         active
311  VLAN0311                         active
312  VLAN0312                         active
313  VLAN0313                         active
314  VLAN0314                         active
315  VLAN0315                         active
316  VLAN0316                         active
317  VLAN0317                         active
```

Handling IP addresses and networks with netaddr

Working and manipulating IP addresses is one of the most important tasks for network engineers. Python developers provide an amazing library that can understand the IP addresses and work on them, called `netaddr`. For example, assume you developed an application and part of it is to get the network and broadcast address for `129.183.1.55/21`. You can do that easily via two built-in methods inside the modules called `network` and `broadcast` respectively:

```
net.network
129.183.0.
net.broadcast
129.183.0.0
```

In general, netaddr provides support for the following features:

Layer 3 addresses:

- IPv4 and IPv6 addresses, subnets, masks, prefixes
- Iterating, slicing, sorting, summarizing, and classifying IP networks
- Dealing with various range formats (CIDR, arbitrary ranges and globs, nmap)
- Set-based operations (unions, intersections, and so on) over IP addresses and subnets
- Parsing a large variety of different formats and notations
- Looking up IANA IP block information
- Generating DNS reverse lookups
- Supernetting and subnetting

Layer 2 addresses:

- Representation and manipulation MAC addresses and EUI-64 identifiers
- Looking up IEEE organisational information (OUI, IAB)
- Generating derived IPv6 addresses

Netaddr installation

The netaddr module can be installed using pip, as shown in the following command:

```
pip install netaddr
```

As a verification for successfully installing the module, you could open PyCharm or the Python console and import the module after installation. If there is no error produced, then the module installed successfully:

```
python
>>>import netaddr
```

Exploring netaddr methods

The `netaddr` module provides two important methods to define the IP address and work on it. The first one is called `IPAddress()` and it's used to define a single classful IP address with the default subnet mask. The second method is `IPNetwork()` and is used to define classless a IP address with CIDR.

Both methods take the IP address as a string and return an IP address or IP network object for this string. There are many operations that could be executed on the returned object. For example, we can check if the IP address is unicast, multicast, loopback, private, public, or even valid or not valid. The output of the previous operation is either `True` or `False`, which can be used inside Python `if` conditions.

Also, the module supports comparison operations such as ==, <, and > to compare two IP addresses, generating the subnets, and it is also possible to retrieve the list of supernets that a given IP address or subnet belongs to. Finally, the `netaddr` module can generate a full list of valid hosts (excluding the network IP and network broadcast):

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"
from netaddr import IPNetwork,IPAddress
def check_ip_address(ipaddr):
    ip_attributes = []
    ipaddress = IPAddress(ipaddr)

    if ipaddress.is_private():
        ip_attributes.append("IP Address is Private")
    else:
        ip_attributes.append("IP Address is public")
```

```
        if ipaddress.is_unicast():
            ip_attributes.append("IP Address is unicast")
        elif ipaddress.is_multicast():
            ip_attributes.append("IP Address is multicast")
        if ipaddress.is_loopback():
            ip_attributes.append("IP Address is loopback")

        return "\n".join(ip_attributes)

    def operate_on_ip_network(ipnet):

        net_attributes = []
        net = IPNetwork(ipnet)
        net_attributes.append("Network IP Address is " + str(net.network) + "
    and Netowrk Mask is " + str(net.netmask))

        net_attributes.append("The Broadcast is " + str(net.broadcast) )
        net_attributes.append("IP Version is " + str(net.version) )
        net_attributes.append("Information known about this network is " +
    str(net.info) )
        net_attributes.append("The IPv6 representation is " + str(net.ipv6()))
        net_attributes.append("The Network size is " + str(net.size))
        net_attributes.append("Generating a list of ip addresses inside the
    subnet")

        for ip in net:
            net_attributes.append("\t" + str(ip))
        return "\n".join(net_attributes)

    ipaddr = raw_input("Please Enter the IP Address: ")
    print check_ip_address(ipaddr)

    ipnet = raw_input("Please Enter the IP Network: ")
    print operate_on_ip_network(ipnet)
```

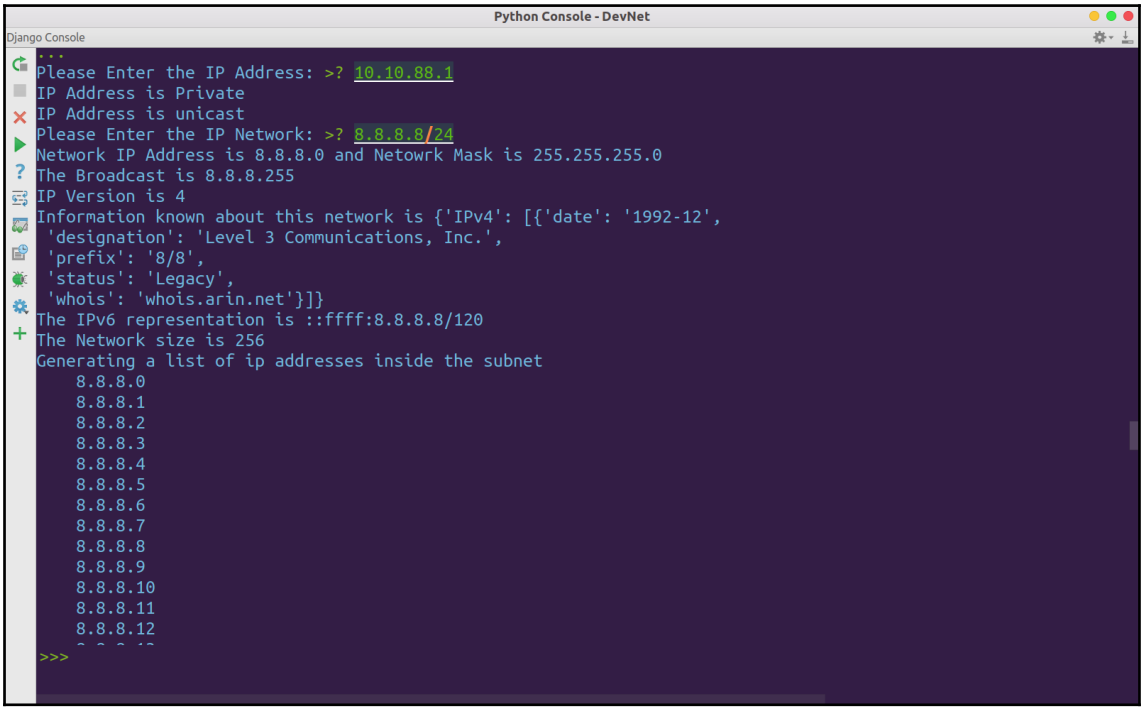
The preceding script first requests the IP address and IP network from the user, using the `raw_input()` function, then will call two user methods, `check_ip_address()` and `operate_on_ip_network()`, and pass the entered values to them. The first function, `check_ip_address()`, will check the IP address entered and try to generate a report about IP address attributes, such as whether it is a unicast IP, multicast, private, or loopback, and will return the output to the user.

The second function `operate_on_ip_network()` takes the IP network and generates the network ID, netmask, broadcast, version, information known about this network, the IPv6 representation, and finally generates all IP addresses inside this subnet.

It's important to notice that `net.info` will work and generate useful information only for public IP addresses, not private.

Notice we need to import the `IP Network` and `IP Address` from the `netaddr` module before using them.

The script output is:



Sample use cases

As our network becomes bigger and starts to contain many devices from different vendors, we need to create modular Python script to automate various tasks in it. In the following sections, we will explore three use cases, which could be used to collect different information from our network and to lower the time needed for troubleshooting a problem, or at least restore the network configuration to its last known good state. This will allow network engineers to focus more on getting their job done and will provide an automated workflow for the business to handle network failure and restoration.

Backup device configuration

Backup device configuration is one of the most important tasks for any network engineer. In this use case, we will design a sample python script that can be used for different vendors and platforms in order to back up the device configuration. We will leverage the `netmiko` library to do this task.

The result files should be formatted with the device IP address in them for easy access or referencing later. For example, the result file for the SW1 backup operation should be `dev_10.10.88.111_.cfg`.

Building the python script

We will start by defining our switches. We want to back up their configuration as a text file and provide the credentials and access details separated by commas. This will allow us to use the `split()` function inside the python script to get the data and use it inside the `ConnectHandler` function. Also, the file can be easily exported and imported from a Microsoft Excel sheet or from any database.

The file structure is:

```
<device_ipaddress>,<username>,<password>,<enable_password>,<vendor>
```

Now we will start building our Python script by importing the file inside it, using the `with open` clause. We use the `readlines()` on the file to have each line as an item inside a list. We will create a `for` loop to iterate over each line and use the `split()` function to get the access details separated by commas and assign them to variables:

```
from netmiko import ConnectHandler
from datetime import datetime

with
open("/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject/Chap
ter5_Using_Python_to_manage_network_devices/UC1_devices.txt") as
devices_file:
```

```
    devices = devices_file.readlines()

for line in devices:
    line = line.strip("\n")
    ipaddr = line.split(",")[0]
    username = line.split(",")[1]
    password = line.split(",")[2]
    enable_password = line.split(",")[3]

    vendor = line.split(",")[4]

    if vendor.lower() == "cisco":
        device_type = "cisco_ios"
        backup_command = "show running-config"

    elif vendor.lower() == "juniper":
        device_type = "juniper"
        backup_command = "show configuration | display set"
```

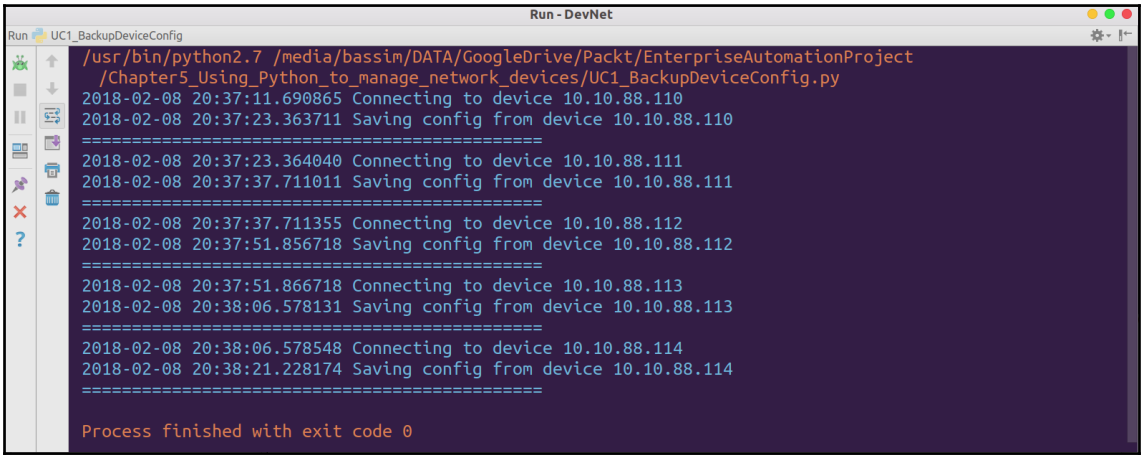
As we need to create a modular and multi-vendor script, we need to have the `if` clause check the vendor in each line and assign a correct `device_type` and `backup_command` to the current device.

Moving on, we are now ready to establish the SSH connection to the device and execute the backup command on it using the `.send_command()` method available inside the `netmiko` module:

```
print str(datetime.now()) + " Connecting to device {}" .format(ipaddr)

net_connect = ConnectHandler(device_type=device_type,
                             ip=ipaddr,
                             username=username,
                             password=password,
                             secret=enable_password)
net_connect.enable()
running_config = net_connect.send_command(backup_command)

print str(datetime.now()) + " Saving config from device {}" .format(ipaddr)

f = open( "dev_" + ipaddr + "_.cfg", "w")
f.write(running_config)
f.close()
print "============================================="
```

In the last few statements, we opened a file for writing and made its name contain the `ipaddr` variable collected from our text file.

The script output is:



Also, notice the backup configuration files are created in the project home directory, and its name contains the IP address of each device:

```
bassim@me-inside:portal$ ls dev*
dev_10.10.88.110_.cfg  dev_10.10.88.112_.cfg  dev_10.10.88.114_.cfg
dev_10.10.88.111_.cfg  dev_10.10.88.113_.cfg
bassim@me-inside:portal$
bassim@me-inside:portal$ more dev_10.10.88.110_.cfg
Building configuration...


Current configuration : 3994 bytes
!
version 15.6
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
!
hostname R1
!
boot-start-marker
boot-end-marker
!
!
enable password access123
!
aaa new-model
!
!
aaa authentication login default local
```

You can design a simple cron job on a Linux server or schedule a job on a Windows server, which runs the previous python script at a specific time. For example, the script could run on a daily basis at midnight and store the configuration in the `latest` directory so the team could refer to it later.

Creating your own access terminal

In Python, and programming in general, you are the vendor! You can create any code combination and procedures you like in order to serve your needs. In the second use case, we will create our own terminal that accesses the router through `telnetlib`. By writing a few words in the terminal, it will be translated too many commands executed in the network device and return output, which could be just printed in the standard output or saved in file:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import telnetlib

connection = telnetlib.Telnet(host="10.10.88.110")
connection.read_until("Username:")
connection.write("admin" + "\n")
connection.read_until("Password:")
connection.write("access123" + "\n")
connection.read_until(">")
connection.write("en" + "\n")
connection.read_until("Password:")
connection.write("access123" + "\n")
connection.read_until("#")
connection.write("terminal length 0" + "\n")
connection.read_until("#")
while True:
    command = raw_input("#:")
    if "health" in command.lower():
        commands = ["show ip int b",
                    "show ip route",
                    "show clock",
                    "show banner motd"
                    ]

    elif "discover" in command.lower():
        commands = ["show arp",
```

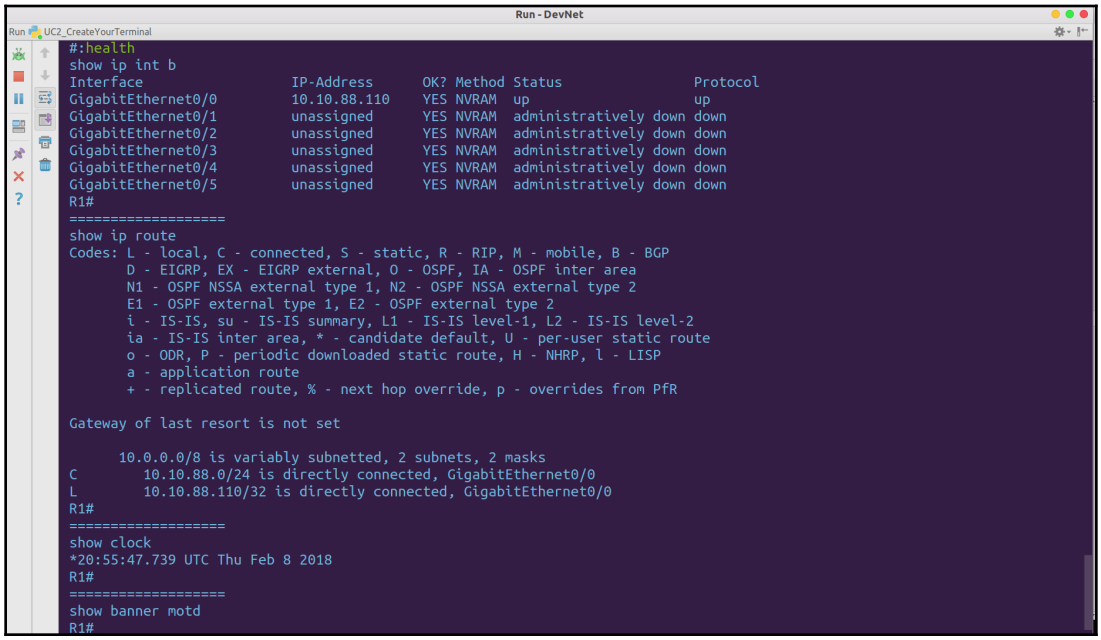
```
                "show version | i uptime",
                "show inventory",
                ]
else:
    commands = [command]
for cmd in commands:
    connection.write(cmd + "\n")
    output = connection.read_until("#")
    print output
    print "===================="
```

First, we establish a telnet connection to the router and enter the user access details till we reach enable mode. Then we create an infinite `while` loop that is always `true`, and we expect a command from the user using the `raw_input()` built-in function. When the user enters any command, the script will capture it and execute it directly to the network device.

However, if the user enters `health` or `discover` keywords then our terminal will be smart enough to execute a series of commands to reflect the desired operation. This should be extremely useful in case of network troubleshooting, and you can extend it with any daily operation. Imagine that you need to troubleshoot OSPF neighbourship problems between two routers. You just need to open your own terminal python script that you already taught him few commands needed for troubleshooting, and write something like `tshoot_ospf`. Once your script sees this magic keyword it will launch a series of multiple commands that print the OSPF neighborship status, interfaces of MTU, advertised network under OSPF, and so on till you find the issue.

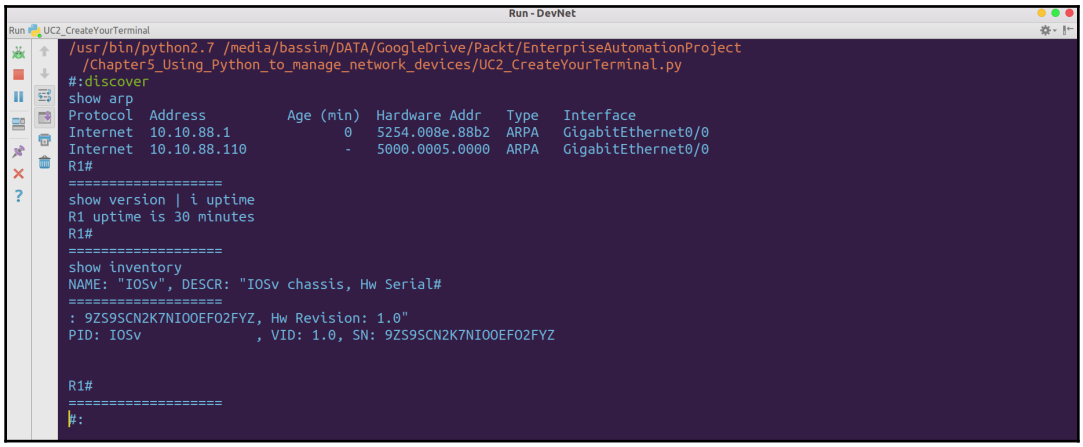
Script output:

Try the first command in our script by writing `health` in the prompt:



As you can see, the script returns the output of multiple commands executed in the device.

Now try the second supported command, `discover`:



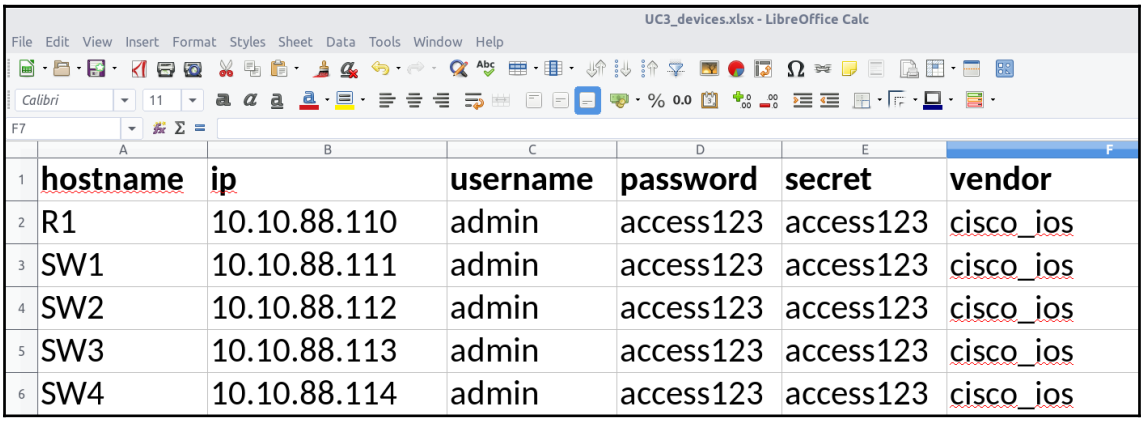
This time the script returns the output of discover commands. In later chapters, we can parse the returned output and extract the useful information from it.

Reading data from an Excel sheet

Network and IT engineers always use the excel sheet to store information about the infrastructure such as IP addresses, the device vendor, and credentials. Python support reading the information from an excel sheet and processes it so you can use it later during the script.

In this use case, we will use the **Excel Read** (**xlrd**) module to read the `UC3_devices.xlsx` file which contains the hostname, IP, username, password, enable password and vendor for our infrastructure and use this information to feed the `netmiko` module.

The Excel sheet will be as shown in the following screenshot:



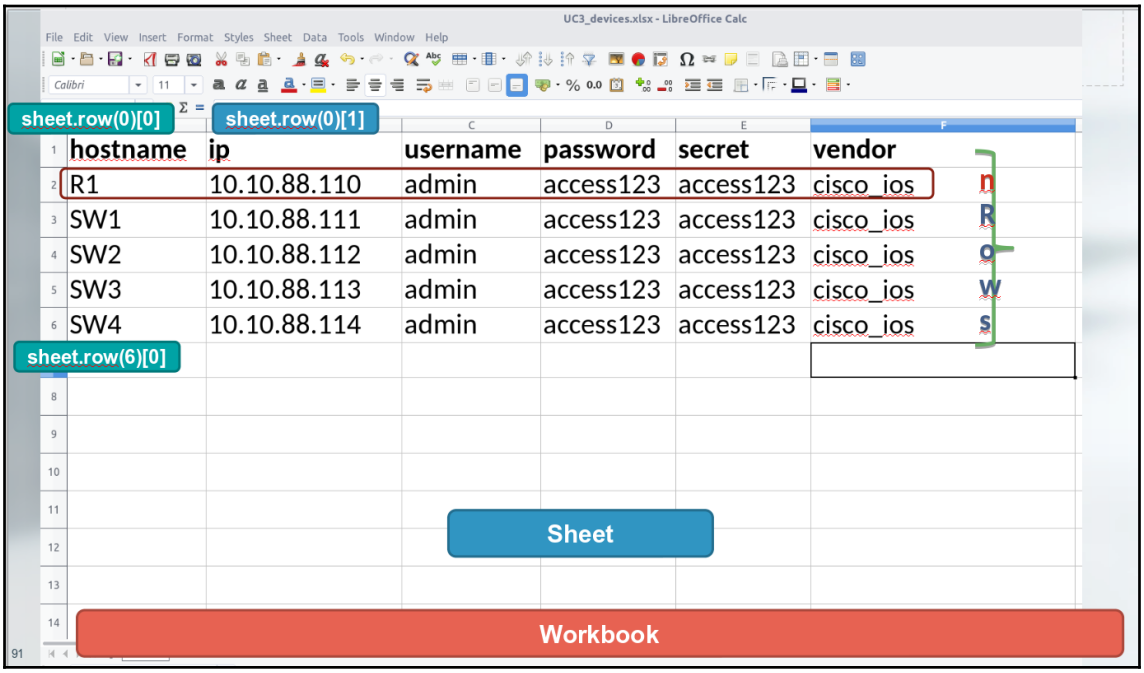
	A	B	C	D	E	F
1	hostname	ip	username	password	secret	vendor
2	R1	10.10.88.110	admin	access123	access123	cisco_ios
3	SW1	10.10.88.111	admin	access123	access123	cisco_ios
4	SW2	10.10.88.112	admin	access123	access123	cisco_ios
5	SW3	10.10.88.113	admin	access123	access123	cisco_ios
6	SW4	10.10.88.114	admin	access123	access123	cisco_ios

First we will need to install the `xlrd` module, using `pip` as we will use it to read the Microsoft excel sheet:

```
pip install xlrd
```

The XLRD module read the excel workbook and convert the row and columns into a matrix. For example, if you need to get the first item on the left, then you will need to access row[0][0]. The next item on the right will be row[0][1] and so on.

Also, when xlrd reads the sheet, it will increase a special counter called `nrows` (number of rows) by one each time it reads a row. Similarly, it will increase the `ncols` (number of columns) by one each time it reads the columns so you can know the size of your matrix via these two parameters:



You can provide the file path to `xlrd` using the `open_workbook()` function. Then you can access your sheet that contains the data either by using `sheet_by_index()` or `sheet_by_name()` functions. For our use case, our data is stored in the first sheet (index=0), and the file path is stored under the chapter name. Then we will iterate over the rows in the sheet and use the `row()` function to access a specific row. The returned output is a list, and we can access any item in it using the index.

Python script:

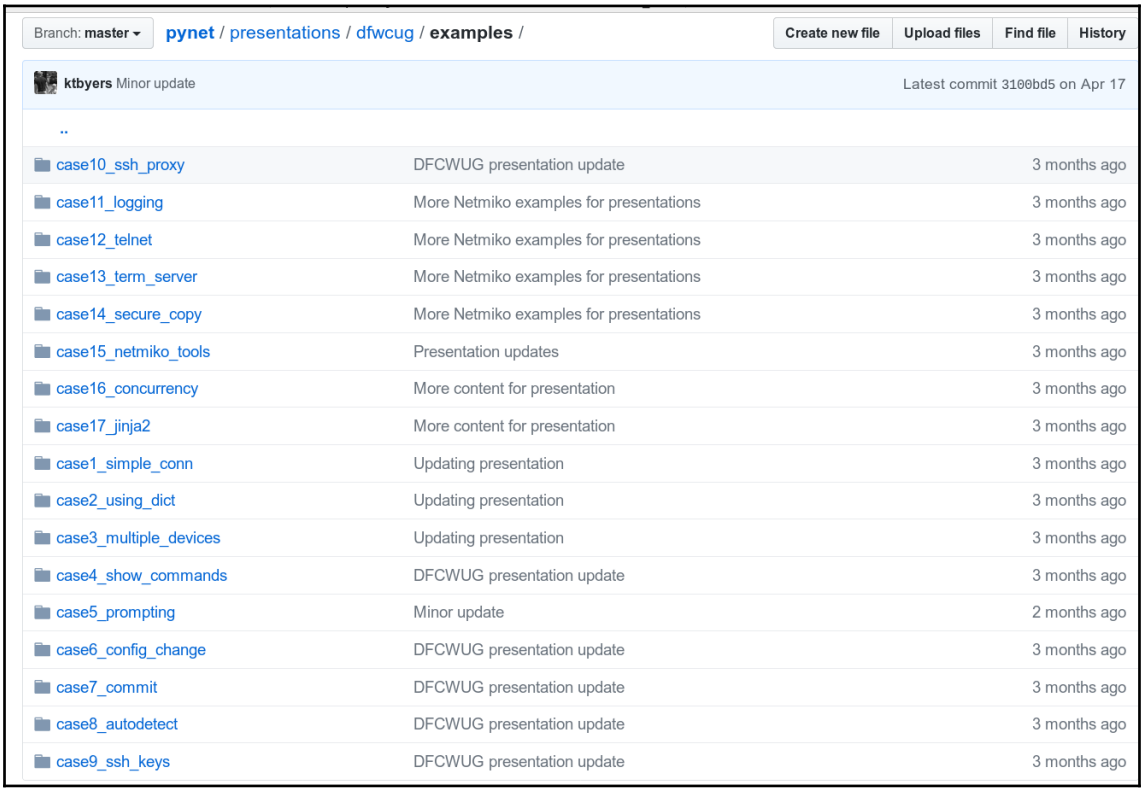
```
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from netmiko import ConnectHandler
from netmiko.ssh_exception import AuthenticationException,
NetMikoTimeoutException
import xlrd
```

```
from pprint import pprint

workbook =
xlrd.open_workbook(r"/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomat
ionProject/Chapter4_Using_Python_to_manage_network_devices/UC3_devices.xlsx
")

sheet = workbook.sheet_by_index(0)

for index in range(1, sheet.nrows):
    hostname = sheet.row(index)[0].value
    ipaddr = sheet.row(index)[1].value
    username = sheet.row(index)[2].value
    password = sheet.row(index)[3].value
    enable_password = sheet.row(index)[4].value
    vendor = sheet.row(index)[5].value

    device = {
        'device_type': vendor,
        'ip': ipaddr,
        'username': username,
        'password': password,
        'secret': enable_password,

    }
    # pprint(device)

    print "########## Connecting to Device {0}
############".format(device['ip'])
    try:
        net_connect = ConnectHandler(**device)
        net_connect.enable()

        print "***** show ip configuration of Device *****"
        output = net_connect.send_command("show ip int b")
        print output

        net_connect.disconnect()

    except NetMikoTimeoutException:
        print "=======SOMETHING WRONG HAPPEN WITH
{0}=======".format(device['ip'])

    except AuthenticationException:
        print "=======Authentication Failed with
{0}=======".format(device['ip'])
```

```
    except Exception as unknown_error:
        print "=======SOMETHING UNKNOWN HAPPEN WITH {0}======="
```

More use cases

Netmiko could be used to realize many network automation use cases. It could be used for uploading, downloading files from remote devices during upgrade, loading configuration from Jinja2 templates, accessing terminal servers, accessing end devices, and many more. You can find a list of some useful use cases at `https://github.com/ktbyers/pynet/tree/master/presentations/dfwcug/examples`:



Branch: master ▾ pynet / presentations / dfwcug / examples / Create new file Upload files Find file History

ktbyers Minor update Latest commit 3100bd5 on Apr 17

..

case10_ssh_proxy	DFCWUG presentation update	3 months ago
case11_logging	More Netmiko examples for presentations	3 months ago
case12_telnet	More Netmiko examples for presentations	3 months ago
case13_term_server	More Netmiko examples for presentations	3 months ago
case14_secure_copy	More Netmiko examples for presentations	3 months ago
case15_netmiko_tools	Presentation updates	3 months ago
case16_concurrency	More content for presentation	3 months ago
case17_jinja2	More content for presentation	3 months ago
case1_simple_conn	Updating presentation	3 months ago
case2_using_dict	Updating presentation	3 months ago
case3_multiple_devices	Updating presentation	3 months ago
case4_show_commands	DFCWUG presentation update	3 months ago
case5_prompting	Minor update	2 months ago
case6_config_change	DFCWUG presentation update	3 months ago
case7_commit	DFCWUG presentation update	3 months ago
case8_autodetect	DFCWUG presentation update	3 months ago
case9_ssh_keys	DFCWUG presentation update	3 months ago

Summary

In this chapter, we started our practical journey into the network automation world with Python. We explored the different tools that are available in python to establish a connection to remote nodes with telnet and SSH and executed commands on them. Also, we learned how to handle IP addresses and network subnets with the help of the `netaddr` module. Finally, we strengthened our knowledge with two practical use cases.

In the next chapter, we will work on the returned output and start to extract useful information from it.

5
Extracting Useful Data from Network Devices

We have already seen in the previous chapter how to access a network device using different methods and protocols, then execute commands on the remote device to get an output back to Python. Now, it's time to extract some useful data from this output.

In this chapter, you'll learn how to use different tools and libraries in Python to extract useful data from returned output and act on it using regular expressions. Also, we will use a special library called `CiscoConfParse` to audit the configuration, then we will learn how to visualize data to generate visually appealing graphs and reports using the `matplotlib` library.

We will cover the following topics in this chapter:

- Understanding parsers
- Introduction to regular expressions
- Configuration auditing using `Ciscoconfparse`
- Visualizing returned data with `matplotlib`

Technical requirements

The following tools should be installed and available in your environment:

- Python 2.7.1x
- PyCharm Community or Pro edition
- EVE-NG lab

You can find the full scripts developed in this chapter at the following GitHub URL:

```
https://github.com/TheNetworker/EnterpriseAutomation.git
```

Understanding parsers

In the previous chapter, we explored different ways to access network devices, execute commands, and return output to our terminal. We now need to work on the returned output and extract some useful information from it. Notice that, from Python's point of view, the output is just a multiline string and Python doesn't differentiate between IP address, interface name, or node hostname because they're all strings. So, the first step is to design and develop our own parser using Python to categorize and differentiate between items based on the important information in the returned output.

After that, you can work on the parsed data and generate graphs that help to visualize or even store them to persistent and external storage or databases.

Introduction to regular expressions

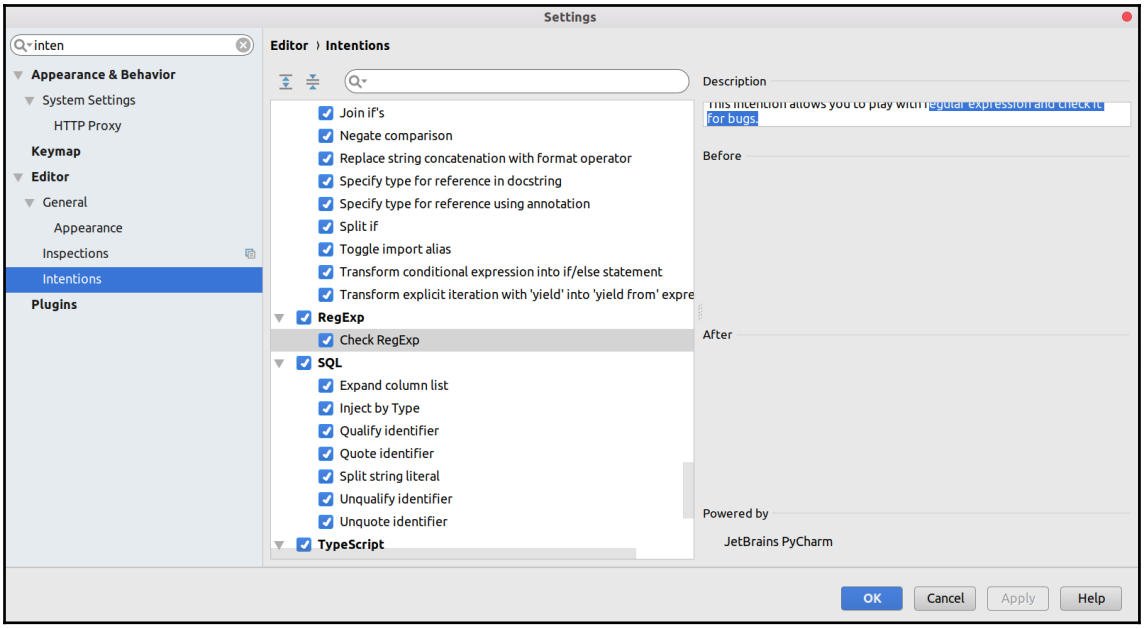
Regular expressions are a language used to match specific occurrences of strings by following their pattern across the whole string. When a match is found, the resulting matched string will be returned back to user and will be held inside a structure in Python format, such as `tuple`, `list`, or `dictionary`. The following table summarizes the most common patterns in regular expressions:

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the *beginning* of the string
abc$	abc at the *end* of the string
a|b	either of a and b
^abc|abc$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does *not* contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\B	perl when *not* followed by a word boundary (e.g. in perlert but not in perl stuff)

Also, one of the important rules in regular expressions is you can write your own regex and surround it with parentheses `()`, which is called the capturing group and helps you to hold important data to reference it later using the capturing group number:

```
line = '30 acd3.b2c6.aac9 FastEthernet0/1'
match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)
print match.group(1)
print match.group(2)
```

PyCharm will automatically color strings written as regular expressions and can help you to check the validity of a regex before applying it to data. Make sure the **Check RegExp** intention is enabled in the settings, as shown here:



Creating a regular expression in Python

You can construct a regular expression in Python using the `re` module that is natively shipped with the Python installation. There are several methods inside this module, such as `search()`, `sub()`, `split()`, `compile()`, and `findall()`, which will return the result as a regex object. Here is a summary of the use of each function:

Function Name	Usage
`search()`	Search and match the first occurrence of the pattern.
`findall()`	Search and match all occurrences of the pattern and return the result as a list.
`Finditer()`	Search and match all occurrences of the pattern and return the result as an iterator.

`compile()`	Compile the regex into a pattern object that has methods for various operations, such as searching for pattern matches or performing string substitutions. This is extremely useful if you use the same regex pattern multiple times inside your script.
`sub()`	Used to replace matched pattern with another string.
`split()`	Used to split on matched pattern and create a list.

Regular expressions are hard to read; for that reason, let's start simple and look at some easy regular expressions at the most basic level.

The first step of working with the `re` module is to import it inside your Python code

```
import re
```

We will start to explore the most common function in the `re` module, which is `search()`, and then we will explore `findall()`. The `search()` function is suitable when you need to find only one match in a string or when you write your regex pattern to match the entire output and need to get the result with a method called `groups()`, as we will see in the following examples.

The syntax of the `re.search()` function is as follows:

```
match = re.search('regex pattern', 'string')
```

The first parameter, `'regex pattern'`, is the regular expression developed in order to match a specific occurrence inside the `'string'`. When a match is found, the `search()` function returns a special match object, otherwise it will return `None`. Note that `search()` will return the first occurrence only of the pattern and will ignore the rest of them. Let's see a few examples of using the `re` module in Python:

Example 1: Searching for a specific IP address

```
import re
intf_ip = 'Gi0/0/0.911            10.200.101.242   YES NVRAM  up
up'
match = re.search('10.200.101.242', intf_ip)

if match:
    print match.group()
```

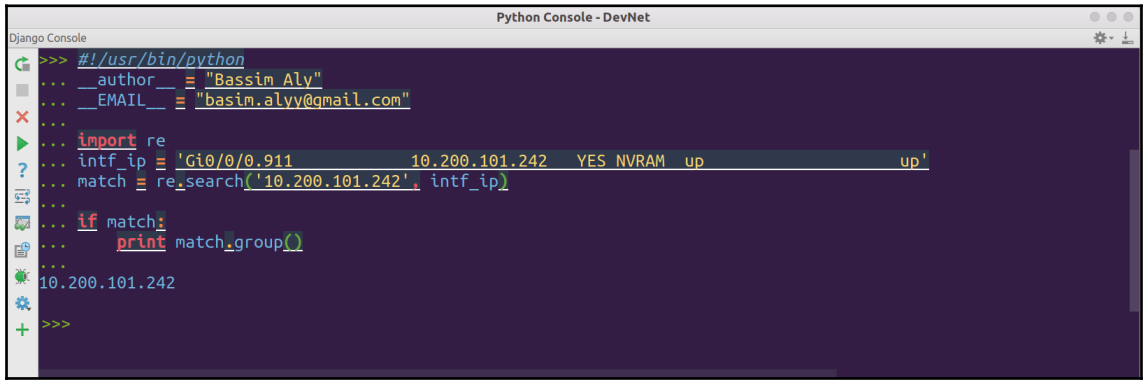
In this example, we can see the following:

- The `re` module is imported into our Python script.
- We have a string that corresponds to interface details and contains the name, IP address, and status. This string could be hardcoded in the script or generated from the network device using the Netmiko library.
- We passed this string to the `search()` function, along with our regular expression, which is just the IP address.
- Then, the script checks whether there's a `match` object returned from the previous operation; if so, it will print it.

The most basic method of testing for a match is via the `re.match` function, as we did in the previous example. The `match` function takes a regular expression pattern and a string value.

Notice we're only searching for a specific string inside the `intf_ip` parameter, not every IP address pattern.

Example 1 output



Example 2: Matching the IP address pattern

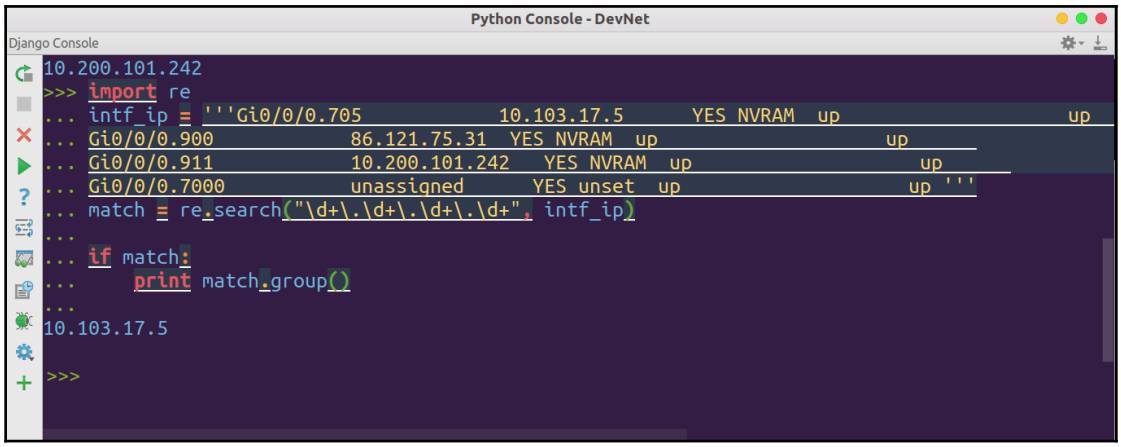
```
import re
intf_ip = '''Gi0/0/0.705            10.103.17.5      YES NVRAM  up
up
Gi0/0/0.900            86.121.75.31  YES NVRAM  up                    up
Gi0/0/0.911            10.200.101.242   YES NVRAM  up                      up
Gi0/0/0.7000           unassigned      YES unset  up                      up
'''
match = re.search("\d+\.\d+\.\d+\.\d+", intf_ip)
```

```
if match:
    print match.group()
```

In this example, we can see the following:

- The `re` module is imported into our Python script.
- We have a multi-line string that corresponds to the interface details and contains the name, IP address, and status.
- We passed this string to the `search()` function along with our regular expression, which is the IP address pattern constructed using both `\d+`, which matches one or more digits, and `\.`, which matches the occurrence of the dot.
- Then, the script checks whether there's a `match` object returned from a previous operation; if so, it will print it. Otherwise, the `None` object is returned.

Example 2 output



Notice the `search()` function returns only the first matched occurrence of the pattern, not all occurrences.

Example 3: Using groups() regular expressions

If you have a long output and you need to extract multiple strings from it, then you could surround the extracted value with `()` and write your regex inside it. This is called a **capturing group** and is used to catch a specific pattern within a long string, as shown in the following snippet:

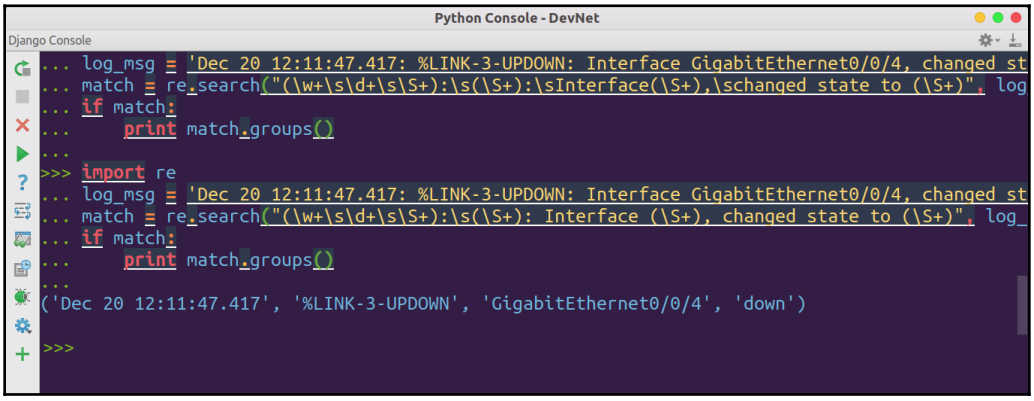
```
import re
log_msg = 'Dec 20 12:11:47.417: %LINK-3-UPDOWN: Interface
```

```
GigabitEthernet0/0/4, changed state to down'
match = re.search("(\w+\s\d+\s\S+):\s(\S+): Interface (\S+), changed state
to (\S+)", log_msg)
if match:
    print match.groups()
```

In this example, we can see the following:

- The `re` module is imported into our Python script.
- We have a string that corresponds to an event that occurred in the router and is stored in logging.
- We passed this string to the `search()` function along with our regular expression. Notice that we enclosed the timestamp, event type, interface name, and the new state of the capturing group and wrote our regex inside it.
- Then, the script checks whether there's a match object returned from the previous operation; if so, it will print it, but this time we used `groups()` instead of `group()`, as we are capturing multiple strings.

Example 3 output



Notice the returned data is in a structured format called a **tuple**. We could use this output later to trigger an event and start, for example, a recovery procedure on a redundant interface.

We could enhance our previous code and use a `Named` group to give each capture group a name that could be referenced later or used to create a dictionary. In this case, we prefixed our regex with `?P<"NAME">` as in the next example (**Example 4** in the GitHub repository):

Example 4: Named group

```
# Example 4: Named group
import re
log_msg = 'Dec 20 12:11:47.417: %LINK-3-UPDOWN: Interface GigabitEthernet0/0/4, changed state to down'
match = re.search("(?P<TIMESTAMP>\w+\s\d+\s\S+):\s(?P<EVENT>\S+): Interface (?P<INTF>\S+), changed state to (?P<STATE>\S+)",
  log_msg)
if match:
    print match.groups()
```

Example 5-1: Searching for multiple lines using re.search()

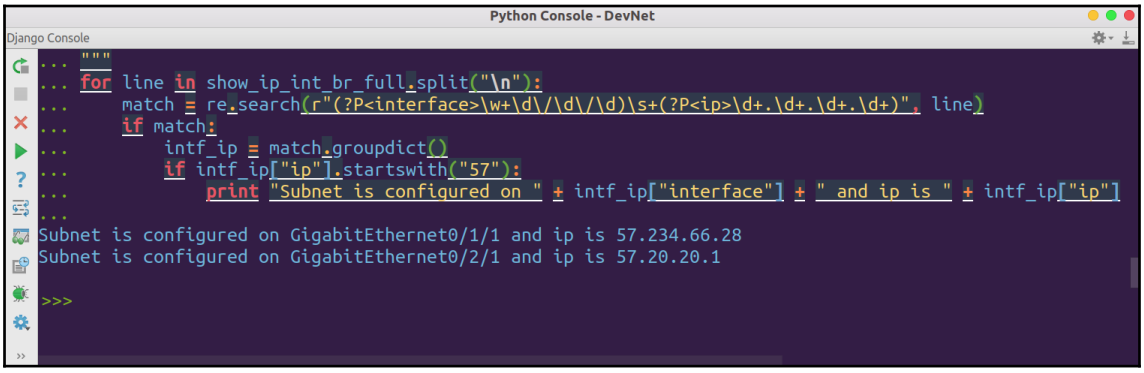
Assume we have multiple lines in the output and we need to check all of them against the regex pattern. Remember that the `search()` function exits when it finds the first pattern match. In that case, we have two solutions. The first one is to feed each line to the search function by splitting the whole string on `"\n"`, and the second solution is to use the `findall()` function. Let's explore the two solutions:

```
import re

show_ip_int_br_full = """
GigabitEthernet0/0/0        110.110.110.1   YES NVRAM  up
up
GigabitEthernet0/0/1        107.107.107.1   YES NVRAM  up
up
GigabitEthernet0/0/2        108.108.108.1   YES NVRAM  up
up
GigabitEthernet0/0/3        109.109.109.1   YES NVRAM  up
up
GigabitEthernet0/0/4   unassigned      YES NVRAM  up                    up
GigabitEthernet0/0/5             10.131.71.1     YES NVRAM  up
up
GigabitEthernet0/0/6          10.37.102.225   YES NVRAM  up
up
GigabitEthernet0/1/0            unassigned      YES unset  up
up
GigabitEthernet0/1/1           57.234.66.28   YES manual up
up
GigabitEthernet0/1/2           10.10.99.70   YES manual up
up
GigabitEthernet0/1/3           unassigned      YES manual deleted
```

```
down
GigabitEthernet0/1/4           192.168.200.1   YES manual up
up
GigabitEthernet0/1/5   unassigned      YES manual down
down
GigabitEthernet0/1/6         10.20.20.1      YES manual down
down
GigabitEthernet0/2/0         10.30.40.1      YES manual down
down
GigabitEthernet0/2/1         57.20.20.1      YES manual down
down

"""
for line in show_ip_int_br_full.split("\n"):
    match =
re.search(r"(?P<interface>\w+\d\/\d\/\d)\s+(?P<ip>\d+.\d+.\d+.\d+)", line)
    if match:
        intf_ip = match.groupdict()
        if intf_ip["ip"].startswith("57"):
            print "Subnet is configured on " + intf_ip["interface"] + " and
ip is " + intf_ip["ip"]
```

The preceding script will split the `show ip interface brief` output and search for a specific pattern, which is the interface name and the IP address configured on it. Based on the matched data, the script will continue to check each IP address and validate it using `start with 57`, then the script will print the corresponding interface and the full IP address.

Example 5-1 output



If you're searching only for the first occurrence, you can optimize the script and only get the first result by breaking the outer `for` loop upon locating the first match, but note that the second match won't be located or printed.

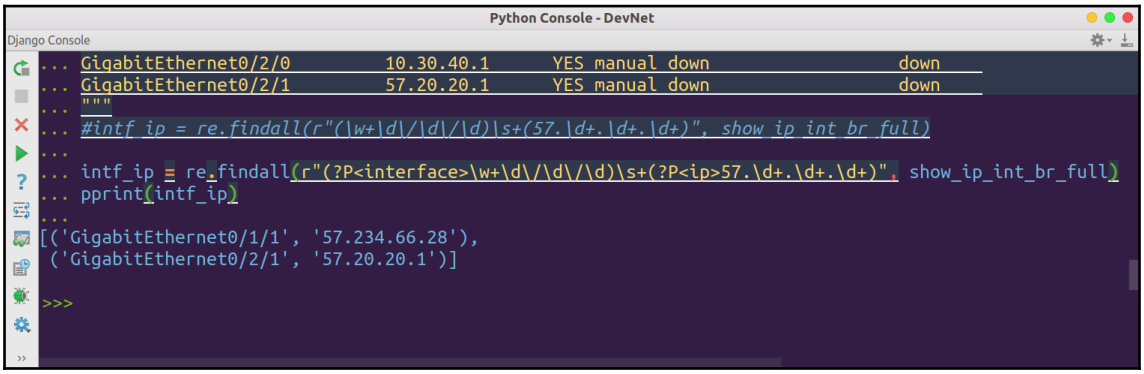
Example 5-2: Searching for multiple lines using re.findall()

The `findall()` function searches for all non-overlapping matches in the provided string and returns a list of strings (unlike the `search` function, which returns the `match` object) that matched by regex pattern if there's no capturing group. If you enclosed your regex with a capturing group, then `findall()` will return a list of tuples. In the following script, we have the same multi-line output and we will use the `findall()` method to get all interfaces that are configured with an IP address that starts with 57:

```
import re
from pprint import pprint
show_ip_int_br_full = """
GigabitEthernet0/0/0        110.110.110.1   YES NVRAM  up
up
GigabitEthernet0/0/1        107.107.107.1   YES NVRAM  up
up
GigabitEthernet0/0/2        108.108.108.1   YES NVRAM  up
up
GigabitEthernet0/0/3        109.109.109.1   YES NVRAM  up
up
GigabitEthernet0/0/4   unassigned      YES NVRAM  up                    up
GigabitEthernet0/0/5             10.131.71.1     YES NVRAM  up
up
GigabitEthernet0/0/6          10.37.102.225   YES NVRAM  up
up
GigabitEthernet0/1/0            unassigned      YES unset  up
up
GigabitEthernet0/1/1           57.234.66.28   YES manual up
up
GigabitEthernet0/1/2           10.10.99.70   YES manual up
up
GigabitEthernet0/1/3           unassigned      YES manual deleted
down
GigabitEthernet0/1/4           192.168.200.1   YES manual up
up
GigabitEthernet0/1/5   unassigned      YES manual down
down
GigabitEthernet0/1/6         10.20.20.1      YES manual down
down
GigabitEthernet0/2/0         10.30.40.1      YES manual down
down
```

```
GigabitEthernet0/2/1          57.20.20.1      YES manual down
down
"""

intf_ip =
re.findall(r"(?P<interface>\w+\d\/\d\/\d)\s+(?P<ip>57.\d+.\d+.\d+)",
show_ip_int_br_full)
pprint(intf_ip)
```

Example 5-2 output:



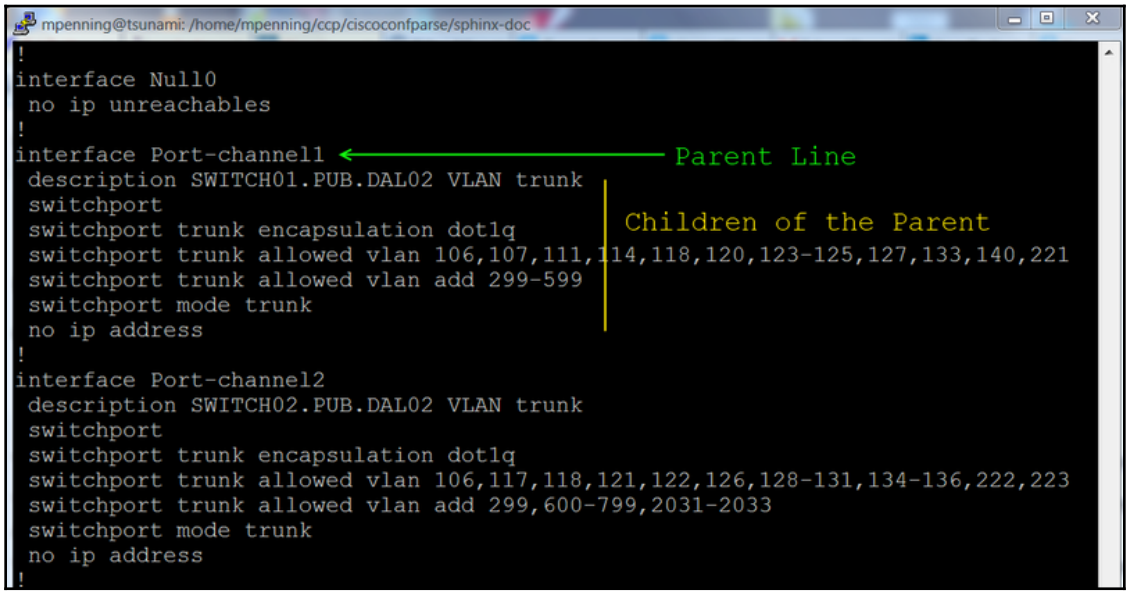
Notice this time we didn't have to write a `for` loop to check each line against the regex pattern. This will be done automatically in the `findall()` method.

Configuration auditing using CiscoConfParse

Applying regular expressions on network configuration to get specific information from the output requires us to write some complex expressions to solve some complex use cases. In some cases, you just need to retrieve some configuration or modify an existing one without going deeply into writing regular expressions, and that was the reason for the birth of the `CiscoConfParse` library (`https://github.com/mpenning/ciscoconfparse`).

CiscoConfParse library

As the official GitHub page says, the library examines an iOS-style config and breaks it into a set of linked parent/child relationships. You can perform complex queries on these relationships:



Source: `https://github.com/mpenning/ciscoconfparse`

So, the first line of the configuration is considered the parent, while the subsequent lines are considered the children of the parent. The `CiscoConfparse` library builds the relationship between parent and child into an object so the end user can easily retrieve the configuration of a specific parent without the need to write complex expressions.

It's extremely important that your configuration file is well-formatted in order to build the correct relationship between the parent and child.

The same concept also applies if you need to inject configuration into the file. The library will search for the given parent and will insert the configuration just under it and save it to the new file. This is helpful in case you need to run a config audit job on multiple files and make sure they all have a consistent configuration.

Supported vendors

As a rule of thumb, any file that has a tab-delimited configuration can be parsed by `CiscoConfParse` and it will build the parent and child relationship.

The following is the list of supported vendors:

- Cisco IOS, Cisco Nexus, Cisco IOS-XR, Cisco IOS-XE, Aironet OS, Cisco ASA, Cisco CatOS
- Arista EOS
- Brocade
- HP switches
- Force10 switches
- Dell PowerConnect switches
- Extreme Networks
- Enterasys
- ScreenOS

Also, starting from version 1.2.4, `CiscoConfParse` can handle the curly braces delimited configuration, which means it can handle the following vendors:

- Juniper Network's Junos OS
- Palo Alto Networks firewall configurations
- F5 Networks configurations

CiscoConfParse installation

`CiscoConfParse` can be installed by using `pip` on the Windows command line or Linux shell:

```
pip install ciscoconfparse
```

```
bassim@me-inside:~$ pip install ciscoconfparse
Collecting ciscoconfparse
  Downloading ciscoconfparse-1.3.1-py2-none-any.whl (85kB)
    100% |████████████████████████████████| 92kB 183kB/s
Collecting colorama (from ciscoconfparse)
  Using cached colorama-0.3.9-py2.py3-none-any.whl
Collecting ipaddr>=2.1.11 (from ciscoconfparse)
  Downloading ipaddr-2.2.0.tar.gz
Collecting dnspython (from ciscoconfparse)
  Downloading dnspython-1.15.0-py2.py3-none-any.whl (177kB)
    100% |████████████████████████████████| 184kB 263kB/s
Building wheels for collected packages: ipaddr
  Running setup.py bdist_wheel for ipaddr ... done
  Stored in directory: /home/bassim/.cache/pip/wheels/3a/75/ef/8677a26e72d7fee90f46b1cb9d8cfd
c0ffe9c738dfd22a54e5
Successfully built ipaddr
Installing collected packages: colorama, ipaddr, dnspython, ciscoconfparse
Successfully installed ciscoconfparse-1.3.1 colorama-0.3.9 dnspython-1.15.0 ipaddr-2.2.0
bassim@me-inside:~$
```

Notice that some additional dependencies are also installed, such as `ipaddr`, `dnsPython`, and `colorama`, which are used by `CiscoConfParse`.

Working with CiscoConfParse

The first example that we will work on is extracting the shutdown interfaces from a sample Cisco configuration located in a file named `Cisco_Config.txt`.

```
from ciscoconfparse import CiscoConfParse
from pprint import pprint

# Find All shutdown interfaces.

orig_config = CiscoConfParse("media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject
  /Chapter5_Extract_useful_data_from_network_devices/Cisco_Config.txt")

shutdown_intfs = orig_config.find_parents_w_child(parentspec=r"^interface",childspec='shutdown')
pprint(shutdown_intfs)
```

In this example, we can see the following:

- From the `CiscoConfParse` module, we imported the `CiscoConfParse` class. Also, we imported the `pprint` module to print the output in readable format to fit the Python console output.
- Then, we provided the `config` file full path to the `CiscoConfParse` class.

- The final step is to use one of the built-in functions such as `find_parents_w_child()` and provide two parameters. The first one is the parent specification, which is searching for anything starting with the `interface` keyword, while the child specification has the `shutdown` keyword.

As you can see, in three simple steps, we were able to get all interfaces that have the shutdown keyword inside and output as a structured list.

Example 1 output

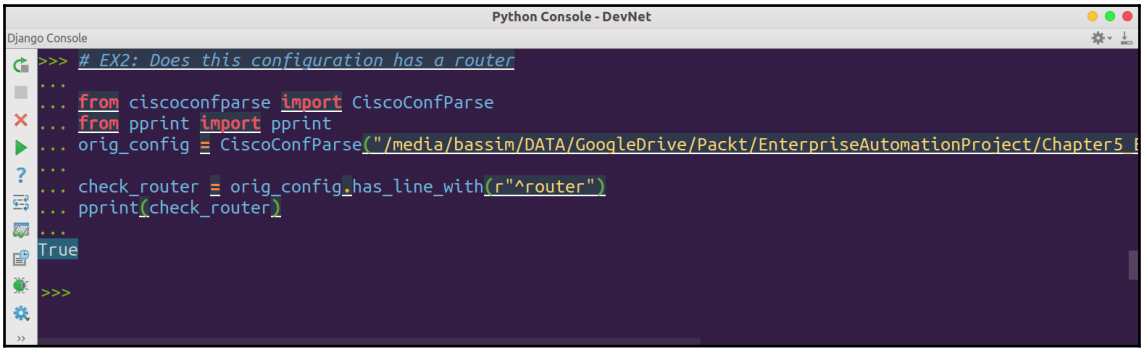
Example 2: Checking the existing of a specific feature

The second example will check whether the router keyword exists within the configuration file as an indication of whether a routing protocol, such as `ospf` or `bgp` is enabled or not. If the module finds it, then the result will be `True`. Otherwise, it will be `False`. This can be achieved by a built-in function within a module called `has_line_with()`:

```
# EX2: Does this configuration has a router

from ciscoconfparse import CiscoConfParse
from pprint import pprint
orig_config = CiscoConfParse("/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject
  /Chapter5_Extract_useful_data_from_network_devices/Cisco_Config.txt" )

check_router = orig_config.has_line_with(r"^router")
pprint(check_router)
```

This method can be used to design a condition inside an `if` statement, as we will see in the next and final example.

Example 2 output



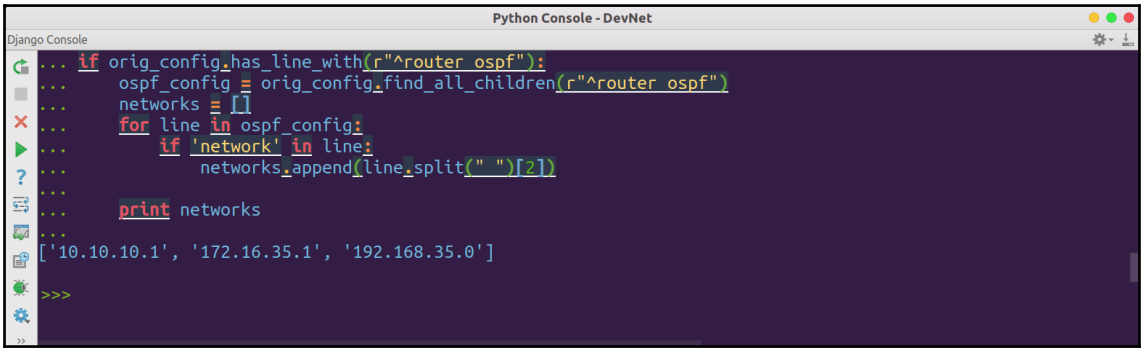
Example 3: Printing specific children from a parent:

```
#EX3: Does OSPF enabled? if yes then find advertised networks

from ciscoconfparse import CiscoConfParse
from pprint import pprint
orig_config = CiscoConfParse("/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject
  /Chapter5_Extract_useful_data_from_network_devices/Cisco_Config.txt")


if orig_config.has_line_with(r"^router ospf"):
    ospf_config = orig_config.find_all_children(r"^router ospf")
    networks = []
    for line in ospf_config:
        if 'network' in line:
            networks.append(line.split(" ")[2])

    print networks
```

In this example, we can see the following:

- From the `CiscoConfParse` module, we imported the `CiscoConfParse` class. Also, we imported the `pprint` module to print the output in readable format to fit the Python console output.
- Then, we provided the `config` file full path to the `CiscoConfParse` class.
- We used one of the built-in functions, such as `find_all_children()`, and provided only the parent. This will instruct the `CiscoConfParse` class to list all configuration lines under this parent.
- Finally, we iterated over the returned output (remember, it's a list) and checked whether the network keyword exists within the string. If yes, then it will append it to the network list, which will be printed at the end.

Example 3 output:



There're many other functions available inside the `CiscoConfParse` module that could be used to easily extract data from the configuration file and return the output in a structured format. Here is a list of other functions:

- `find_lineage`
- `find_lines()`
- `find_all_children()`
- `find_blocks()`
- `find_parent_w_children()`
- `find_children_w_parent()`
- `find_parent_wo_children()`
- `find_children_wo_parent()`

Visualizing returned data with matplotLib

As an old saying goes, *a picture is worth a thousand words*. There's a lot of information that could be extracted from the network, such as interface status, interface counters, router updates, packets dropped, traffic volume, and more. Visualizing this data and putting it into a graph will help you to see the big picture of your network. Python has an excellent library called **matplotlib** (`https://matplotlib.org/`) that is used to generate graphs and customize them.

Matplotlib is capable of creating most kinds of charts, such as line graphs, scatter plots, bar charts, pie charts, stack plots, 3D graphs, and geographic map graphs.

Matplotlib installation

We will start by first installing the library from PYpI using `pip`. Notice some additional packages will be installed along with matplotlib, such as `numpy` and `six`:

```
pip install matplotlib
```

```
bassim@me-inside:~$ pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-2.1.2-cp27-cp27mu-manylinux1_x86_64.whl (15.0MB)
    100% |████████████████████████████████| 15.0MB 79kB/s
Collecting cycler>=0.10 (from matplotlib)
  Downloading cycler-0.10.0-py2.py3-none-any.whl
Collecting numpy>=1.7.1 (from matplotlib)
  Downloading numpy-1.14.1-cp27-cp27mu-manylinux1_x86_64.whl (12.1MB)
    100% |████████████████████████████████| 12.1MB 122kB/s
Collecting backports.functools-lru-cache (from matplotlib)
  Downloading backports.functools_lru_cache-1.5-py2.py3-none-any.whl
Collecting subprocess32 (from matplotlib)
  Downloading subprocess32-3.2.7.tar.gz (54kB)
    100% |████████████████████████████████| 61kB 248kB/s
Collecting pytz (from matplotlib)
  Downloading pytz-2018.3-py2.py3-none-any.whl (509kB)
    100% |████████████████████████████████| 512kB 467kB/s
Collecting six>=1.10 (from matplotlib)
  Using cached six-1.11.0-py2.py3-none-any.whl
Collecting python-dateutil>=2.1 (from matplotlib)
  Using cached python_dateutil-2.6.1-py2.py3-none-any.whl
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
  Downloading pyparsing-2.2.0-py2.py3-none-any.whl (56kB)
    100% |████████████████████████████████| 61kB 234kB/s
Building wheels for collected packages: subprocess32
  Running setup.py bdist_wheel for subprocess32 ... done
  Stored in directory: /home/bassim/.cache/pip/wheels/7d/4c/a4/ce9ceb463dae01f4b95e670abd9afc
```

Now, try to import `matplotlib` and, if no errors are printed, then the module is successfully imported:

```
bassim@me-inside:~$
bassim@me-inside:~$ python
Python 2.7.14 (default, Sep 23 2017, 22:06:14)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import matplotlib
>>>
```

Hands-on with matplotlib

We will start with simple examples to explore matplotlib's functionality. The first thing we do usually is import `matplotlib` into our Python script:

```
import matplotlib.pyplot as plt
```

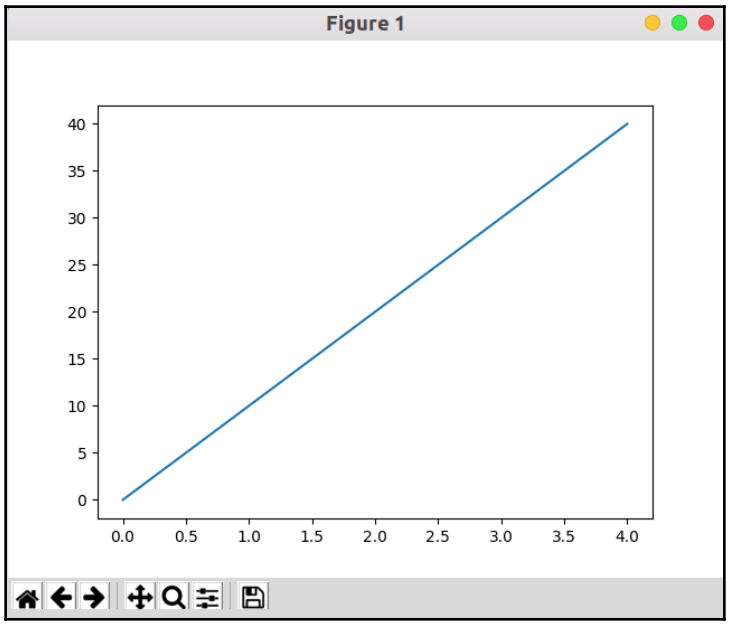
Notice we imported `pyplot` as a short name, `plt`, to be used inside our script. Now, we will use the `plot()` method inside it to plot our data, which consists of two lists. The first list represents the values of the *x*-axis while the second list represents the values of the *y*-axis:

```
plt.plot([0, 1, 2, 3, 4], [0, 10, 20, 30, 40])
```

Now, the values are dropped into the plot.

The last step is to show that plot as a window using the `show()` method:

```
plt.show()
```



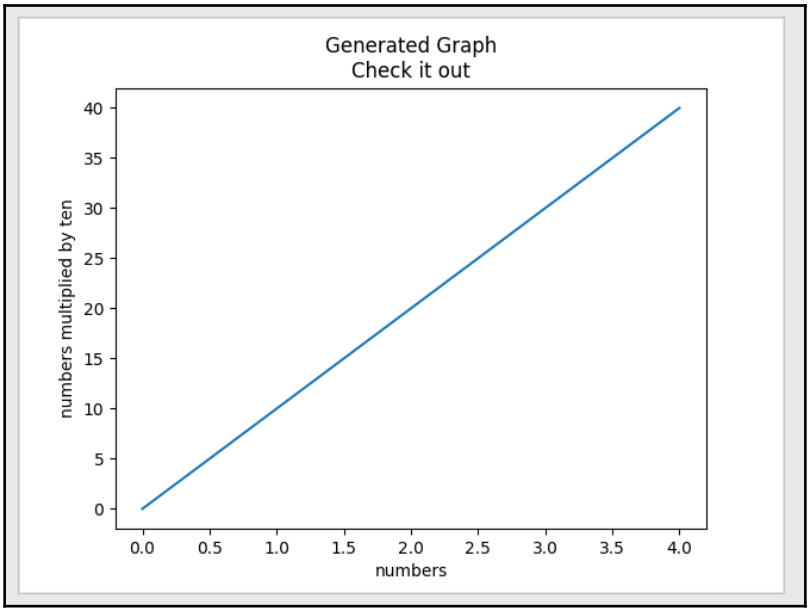
You may need to install `Python-tk` in Ubuntu in order to view the graph. Use `apt install Python-tk`.

The resulted graph will show a line representing the input values of the x and y axes. In the window, you can do the following:

- Move the graph around with the cross icon
- Resize the graph
- Zoom into a specific area with the zoom icon
- Reset to the original view with the home icon
- Save the figure with the save icon

You can customize the generated figure by adding a title to it and labels to both axes. Also, add a legend that explains the meaning of each line in case there are multiple lines on the same graph:

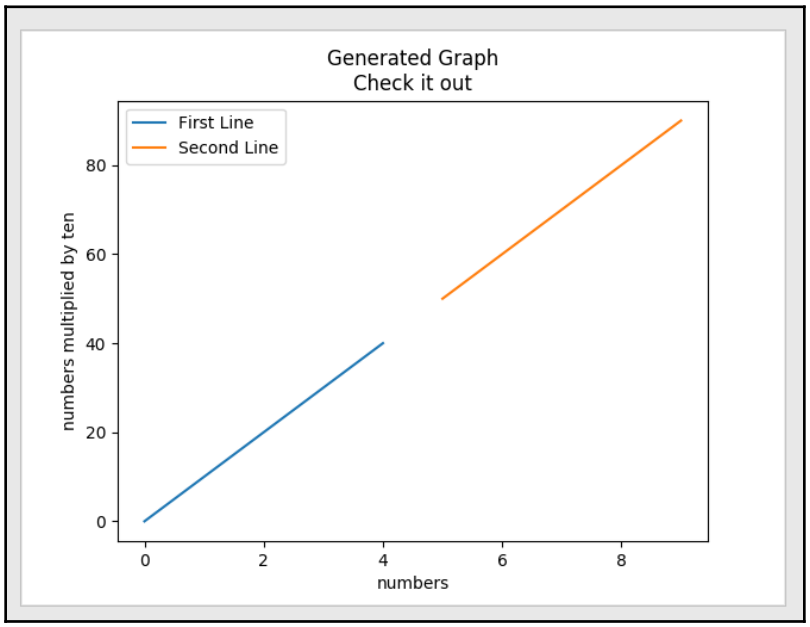
```
import matplotlib.pyplot as plt
plt.plot([0, 1, 2, 3, 4], [0, 10, 20, 30, 40])
plt.xlabel("numbers")
plt.ylabel("numbers multiplied by ten")
plt.title("Generated Graph\nCheck it out")
plt.show()
```



Notice that we usually don't hardcode the plotted values inside the Python script, but we will get them externally from the network, as we will see in the next example.

Also, you can plot multiple datasets on the same figure. You can add another list that represents data to the previous figure and `matplotlib` will draw it. Also, you can add labels to differentiate between the datasets on the graph. The legend for these labels will be printed on the graph using the `legend()` function:

```
import matplotlib.pyplot as plt
plt.plot([0, 1, 2, 3, 4], [0, 10, 20, 30, 40], label="First Line")
plt.plot([5, 6, 7, 8, 9], [50, 60, 70, 80, 90], label="Second Line")
plt.xlabel("numbers")
plt.ylabel("numbers multiplied by ten")
plt.title("Generated Graph\nCheck it out")
plt.legend()
plt.show()
```



Visualizing SNMP using matplotlib

In this use case, we will utilize the `pysnmp` module to send SNMP `GET` requests to our router, retrieve the input and output traffic rates for a specific interface, and visualize the output using the `matplotlib` library. The OIDs used are `.1.3.6.1.4.1.9.2.2.1.1.6` and `.1.3.6.1.4.1.9.2.2.1.1.8`, which represent the input and output rates respectively:

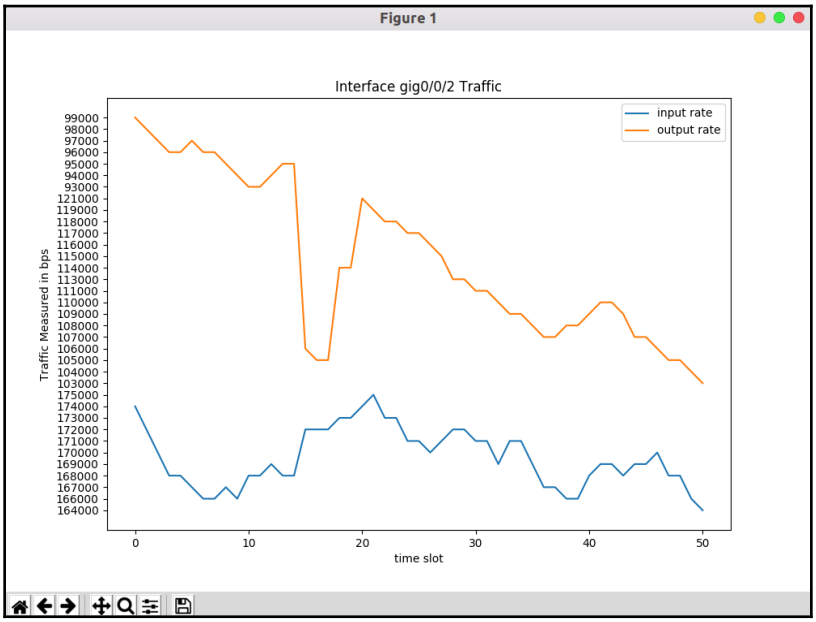
```
from pysnmp.entity.rfc3413.oneliner import cmdgen
import time
import matplotlib.pyplot as plt

cmdGen = cmdgen.CommandGenerator()

snmp_community = cmdgen.CommunityData('public')
snmp_ip =  cmdgen.UdpTransportTarget(('10.10.88.110', 161))
snmp_oids = [".1.3.6.1.4.1.9.2.2.1.1.6.3",".1.3.6.1.4.1.9.2.2.1.1.8.3"]

slots = 0
input_rates = []
output_rates = []
while slots <= 50:
    errorIndication, errorStatus, errorIndex, varBinds =
cmdGen.getCmd(snmp_community, snmp_ip, *snmp_oids)

    input_rate = str(varBinds[0]).split("=")[1].strip()
    output_rate = str(varBinds[1]).split("=")[1].strip()

    input_rates.append(input_rate)
    output_rates.append(output_rate)

    time.sleep(6)
    slots = slots + 1
    print slots

time_range = range(0, slots)

print input_rates
print output_rates
# plt.figure()
plt.plot(time_range, input_rates, label="input rate")
plt.plot(time_range, output_rates, label="output rate")
plt.xlabel("time slot")
plt.ylabel("Traffic Measured in bps")
plt.title("Interface gig0/0/2 Traffic")
```

```
plt.legend()
plt.show()
```

In this example, we can see the following:

- We imported `cmdgen` from the `pysnmp` module, which was used to create SNMP `GET` commands for the router. We also imported the `matplotlib` module.
- Then, we used `cmdgen` to define the transport channel properties between Python and the router and provide the SNMP community.
- `pysnmp` will start to send the SNMP GET requests with the provided OIDs and return the output and errors (if any) to `errorIndication`, `errorStatus`, `errorIndex`, and `varBinds`. We are interested in `varBinds` as it holds the actual values for the input and output traffic rate.
- Note that `varBinds` will be in the form of `<oid> = <value>`, so we extracted only the value and added it to the corresponding list we created before.
- This operation will be repeated 100 times at 6-second intervals to collect useful data.
- Finally, we provided the collected data to the `plt` imported from `matplotlib` and customized the graph by providing the `xlabel`, `ylabel`, title, and `legends`:

Script output:



Summary

In this chapter, we learned how to use different tools and techniques inside Python to extract useful data from returned output and act upon it. Also, we used a special library called `CiscoConfParse` to audit the configuration and learned how to visualize data to generate appealing graphs and reports.

In the next chapter, we will learn how to write a template and use it to generate configurations with a Jinja2 templating language.

6
Configuration Generator with Python and Jinja2

This chapter introduces you to the YAML format for representing data and generating a configuration from the golden templates created by the Jinja2 language. We will use these two concepts in both Ansible and Python to create a data model store for our configuration.

We will cover the following topics in this chapter:

- What is YAML?
- Building golden configuration templates with Jinja2

What is YAML?

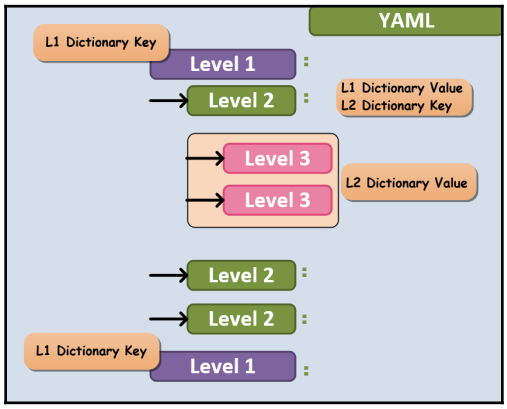
YAML Ain't Markup Language (**YAML**) is often called a data serialization language. It was intended to be human-readable and organize data into a structured format. Programming languages can understand the content of YAML files (which usually have a `.yml` or `.yaml` extension) and map them to built-in data types. For example, when you consume a `.yaml` file in your Python script, it will automatically convert the content into either a dictionary `{}` or list `[]`, so you can work and iterate over it.

YAML rules help to construct a readable file so it's important to understand them in order to write a valid and well formatted YAML file.

YAML file formatting

There're a few rules to follow while developing YAML files. YAML uses indentation (like Python), which builds the relationship of items with one another:

1. So, the first rule when writing a YAML file is to make your indentation consistent, using either whitespace or tabs, and don't mix them.
2. The second rule is to use a colon `:` when creating a dictionary with a key and value (sometimes they're called associative arrays in `yaml`). The item to the left of the colon is the key, while the item to the right of the colon is the value.
3. The third rule is to use dashes `"-"` when grouping items inside a list. You can mix dictionaries and lists inside the YAML file in order to effectively describe your data. The left-hand side serves as a dictionary key, while the right-hand side serves as a dictionary value. You can create any number of levels to have structured data:



Let's take an example and apply these rules to it:

```
---
my_datacenter:
  GW:
    eve_port: 32773
    device_template: vIOSL3_Template
    hostname: R1
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.110
    mgmt_subnet: 255.255.255.0
    enabled_ports:
      - gig0/0
      - gig0/1
      - gig0/2

  switch1:
    eve_port: 32769
    device_template: vIOSL2_Template
    hostname: SW1
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.111
    mgmt_subnet: 255.255.255.0

  switch2:
    eve_port: 32770
    device_template: vIOSL2_Template
    hostname: SW2
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.112
    mgmt_subnet: 255.255.255.0

```

There are a number of things to look at it. Firstly, the file has one top level, `my_datacenter`, which serves as a top-level key and its values consists of all the indented lines after it, which are `GW`, `switch1`, and `switch2`. Those items also serve as keys and have values inside them, which are `eve_port`, `device_template`, `hostname`, `mgmt_int`, `mgmt_ip`, and `mgmt_subnet` and which serve as Level 3 keys and Level 2 values at the same time.

The other thing to notice is `enabled_ports`, which is a key but has a value that serves as a lists. We know this because the next level of indentation is a dash.

Notice that all interfaces are sibling elements because they have the same level of indentation.

Finally, it's not required to have a single or double quotation around strings. Python will do that automatically when we load the file into it and it will also determine the data type and location of each item based on indentation.

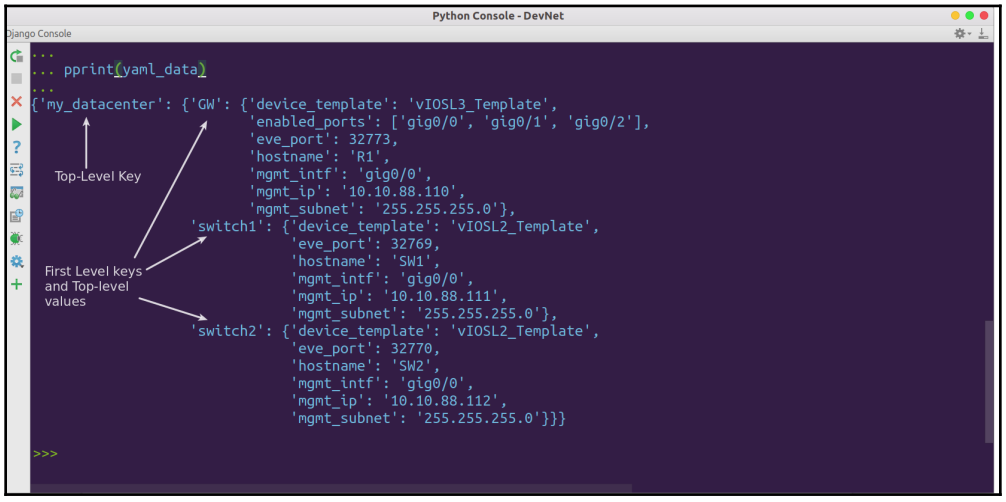
Now, let's develop a Python script that reads this YAML file and converts it into dictionaries and lists using the `yaml` module:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import yaml
from pprint import pprint


with open(r'/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject
  /Chapter6_Configuration_generator_with_python_and_jinja2/yaml_example.yml', 'r') as yaml_file:
    yaml_data = yaml.load(yaml_file)  # This is to read the file content

pprint(yaml_data)

```

In this example, we can see the following:

- We imported the `yaml` module inside our Python script in order to handle the YAML files. Also, we imported the `pprint` function to show the hierarchy of nested dictionaries and lists.
- Then, we opened the `yaml_example.yml` file using the `with` clause and the `open()` function as a `yaml_file`.
- Finally, we use the `load()` function to load the file into the `yaml_data` variable. At this stage, the Python interpreter will analyze the `yaml` file's content and build the relationships between items, then convert them to the standard data type. The output can be shown at the console using the `pprint()` function.

Script output



It's now fairly easy to access any information using standard Python methods. For example, you can access the `switch1` config by using `my_datacenter` followed by the `switch1` keys, as in the following code snippet:

```
pprint(yaml_data['my_datacenter']['switch1'])

{'device_template': 'vIOSL2_Template',
 'eve_port': 32769,
 'hostname': 'SW1',
 'mgmt_intf': 'gig0/0',
 'mgmt_ip': '10.10.88.111',
 'mgmt_subnet': '255.255.255.0'}
```

Also, you can iterate over the keys with a simple `for` loop and print the values of any level:

```
for device in yaml_data['my_datacenter']:
    print device

GW
switch2
switch1
```

As a best practice, it's recommended you keep the key names consistent and change only the values while you describe your data. For example, the `hostname`, `mgmt_intf`, and `mgmt_ip` items exist on all devices with the same name, while they have different values in the `.yaml` file.

Text editor tips

Correct indentation is very important for YAML data. It's recommended to use an advanced text editor such as, Sublime Text or Notepad++, as they have options that convert the tabs to a specific number of whitespaces. At the same time, you can choose the specific **tab indentation size** to be 2 or 4. So, your editor will convert the tab to a static number of whitespaces whenever you click on the *Tab* button. Finally, you can choose to display vertical lines at each indentation to ensure that lines are indented the same amount.

Please note that Microsoft Windows Notepad doesn't have that option and this may result in a formatting error in your YAML file.

The following is an example of an advanced editor called Sublime Text that can be configured with the aforementioned options:

The screenshot shows the vertical line guides that ensure that the sibling items are at the same indentation level and number of spaces when you click on **Tab**.

Building a golden configuration with Jinja2

Most network engineers have a text file that serves as a template for a specific device configuration. This file contains sections of network configuration with many values. When the network engineer wants to provision a new device or change its configuration, they will basically replace specific values from this file with another one to generate a new configuration.

Using Python and Ansible, later in this book we will automate this process efficiently using the Jinja2 template language (`http://jinja.pocoo.org`). The core concept of and driver for developing Jinja2 is to have a unified syntax across all template files for specific network/system configurations and to separate the data from the actual configuration. This allows us to use the same template multiple times but with a different set of data. Also, as shown on the Jinja2 web page, it has some unique features that make it stand out from the other template languages.

The following are some of the features mentioned on the official website:

- Powerful automatic HTML escaping system for cross-site scripting prevention.
- High performance with just-in-time compilation to Python bytecode. Jinja2 will translate your template sources on first load into Python bytecode for the best runtime performance.
- Optional ahead-of-time compilation.
- Easy to debug with a debug system that integrates template compile and runtime errors into the standard Python traceback system.
- Configurable syntax: For instance, you can reconfigure Jinja2 to better fit output formats, such as LaTeX or JavaScript.
- Template designer helpers: Jinja2 ships with a wide range of useful little helpers that help solve common tasks in templates, such as breaking up sequences of items into multiple columns.

Another important Jinja feature is *template inheritance*, with which we can create a *base/parent template* that defines a basic structure for our system or the Day 0 initial configuration for all devices. This initial configuration will be the base configuration and contains the common pieces such as usernames, management subnet, default routes, and SNMP communities. The other *child templates* extend the base template and inherit it.

The terms Jinja and Jinja2 are used interchangeably throughout this chapter.

Let's take a few examples of building templates before we deep dive into more features provided by the Jinja2 language:

1. First, we need to make sure that Jinja2 is installed in your system by using the following command:

   ```
   pip install jinja2
   ```

   The package will be downloaded from PyPi and then will be installed on the site packages.

2. Now, open your favorite text editor and write the following template, which represents a simple Day 0 (initial) configuration for a Layer 2 switch that configures the device hostname, some `aaa` parameters, default VLANs that should exist on each switch, and the management of IP addresses:

```
hostname {{ hostname }}

aaa new-model
aaa session-id unique
aaa authentication login default local
aaa authorization exec default local none
vtp mode transparent
vlan 10,20,30,40,50,60,70,80,90,100,200

int {{ mgmt_intf }}
no switchport
no shut
ip address {{ mgmt_ip }} {{ mgmt_subnet }}
```

Some text editors (such as Sublime Text and Notepad++) provide support for Jinja2 and can do syntax highlighting and auto-completion for you, either by natively supporting it or through extension.

Notice that in the previous template, the variables were written in double curly braces `{{ }}`. So, when the Python script loads the template, it will replace those variables with the desired values:

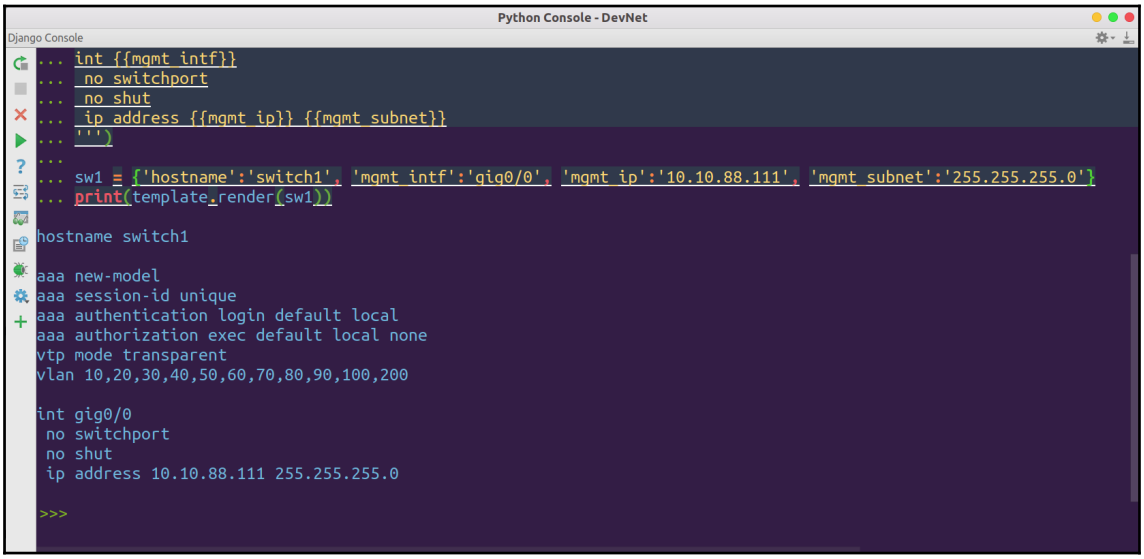
```
#!/usr/bin/python

from jinja2 import Template
template = Template('''
hostname {{hostname}}

aaa new-model
aaa session-id unique
aaa authentication login default local
aaa authorization exec default local none
vtp mode transparent
vlan 10,20,30,40,50,60,70,80,90,100,200

int {{mgmt_intf}}
 no switchport
 no shut
 ip address {{mgmt_ip}} {{mgmt_subnet}}
''')
```

```
sw1 = {'hostname': 'switch1', 'mgmt_intf': 'gig0/0', 'mgmt_ip':
'10.10.88.111', 'mgmt_subnet': '255.255.255.0'}
print(template.render(sw1))
```

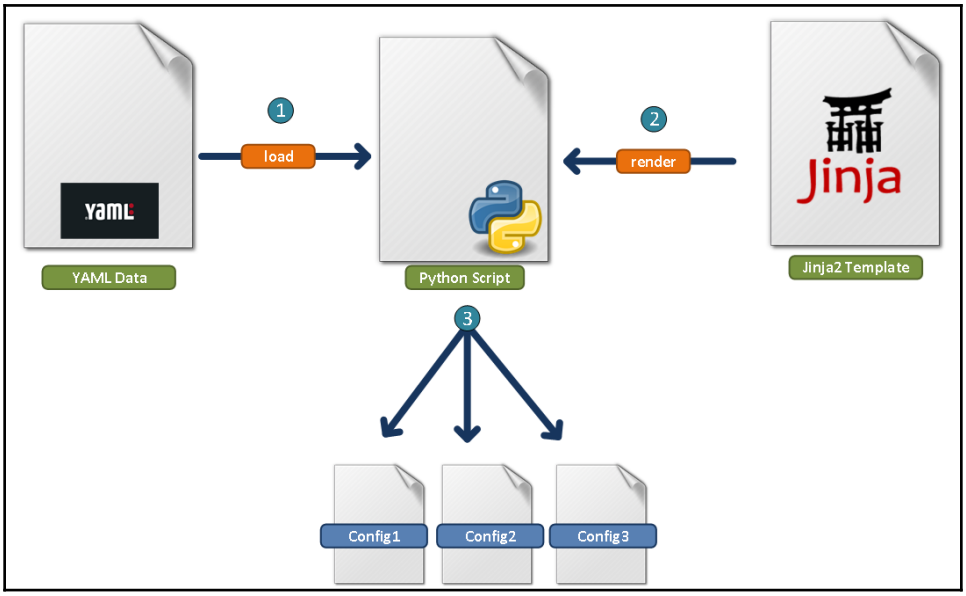
In this example, we can see the following:

- The first thing is we imported the `Template` class from the `jinja2` module. This class will validate and parse the Jinja2 file.
- Then, we defined a variable, `sw1`, as a dictionary with keys that have names equal to variables inside the template. The dictionary values will be the data that renders the template.
- Finally, we used the `render()` method inside the template which takes `sw1` as an input to connect the Jinja2 template with the rendered values and prints the configuration.

Script output



Now, let's enhance our script and use YAML to render the template instead of hard-coding the values inside dictionaries. The concept is simple: we will model the `day0` configuration for our lab inside the YAML file, then load this file into our Python script using `yaml.load()` and use the output to feed the Jinja2 template, which will result in generating the `day0` configuration files for each device:



First, we will extend the YAML file that we developed last time and add other devices to it while keeping the hierarchy for each node the same:

```
---
dc1:
  GW:
    eve_port: 32773
    device_template: vIOSL3_Template
    hostname: R1
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.110
    mgmt_subnet: 255.255.255.0


  switch1:
    eve_port: 32769
    device_template: vIOSL2_Template
    hostname: SW1
    mgmt_intf: gig0/0
```

```
    mgmt_ip: 10.10.88.111
    mgmt_subnet: 255.255.255.0

  switch2:
    eve_port: 32770
    device_template: vIOSL2_Template
    hostname: SW2
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.112
    mgmt_subnet: 255.255.255.0

  switch3:
    eve_port: 32769
    device_template: vIOSL2_Template
    hostname: SW3
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.113
    mgmt_subnet: 255.255.255.0

  switch4:
    eve_port: 32770
    device_template: vIOSL2_Template
    hostname: SW4
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.114
    mgmt_subnet: 255.255.255.0
```

Following is the Python script:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import yaml
from jinja2 import Template


with
open('/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject/Chap
ter6_Configuration_generator_with_python_and_jinja2/network_dc.yml', 'r')
as yaml_file:
    yaml_data = yaml.load(yaml_file)

router_day0_template = Template("""
hostname {{hostname}}
int {{mgmt_intf}}
  no shutdown
  ip add {{mgmt_ip}} {{mgmt_subnet}}
```

```
lldp run

ip domain-name EnterpriseAutomation.net
ip ssh version 2
ip scp server enable
crypto key generate rsa general-keys modulus 1024

snmp-server community public RW
snmp-server trap link ietf
snmp-server enable traps snmp linkdown linkup
snmp-server enable traps syslog
snmp-server manager

logging history debugging
logging snmp-trap emergencies
logging snmp-trap alerts
logging snmp-trap critical
logging snmp-trap errors
logging snmp-trap warnings
logging snmp-trap notifications
logging snmp-trap informational
logging snmp-trap debugging

""")


switch_day0_template = Template("""
hostname {{hostname}}

aaa new-model
aaa session-id unique
aaa authentication login default local
aaa authorization exec default local none
vtp mode transparent
vlan 10,20,30,40,50,60,70,80,90,100,200

int {{mgmt_intf}}
 no switchport
 no shut
 ip address {{mgmt_ip}} {{mgmt_subnet}}

snmp-server community public RW
snmp-server trap link ietf
snmp-server enable traps snmp linkdown linkup
snmp-server enable traps syslog
snmp-server manager

logging history debugging
```

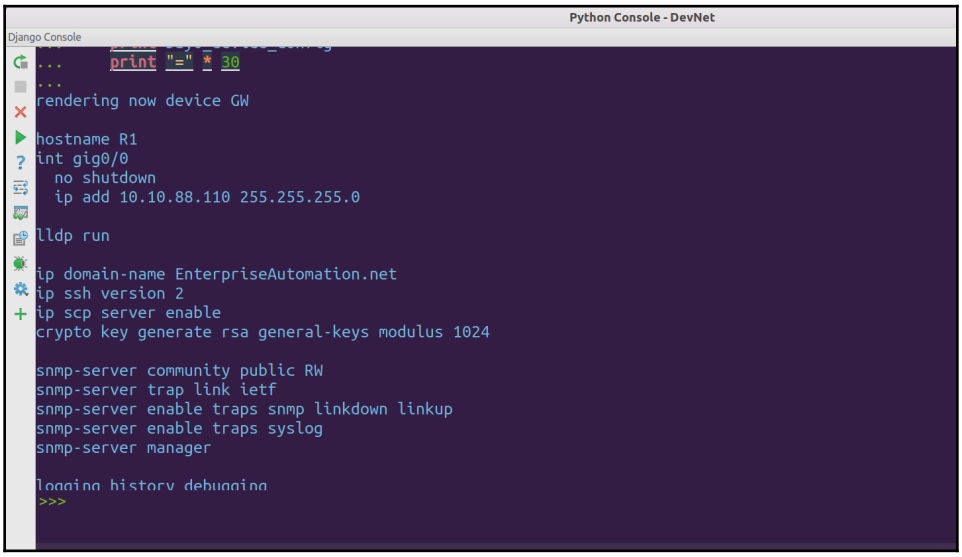
```
logging snmp-trap emergencies
logging snmp-trap alerts
logging snmp-trap critical
logging snmp-trap errors
logging snmp-trap warnings
logging snmp-trap notifications
logging snmp-trap informational
logging snmp-trap debugging

""")

for device,config in yaml_data['dc1'].iteritems():
    if config['device_template'] == "vIOSL2_Template":
        device_template = switch_day0_template
    elif config['device_template'] == "vIOSL3_Template":
        device_template = router_day0_template

    print("rendering now device {0}" .format(device))
    Day0_device_config = device_template.render(config)

    print Day0_device_config
    print "=" * 30
```

In this example, we can see the following:

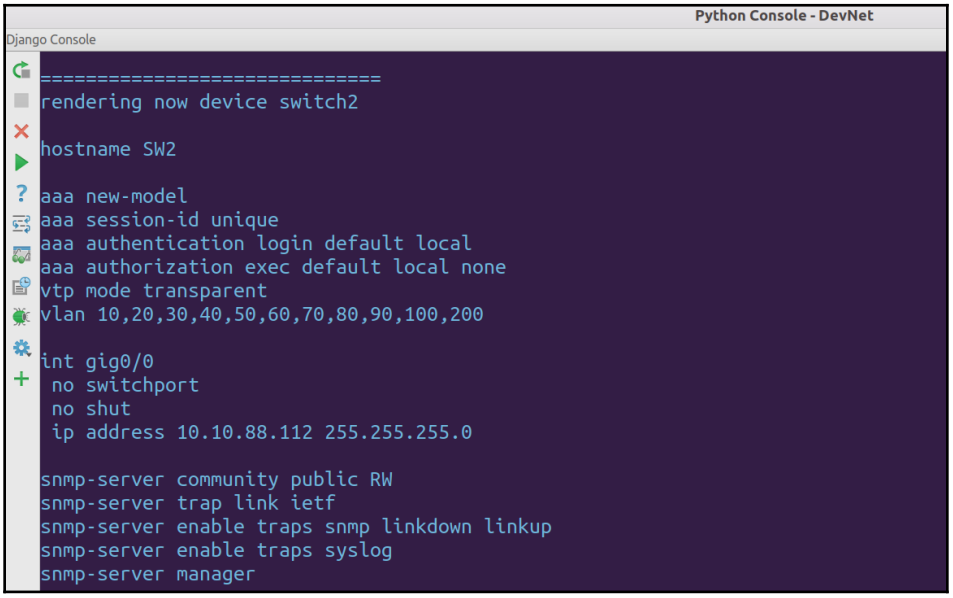
- We imported the `yaml` and `Jinja2` modules as usual
- Then, we instructed the script to load the `yaml` file into the `yaml_data` variable, which will convert it into a series of dictionaries and lists
- Two templates for router and switch configuration are defined as `router_day0_template` and `switch_day0_template` respectively
- The `for` loop will iterate over devices of `dc1` and check the `device_template`, then will render configuration for each device

Script output

Following is the router configuration (output omitted):



Following is the switch 1 configuration (output omitted):



Reading templates from the filesystem

A common approach for Python developers is to move the static, hard-coded values and templates outside the Python script and keep only the logic inside the script. This approach keeps your program clean and scalable, while allowing other team members who don't have much knowledge of Python to get the desired output by changing the input, and Jinja2 is no exception to this approach. You can use the `FileSystemLoader()` class inside the Jinja2 module to load the template from the operating system directories. We will modify our code and move both the `router_day0_template` and `switch_day0_template` contents from the script to text files, then load them into our script.

Python code

```
import yaml
from jinja2 import FileSystemLoader, Environment


with
open('/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject/Chap
ter6_Configuration_generator_with_python_and_jinja2/network_dc.yml', 'r')
as yaml_file:
    yaml_data = yaml.load(yaml_file)


template_dir =
"/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject/Chapter6_
Configuration_generator_with_python_and_jinja2"

template_env = Environment(loader=FileSystemLoader(template_dir),
                           trim_blocks=True,
                           lstrip_blocks= True
                           )


for device,config in yaml_data['dc1'].iteritems():
    if config['device_template'] == "vIOSL2_Template":
        device_template =
template_env.get_template("switch_day1_template.j2")
    elif config['device_template'] == "vIOSL3_Template":
        device_template =
template_env.get_template("router_day1_template.j2")

    print("rendering now device {0}" .format(device))
    Day0_device_config = device_template.render(config)
```

```
print Day0_device_config
print "=" * 30
```

In this example, instead of loading the `Template()` class from the Jinja2 module as we did before, we will import `Environment()` and `FileSystemLoader()`, which are used to read the Jinja2 file from the specific operating system directory by providing them with `template_dir` where our templates are stored. Then, we will use the created `template_env` object, along with the `get_template()` method, to get the template name and render it with the configuration.

Make sure your template file has a `.j2` extension at the end. This will make PyCharm recognize the text inside the file as a Jinja2 template and hence provide syntax highlighting and better code completion.

Using Jinja2 loops and conditions

Loops and conditions in Jinja2 are used to enhance our template and add more functionality to it. We will start by understanding how to add the `for` loop inside the template in order to iterate over passed values from YAML. For example, we may need to add a switch configuration under each interface, such as using the switchport mode and configure the VLAN ID which will be configured under the access port, or configure the allowed VLANs range in the case of the trunk ports.

On the other hand, we may need to enable some interfaces in the router and add custom configurations to it, such as MTU, speed, and duplex. So, we will use the `for` loop.

Notice that part of our script logic will now be moved from Python to the Jinja2 template. The Python script will just read the template, either externally from the operating system or through the `Template()` class inside the script, then render the template with the parsed values from the YAML file.

The basic structure of `for` loops inside Jinja2 is as follows:

```
{% for key, value in var1.iteritems() %}
configuration snippets
{% endfor %}
```

Notice the use of `{% %}` to define logic inside the Jinja2 file.

Also, `iteritems()` has the same function as iterating over the Python dictionary, which is iterating over the key and value pairs. The loop will return both the key and value for each element inside the `var1` dictionary.

Also, we can have an `if` condition that validates a specific condition and, if it's true, then the configuration snippets will be added to the rendered file. The basic `if` structure will be as shown in the following snippet:

```
{% if enabled_ports %}
configuration snippet goes here and added to template if the condition is
true
{% endif %}
```

Now, we will modify our `.yaml` file which describes the data center devices, and add the interface configuration and enabled ports for each device:

```
---
dc1:
  GW:
    eve_port: 32773
    device_template: vIOSL3_Template
    hostname: R1
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.110
    mgmt_subnet: 255.255.255.0
    enabled_ports:
      - gig0/0
      - gig0/1
      - gig0/2

  switch1:
    eve_port: 32769
    device_template: vIOSL2_Template
    hostname: SW1
    mgmt_intf: gig0/0
    mgmt_ip: 10.10.88.111
    mgmt_subnet: 255.255.255.0
    interfaces:
      gig0/1:
        vlan: [1,10,20,200]
        description: TO_DSW2_1
        mode: trunk
      gig0/2:
        vlan: [1,10,20,200]
        description: TO_DSW2_2
        mode: trunk
      gig0/3:
```

```
      vlan: [1,10,20,200]
      description: TO_ASW3
      mode: trunk
    gig1/0:
      vlan: [1,10,20,200]
      description: TO_ASW4
      mode: trunk
  enabled_ports:
    - gig0/0
    - gig1/1

switch2:
  eve_port: 32770
  device_template: vIOSL2_Template
  hostname: SW2
  mgmt_intf: gig0/0
  mgmt_ip: 10.10.88.112
  mgmt_subnet: 255.255.255.0
  interfaces:
    gig0/1:
      vlan: [1,10,20,200]
      description: TO_DSW1_1
      mode: trunk
    gig0/2:
      vlan: [1,10,20,200]
      description: TO_DSW1_2
      mode: trunk
    gig0/3:
      vlan: [1,10,20,200]
      description: TO_ASW3
      mode: trunk
    gig1/0:
      vlan: [1,10,20,200]
      description: TO_ASW4
      mode: trunk
  enabled_ports:
    - gig0/0
    - gig1/1

switch3:
  eve_port: 32769
  device_template: vIOSL2_Template
  hostname: SW3
  mgmt_intf: gig0/0
  mgmt_ip: 10.10.88.113
  mgmt_subnet: 255.255.255.0
  interfaces:
    gig0/1:
```

```
      vlan: [1,10,20,200]
      description: TO_DSW1
      mode: trunk
    gig0/2:
      vlan: [1,10,20,200]
      description: TO_DSW2
      mode: trunk
    gig1/0:
      vlan: 10
      description: TO_Client1
      mode: access
    gig1/1:
      vlan: 20
      description: TO_Client2
      mode: access
  enabled_ports:
    - gig0/0

switch4:
  eve_port: 32770
  device_template: vIOSL2_Template
  hostname: SW4
  mgmt_intf: gig0/0
  mgmt_ip: 10.10.88.114
  mgmt_subnet: 255.255.255.0
  interfaces:
    gig0/1:
      vlan: [1,10,20,200]
      description: TO_DSW2
      mode: trunk
    gig0/2:
      vlan: [1,10,20,200]
      description: TO_DSW1
      mode: trunk
    gig1/0:
      vlan: 10
      description: TO_Client1
      mode: access
    gig1/1:
      vlan: 20
      description: TO_Client2
      mode: access
  enabled_ports:
    - gig0/0
```

Notice, that we categorized the switch ports to either trunk port or access port, and also added the vlans for each one.

According to the `yaml` file, the incoming packets to the interface with switchport access mode will be tagged with the VLAN. In case of the switchport mode trunk, the incoming packets be allowed if it has a vlan ID belong to the configured list.

Now, we will create two additional templates for devices Day 1 (operational) configuration. The first template will be `router_day1_template` and the second will be `switch_day1_template`, and both of them will inherit from the corresponding day0 template that we developed before:

router_day1_template:

```
{% include 'router_day0_template.j2' %}

{% if enabled_ports %}
    {% for port in enabled_ports %}
interface {{ port }}
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
    {% endfor %}

{% endif %}
```

switch_day1_template:

```
{% include 'switch_day0_template.j2' %}

{% if enabled_ports %}
    {% for port in enabled_ports %}
interface {{ port }}
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto

    {% endfor %}
{% endif %}
```

```
{% if interfaces %}
    {% for intf,intf_config in interfaces.items() %}
interface {{ intf }}
 description "{{intf_config['description']}}"
 no shutdown
 duplex full
        {% if intf_config['mode'] %}
            {% if intf_config['mode'] == "access" %}
 switchport mode {{intf_config['mode']}}
 switchport access vlan {{intf_config['vlan']}}

            {% elif intf_config['mode'] == "trunk" %}
 switchport {{intf_config['mode']}} encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan {{intf_config['vlan']|join(',')}}

            {% endif %}
        {% endif %}
    {% endfor %}
{% endif %}
```

Notice the use of the `{% include <template_name.j2> %}` tag, which refers to the day0 template of the device.

This template will be rendered first and filled with passed values from YAML, then the next parts will be filled.

The Jinja2 language inherits many writing styles and features from the Python language. Although it's not mandatory to follow the indentation rule when developing the template and inserting the tags, the author prefers to have it in a readable Jinja2 template.

Script output:

```
rendering now device GW
hostname R1
int gig0/0
  no shutdown
  ip add 10.10.88.110 255.255.255.0
lldp run
ip domain-name EnterpriseAutomation.net
ip ssh version 2
ip scp server enable
crypto key generate rsa general-keys modulus 1024
snmp-server community public RW
```

```
snmp-server trap link ietf
snmp-server enable traps snmp linkdown linkup
snmp-server enable traps syslog
snmp-server manager
logging history debugging
logging snmp-trap emergencies
logging snmp-trap alerts
logging snmp-trap critical
logging snmp-trap errors
logging snmp-trap warnings
logging snmp-trap notifications
logging snmp-trap informational
logging snmp-trap debugging
interface gig0/0
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
interface gig0/1
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
interface gig0/2
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
==============================
rendering now device switch1
hostname SW1
aaa new-model
aaa session-id unique
aaa authentication login default local
aaa authorization exec default local none
vtp mode transparent
vlan 10,20,30,40,50,60,70,80,90,100,200
int gig0/0
 no switchport
 no shut
 ip address 10.10.88.111 255.255.255.0
snmp-server community public RW
snmp-server trap link ietf
snmp-server enable traps snmp linkdown linkup
snmp-server enable traps syslog
```

```
snmp-server manager
logging history debugging
logging snmp-trap emergencies
logging snmp-trap alerts
logging snmp-trap critical
logging snmp-trap errors
logging snmp-trap warnings
logging snmp-trap notifications
logging snmp-trap informational
logging snmp-trap debugging
interface gig0/0
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
interface gig1/1
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
interface gig0/2
 description "TO_DSW2_2"
 no shutdown
 duplex full
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 1,10,20,200
interface gig0/3
 description "TO_ASW3"
 no shutdown
 duplex full
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 1,10,20,200
interface gig0/1
 description "TO_DSW2_1"
 no shutdown
 duplex full
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 1,10,20,200
interface gig1/0
 description "TO_ASW4"
 no shutdown
 duplex full
 switchport trunk encapsulation dot1q
```

```
 switchport mode trunk
 switchport trunk allowed vlan 1,10,20,200
==============================

<switch2 output omitted>

==============================
rendering now device switch3
hostname SW3
aaa new-model
aaa session-id unique
aaa authentication login default local
aaa authorization exec default local none
vtp mode transparent
vlan 10,20,30,40,50,60,70,80,90,100,200
int gig0/0
 no switchport
 no shut
 ip address 10.10.88.113 255.255.255.0
snmp-server community public RW
snmp-server trap link ietf
snmp-server enable traps snmp linkdown linkup
snmp-server enable traps syslog
snmp-server manager
logging history debugging
logging snmp-trap emergencies
logging snmp-trap alerts
logging snmp-trap critical
logging snmp-trap errors
logging snmp-trap warnings
logging snmp-trap notifications
logging snmp-trap informational
logging snmp-trap debugging
interface gig0/0
    no switchport
    no shutdown
    mtu 1520
    duplex auto
    speed auto
interface gig0/2
 description "TO_DSW2"
 no shutdown
 duplex full
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 1,10,20,200
interface gig1/1
 description "TO_Client2"
```

```
 no shutdown
 duplex full
 switchport mode access
 switchport access vlan 20
interface gig1/0
 description "TO_Client1"
 no shutdown
 duplex full
 switchport mode access
 switchport access vlan 10
interface gig0/1
 description "TO_DSW1"
 no shutdown
 duplex full
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 1,10,20,200
==============================
<switch4 output omitted>
```

Summary

In this chapter, we learned about YAML and its formatting and how to work with text editors. We also learned about Jinja2 and its configuration. Then, we explored the ways in which we can use loops and conditions in Jinja2.

In the next chapter, we will learn how to instantiate and execute Python code in parallel using multiprocessing.

7
Parallel Execution of Python Script

Python has become the *de facto* standard for network automation. Many network engineers already use it on a daily basis to automate networking tasks, from configuration, to operation, to troubleshooting network problems. In this chapter, we will visit one of the advanced topics in Python: scratching the surface of Python's multiprocessing nature and learning how to use it to accelerate script execution time.

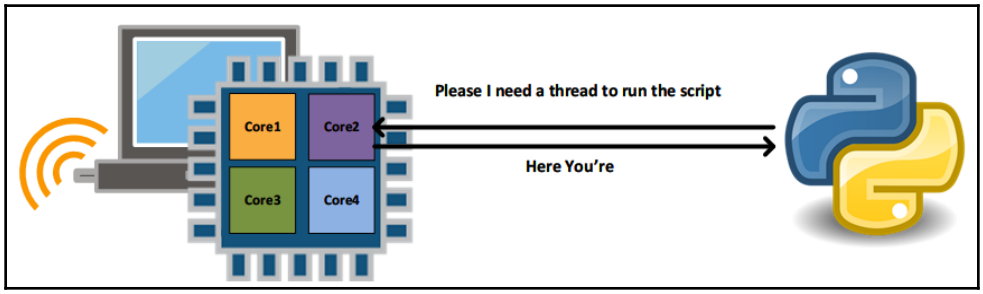
We will cover the following topics in this chapter:

- How Python code is executed in an OS
- The Python multiprocessing library

How a computer executes your Python script

This is how your computer's operating system executes Python script:

1. When you type `python <your_awesome_automation_script>.py` in the shell, Python (which runs as a process) instructs your computer processor to schedule a thread (which is the smallest unit of processing):



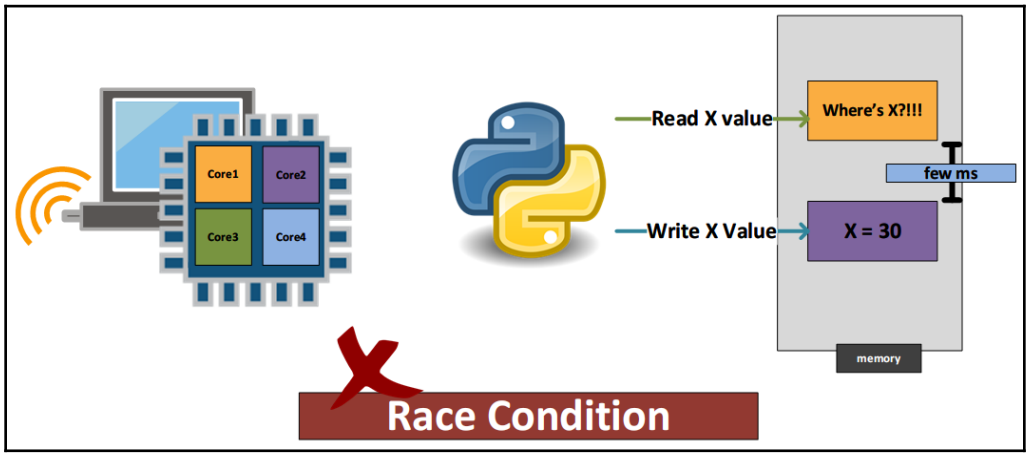
2. The allocated thread will start to execute your script, line by line. A thread can do anything, including interacting with I/O devices, connecting to routers, printing output, performing mathematical equations, and more.
3. Once the script hits the **End of File** (**EOF**), the thread will be terminated and returned to the free pool, to be used by other processes. Then, the script is terminated.

In Linux, you can use `#strace –p <pid>` to trace a specific thread execution.

The more threads that you assign to your script (and that are permitted by your processor or OS), the faster your script will run. Actually, threads are sometimes called **workers** or **slaves**.

I have a feeling that you have this little idea in your head: Why wouldn't we assign a lot of threads, from all cores, to Python script, in order to get the job done quickly?

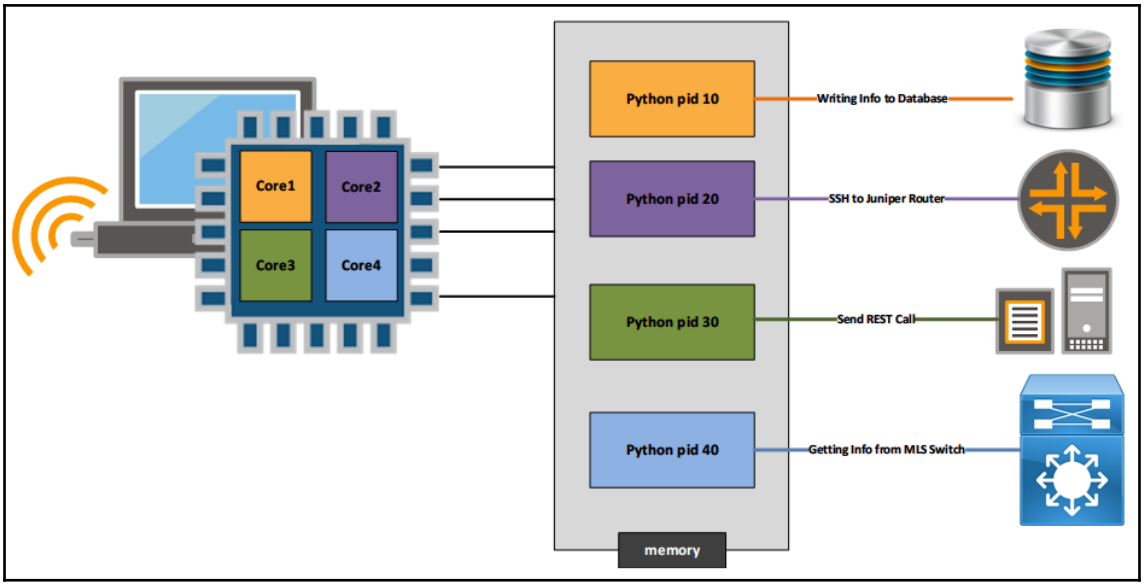
The problem with assigning a lot of threads to one process without special handling is the **race condition**. The operating systems will allocate memory to your process (in this case, it's the Python process), to be used at runtime and accessed by all threads—*all of them at the same time*. Now, imagine that one of those threads reads some data before it's actually written by another thread! You don't know the order in which the threads will attempt to access the shared data; this is the race condition:



One available solution is to make the thread acquire a lock. In fact, Python, by default, is optimized to run as a single-threaded process, and has something called **Global Interpreter Lock** (**GIL**). GIL does not allow multiple threads to execute Python code at the same time, in order to prevent conflicts between threads.

But, rather than having multiple threads, why don't we have multiple processes?

The beauty of multiple processes, as compared to multiple threads, is that you don't have to be afraid of data corruption due to shared data. Each spawned process will have its own allocated memory, which won't be accessed by other Python processes. This allows us to execute parallel tasks at the same time:



Also, from Python's point of view, each process has its own GIL. So, there's no resource conflict or race condition here.

Python multiprocessing library

The `multiprocessing` module is Python's standard library that is shipped with Python binaries, and it is available from Python 2.6. There's also the `threading` module, which allows you to spawn multiple threads, but they all share the same memory space. Multiprocessing comes with more advantages than threading. One of them is isolated memory space for each process, and it can take advantage of multiple CPUs and cores.

Getting started with multiprocessing

First, you need to import the module for your Python script:

```
import multiprocessing as mp
```

Then, wrap your code with a Python function; this will allow the process to target this function and mark it as a parallel execution.

Let's suppose that we have code that connects to the router and executes commands on it using the `netmiko` library, and we want to connect to all of the devices in parallel. This is a sample serial code that will connect to each device and execute the passed command, and then continue with the second device, and so on:

```
from netmiko import ConnectHandler
from devices import R1, SW1, SW2, SW3, SW4

nodes = [R1, SW1, SW2, SW3, SW4]

for device in nodes:
    net_connect = ConnectHandler(**device)
    output = net_connect.send_command("show run")
    print output
```

The Python file `devices.py` is created on the same directory as our script, and it contains the login details and credentials for each device in a `dictionary` format:

```
R1 = {"device_type": "cisco_ios_ssh",
      "ip": "10.10.88.110",
      "port": 22,
      "username": "admin",
      "password": "access123",
      }

SW1 = {"device_type": "cisco_ios_ssh",
       "ip": "10.10.88.111",
       "port": 22,
       "username": "admin",
       "password": "access123",
       }

SW2 = {"device_type": "cisco_ios_ssh",
       "ip": "10.10.88.112",
       "port": 22,
       "username": "admin",
       "password": "access123",
```

```
        }

SW3 = {"device_type": "cisco_ios_ssh",
        "ip": "10.10.88.113",
        "port": 22,
        "username": "admin",
        "password": "access123",
        }

SW4 = {"device_type": "cisco_ios_ssh",
        "ip": "10.10.88.114",
        "port": 22,
        "username": "admin",
        "password": "access123",
        }
```

Now, if we want to use the multiprocessing module instead, we need to redesign the script and move the code to be under a function; then, we will assign a number of processes equal to the number of devices (one process will connect to one device and execute the command) and set the target of the process to execute this function:

```
from netmiko import ConnectHandler
from devices import R1, SW1, SW2, SW3, SW4
import multiprocessing as mp
from datetime import datetime

nodes = [R1, SW1, SW2, SW3, SW4]

def connect_to_dev(device):

    net_connect = ConnectHandler(**device)
    output = net_connect.send_command("show run")
    print output

processes = []

start_time = datetime.now()
for device in nodes:
    print("Adding Process to the list")
    processes.append(mp.Process(target=connect_to_dev, args=[device]))

print("Spawning the Process")
for p in processes:
    p.start()

print("Joining the finished process to the main truck")
```

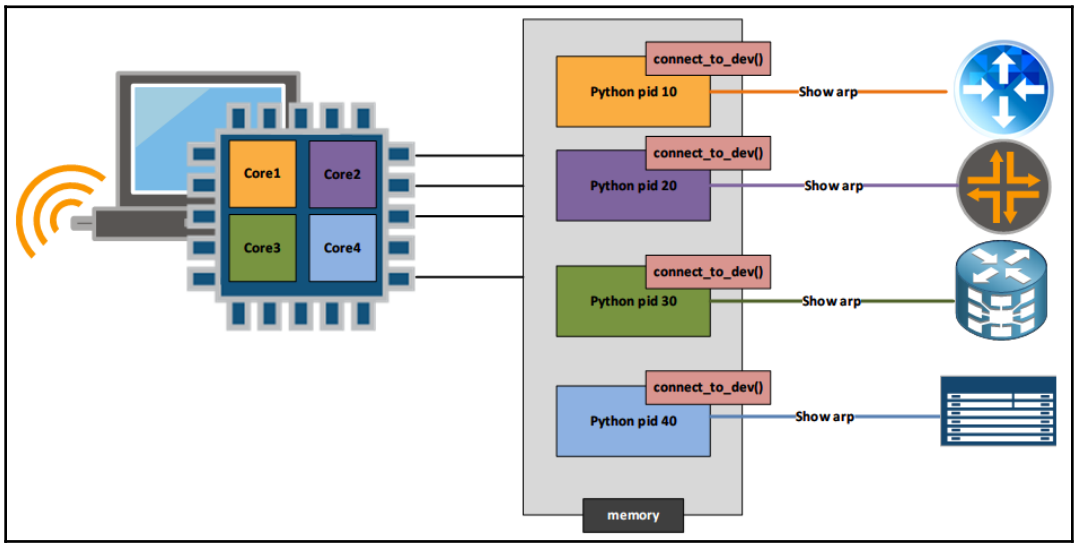
```
for p in processes:
    p.join()

end_time = datetime.now()
print("Script Execution tooks {}".format(end_time - start_time))
```

In the preceding example, the following applies:

- We imported a multiprocess module as `mp`. One of the most important classes available inside the module is `Process`, which takes our `netmiko connect` function as a target argument. Also, it accepts passing an argument to the target function.
- Then, we iterated over our nodes and created a process for each device and appended that process to the processes list.
- The `start()` method, which is available in the module, is used to spawn and then it starts the process execution.
- Finally, the script execution time is calculated by subtracting the script start time from the script end time.

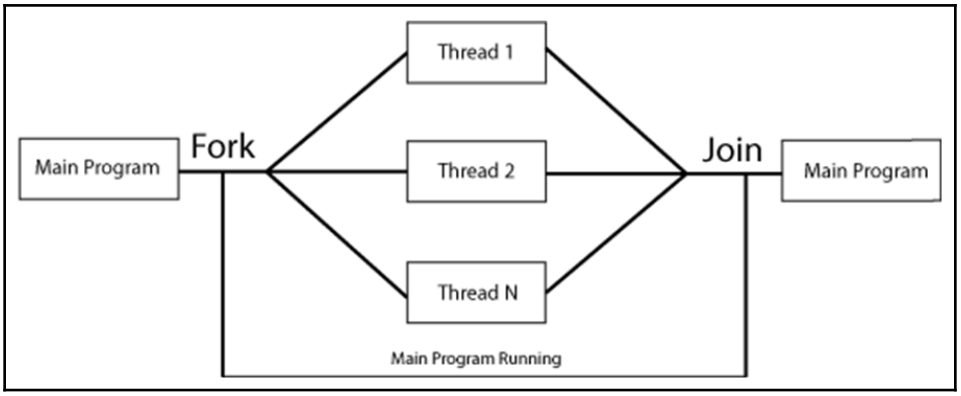
Behind the scenes, the main thread that executes the main script will start to fork a number of processes equal to the number of devices. Each of them targets one function that executes `show run` on all devices at the same time and stores the output in a variable, without affecting each other.

This is a sample view of the processes inside Python:



Now, when you execute the full code, one final thing needs to be done. You need to join the forked process to the main thread/truck, in order to smoothly finish the program's execution:

```
for p in processes:
    p.join()
```



The `join()` method used in the preceding example has nothing to do with the original `join()`, available as a string method; it's only used to join the process to the main thread.

Intercommunication between processes

Sometimes, you will have a process that needs to pass or exchange information with other processes during runtime. The multiprocessing module has a `Queue` class that implements a special list, within which a process can insert and consume data. There are two methods available inside of this class: `get()` and `put()`. The `put()` method is used to add data to the `Queue`, whereas getting data from the queue is done via the `get()` method. In the next example, we will use `Queue` to pass data from a subprocess to a parent process:

```
import multiprocessing
from netmiko import ConnectHandler
from devices import R1, SW1, SW2, SW3, SW4
from pprint import pprint

nodes = [R1, SW1, SW2, SW3, SW4]

def connect_to_dev(device, mp_queue):
```

```
        dev_id = device['ip']
        return_data = {}
        net_connect = ConnectHandler(**device)
        output = net_connect.send_command("show run")
        return_data[dev_id] = output
        print("Adding the result to the multiprocess queue")
        mp_queue.put(return_data)

    mp_queue = multiprocessing.Queue()
    processes = []

    for device in nodes:
        p = multiprocessing.Process(target=connect_to_dev, args=[device,
    mp_queue])
        print("Adding Process to the list")
        processes.append(p)
        p.start()

    for p in processes:
        print("Joining the finished process to the main truck")
        p.join()

    results = []
    for p in processes:
        print("Moving the result from the queue to the results list")
        results.append(mp_queue.get())

    pprint(results)
```

In the preceding example, the following applies:

- We imported another class, called `Queue()`, from the `multiprocess` module, and instantiated it into the `mp_queue` variable.
- Then, during the process creation, we appended this queue as an argument side-by-side with the device, so every process will have access to the same queue and be able to write data to it.
- The `connect_to_dev()` function connects to each device and executes the `show run` command on the Terminal, then writes the output to the shared queue.

Note that we formatted the output as dictionary items, `{ip:<command_output>}`, before adding it to the shared queue using `mp_queue.put()`.

- After the processes finished execution and joined the main (parent) process, we used `mp_queue.get()` to retrieve the queue items in a results list, then used `pprint` to prettyprint the output.

Summary

In this chapter, we learned about the Python multiprocessing library and how to instantiate and execute Python code in parallel.

In the next chapter, we will learn how to prepare a lab environment and explore automation options to speed up server deployment.

8
Preparing a Lab Environment

In this chapter, we will set a lab up by using two popular Linux distributions: CentOS and Ubuntu. CentOS is a community-driven Linux operating system that targets enterprise servers, and it's known for its compatibility with **Red Hat Enterprise Linux** (**RHEL**). Ubuntu is another Linux distribution that is based on the famous Debian operating system; it's currently developed by Canonical Ltd., which provides it with commercial support.

We will also learn how to install both Linux distributions with a free and open software called **Cobbler,** which will automatically boot the server with a Linux image and customize it using the `kickstart` for CentOS and Anaconda for Debian-based system.

The following topics will be covered in this chapter:

- Getting the Linux operating system
- Creating an automation machine on a hypervisor
- Getting started with Cobbler

Getting the Linux operating system

In the next sections, we are going to create two Linux machines, CentOS and Ubuntu, on different hypervisors. The machines will serve as the automation server in our environment.

Downloading CentOS

CentOS binaries can be downloaded through multiple methods. You can download them directly from multiple FTP servers around the world, or you can download them as torrents, from people who seed them. Also, CentOS is available in two flavors:

- Minimal ISO: Provides the basic server, with essential packages
- Everything ISO: Provides the server and all available packages from the main repositories

First, head to the CentOS project link (`https://www.centos.org/`) and click on the **Get CentOS Now** button, as shown in the following screenshot:

Then, choose the minimal ISO image, and download it from any available download site.

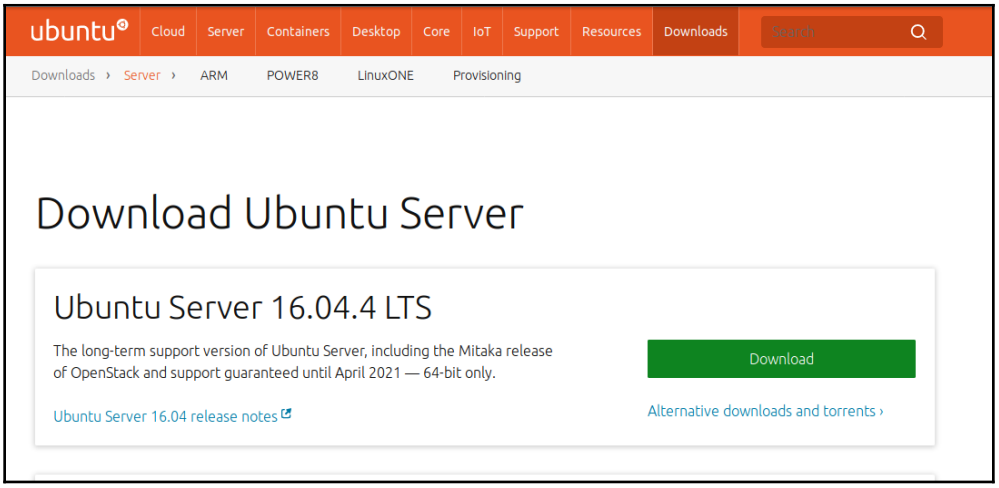
CentOS is available for multiple cloud providers, such as Google, Amazon, Azure, and Oracle Cloud. You can find all of the cloud images at `https://cloud.centos.org/centos/7/images/`.

Downloading Ubuntu

Ubuntu is widely known for providing a good desktop experience to end users. Canonical (the Ubuntu developers) work with many server vendors to certify Ubuntu on different hardware. Canonical also provide a server version for Ubuntu, which offers as many features as in 16.04, such as:

- Support from Canonical until 2021
- Ability to run on all major architectures—x86, x86-64, ARM v7, ARM64, POWER8, and IBM s390x (LinuxONE)
- Support for ZFS, a next generation volume management filesystem ideal for servers and containers
- LXD Linux container hypervisor enhancements, including QoS and resource controls (CPU, memory, block I/O, and storage quota)
- Installation snaps, for simple application installation and release management.
- First production release of DPDK—line speed kernel networking
- Linux 4.4 kernel and `systemd` service manager
- Certification as a guest on AWS, Microsoft Azure, Joyent, IBM, Google Cloud Platform, and Rackspace
- Updates for Tomcat (v8), PostgreSQL (v9.5), Docker v (1.10), Puppet (v3.8.5), QEMU (v2.5), Libvirt (v1.3.1), LXC (v2.0), MySQL (v5.6), and more

You can download the Ubuntu LTS by browsing to `https://www.ubuntu.com/download/server` and choosing **Ubuntu 16.04 LTS**:



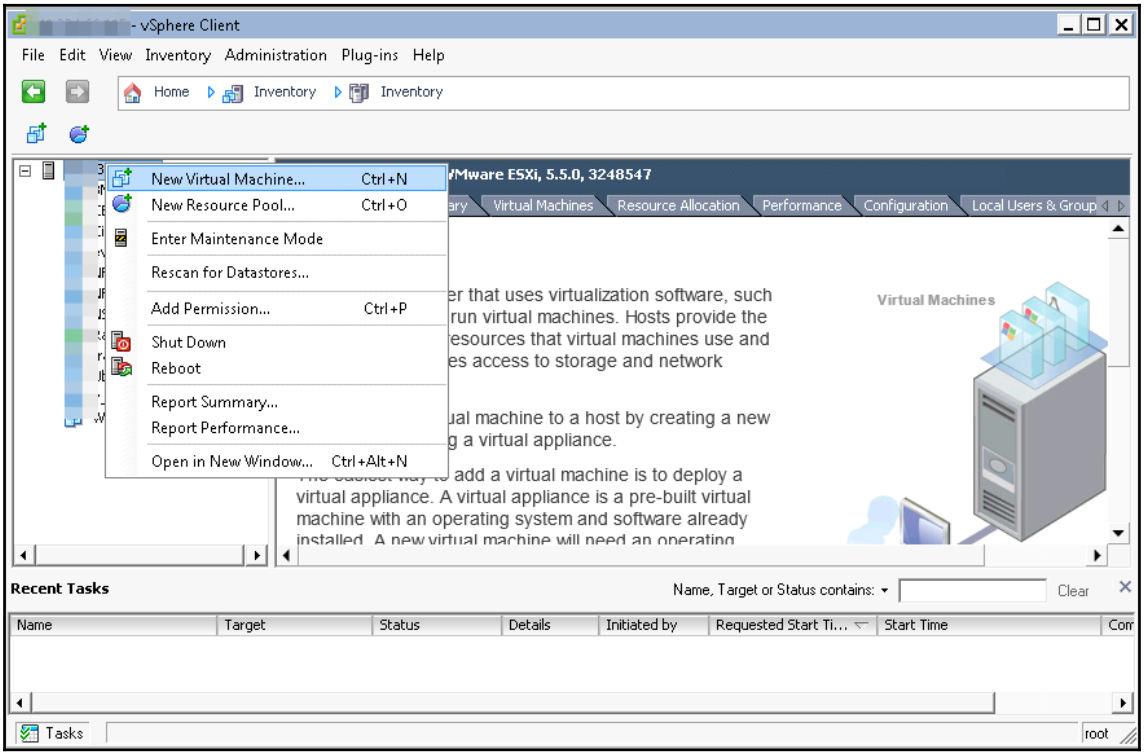
Creating an automation machine on a hypervisor

After downloading the ISO files, we will create a Linux machine over VMware ESXi and KVM hypervisors.

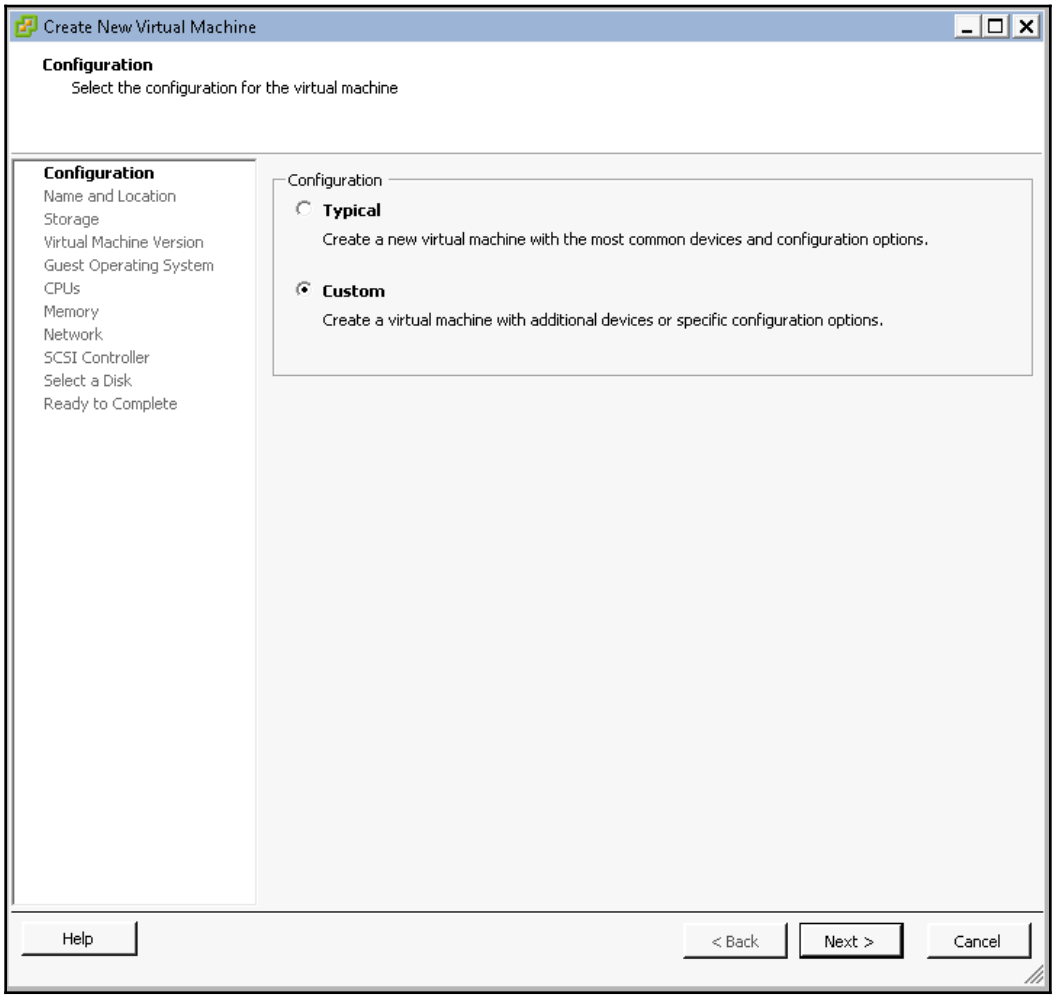
Creating a Linux machine over VMware ESXi

We will use the VMware vSphere client to create a virtual machine. Log in to one of the available ESXi servers using root credentials. First, you will need to upload either the Ubuntu or CentOS ISO to the VMware data store. Then, follow these steps to create the machine:

1. Right-click on the server name and choose **New Virtual Machine**:

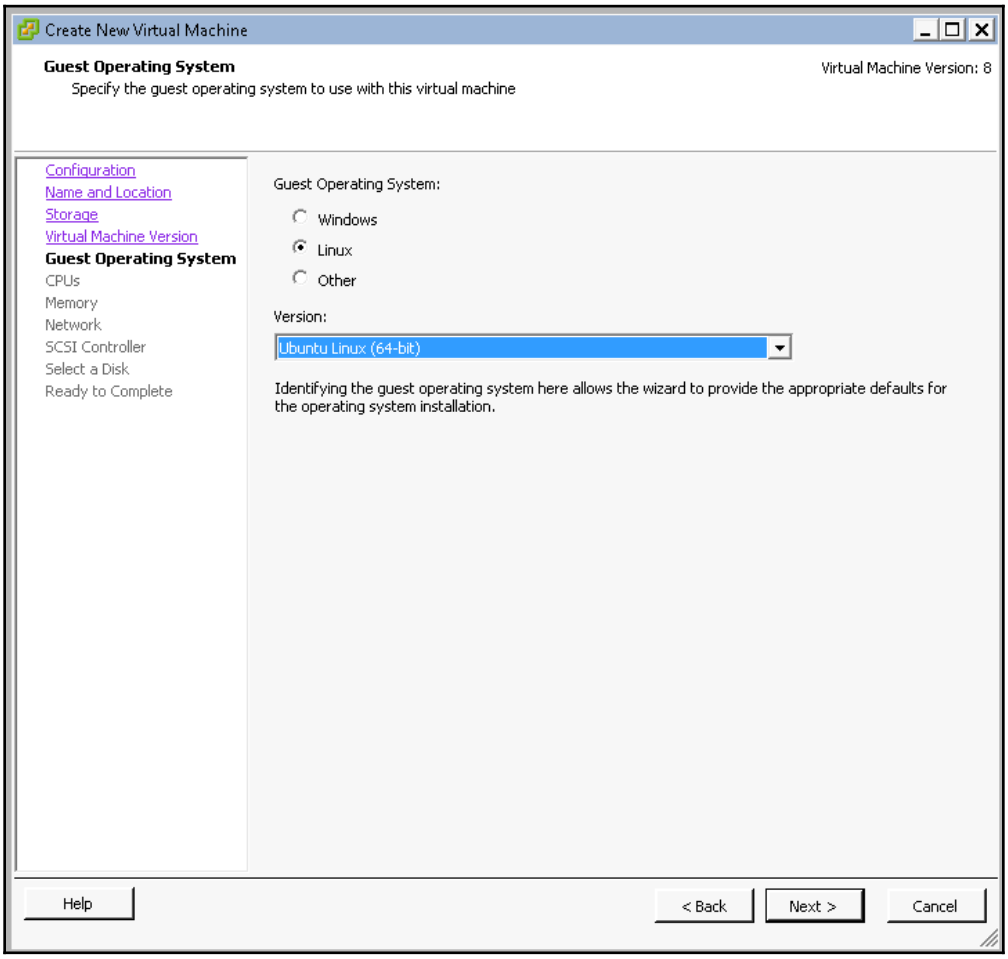


2. Choose a **Custom** installation, so that you will have more options during the installation:



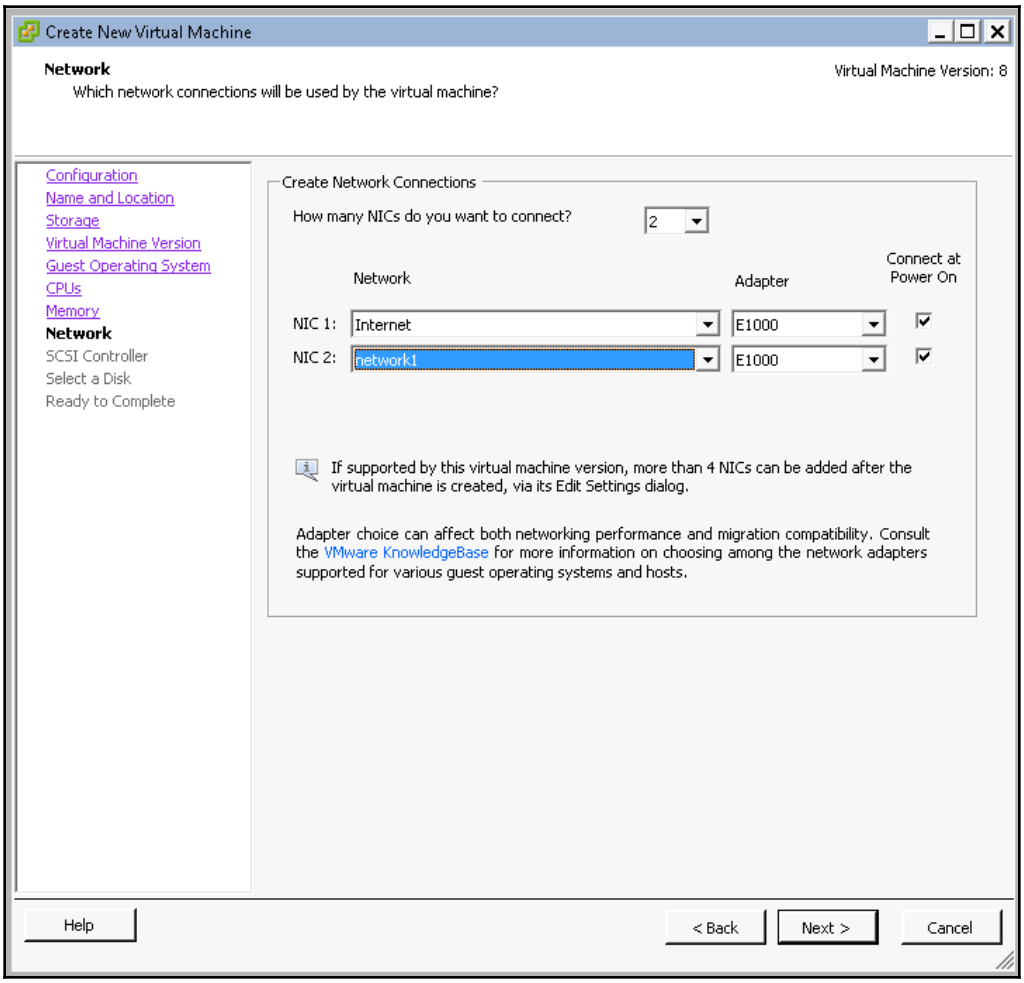
3. Provide a name for the VM: **AutomationServer**.
4. Choose the machine version: **8**.
5. Choose the data store on which the machine will be created.

6. Choose the guest operating system: either **Ubuntu Linux (64-bit)** or **Red Hat version 6/7**:



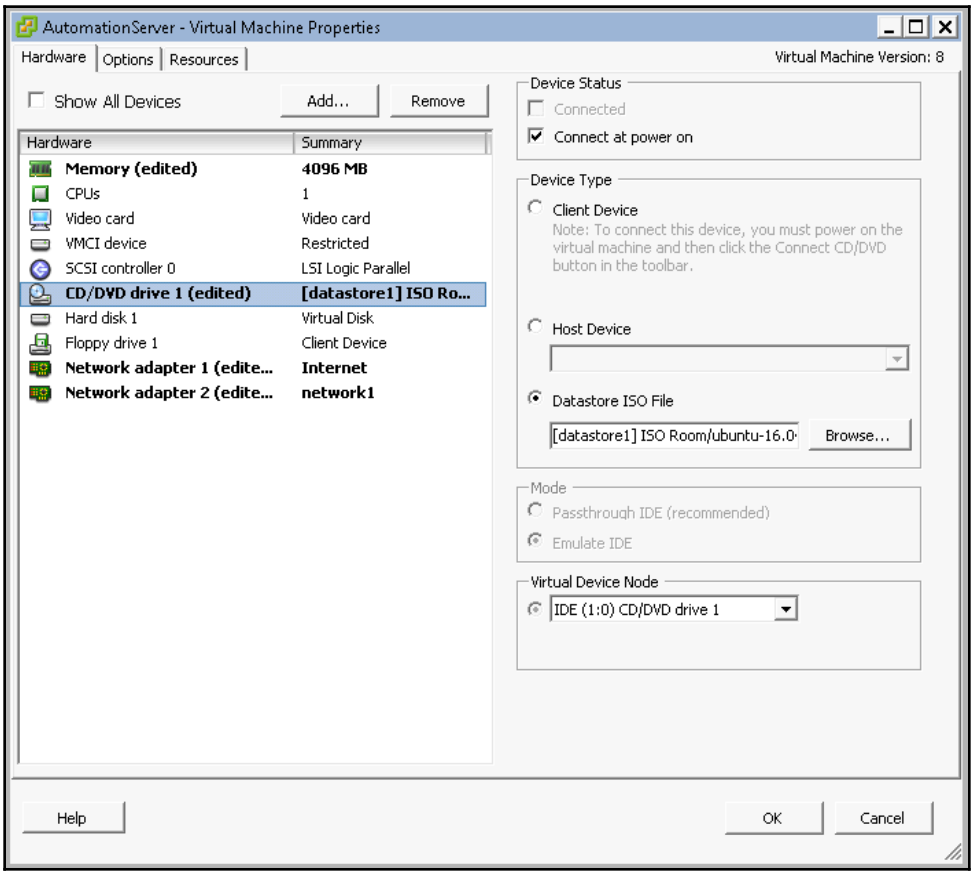
7. The VM specification shouldn't have less than 2 vCPU and 4 GB RAM, in order to have efficient performance. Select them in the **CPU** and **Memory** tabs respectively.

8. In the **Network** tab, select two interfaces with **E1000** adapters. One of these interfaces will connect to the internet, and the second interface will manage the clients:

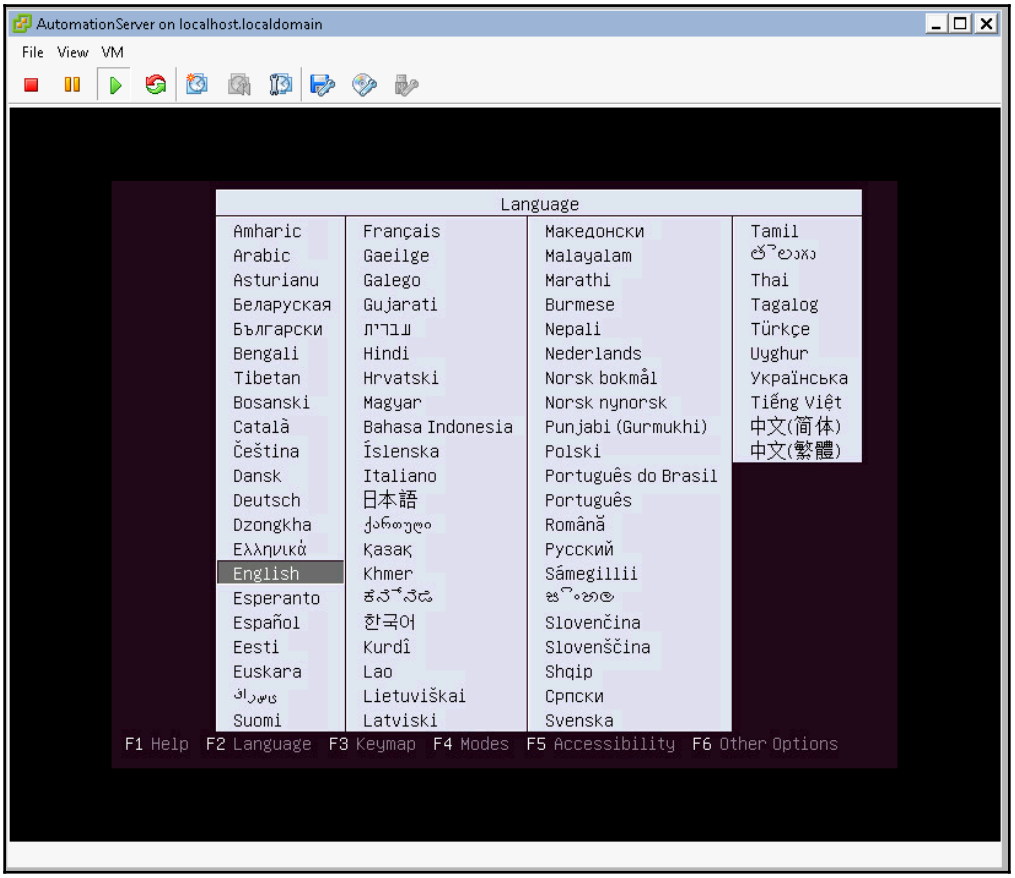


9. Choose the default SCSI controller for the system. In my case, it will be **LSI logical parallel**.
10. Select a **Create a new virtual disk** and provide **20 GB** as the disk size for the VM.

11. Now the virtual machine is ready, and you can start the Linux OS installation. Associate the uploaded image to the CD/DVD drive, and make sure that the **Connect at power on** option is selected:



Once it starts running, you will be asked to choose a language:

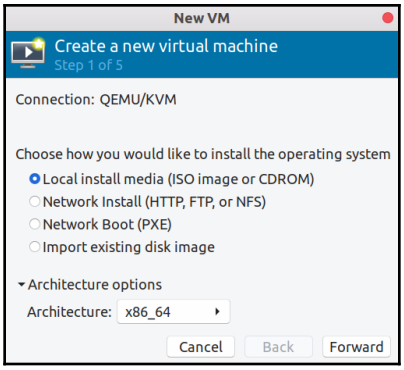


Complete the CentOS/Ubuntu installation steps as usual.

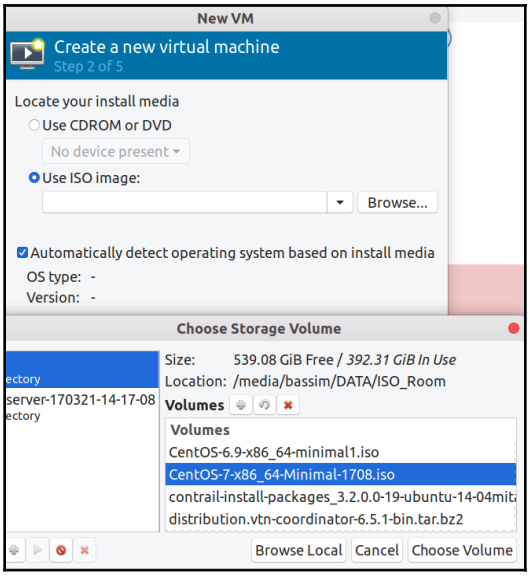
Creating a Linux machine over KVM

We will use the `virt-manager` utility, available in KVM, to launch the desktop administration for KVM. We will then create a new VM:

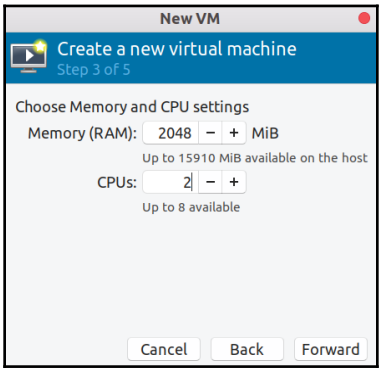
1. Here, we will choose the installation method as **Local install media (ISO image or CDROM):**



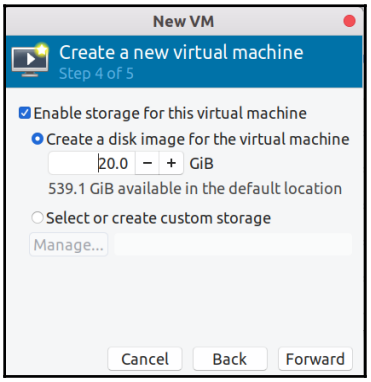
2. Then, we will click on **Browse** and choose the previously downloaded image (CentOS or Ubuntu). You will notice that the KVM successfully detects the OS type and version:



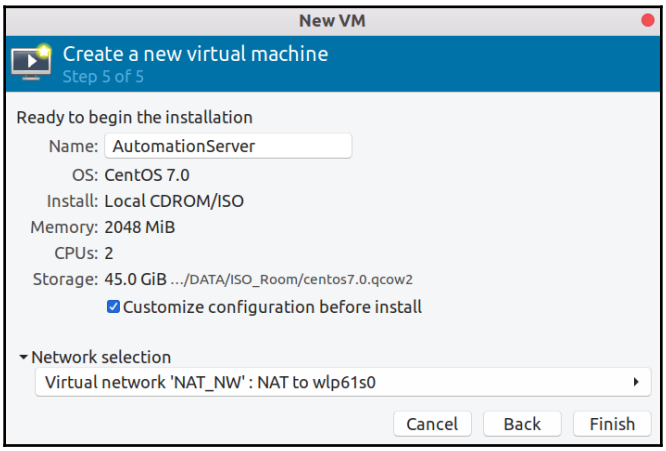
3. Then, we will choose the machine specifications in terms of CPUs, memory, and storage:



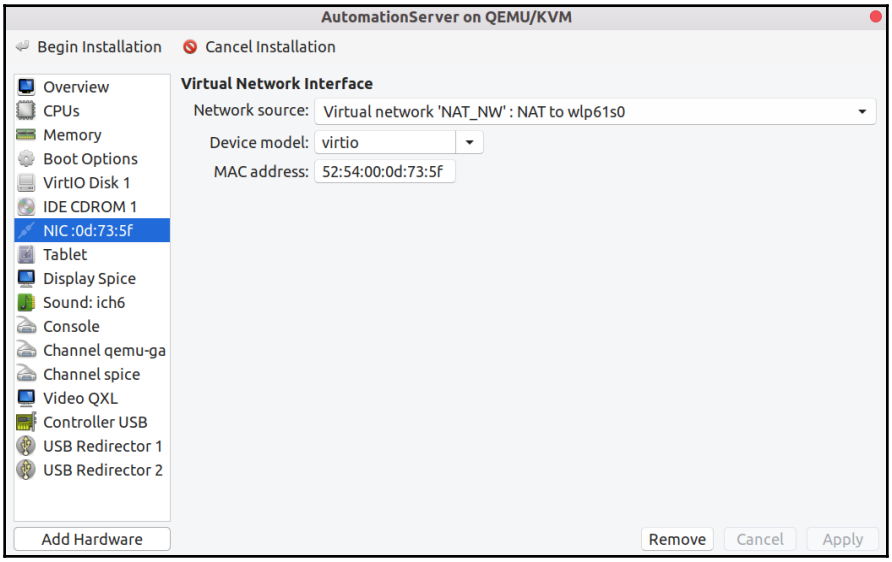
4. Choose the appropriate storage space for your machine:



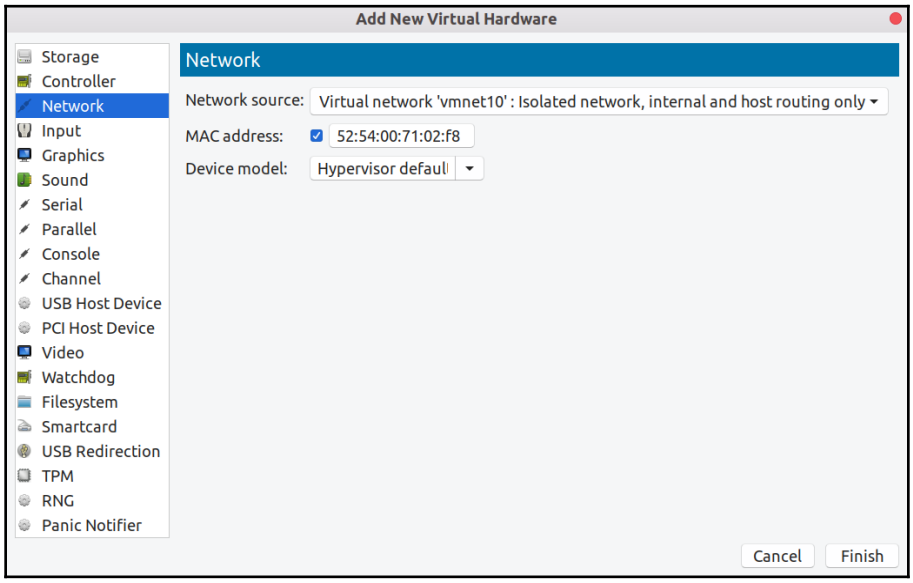
5. The final step is to choose a name, and then click on the **Customize Configuration before install** option, in order to add an additional network interface to the automation server. Then, click on **Finish:**



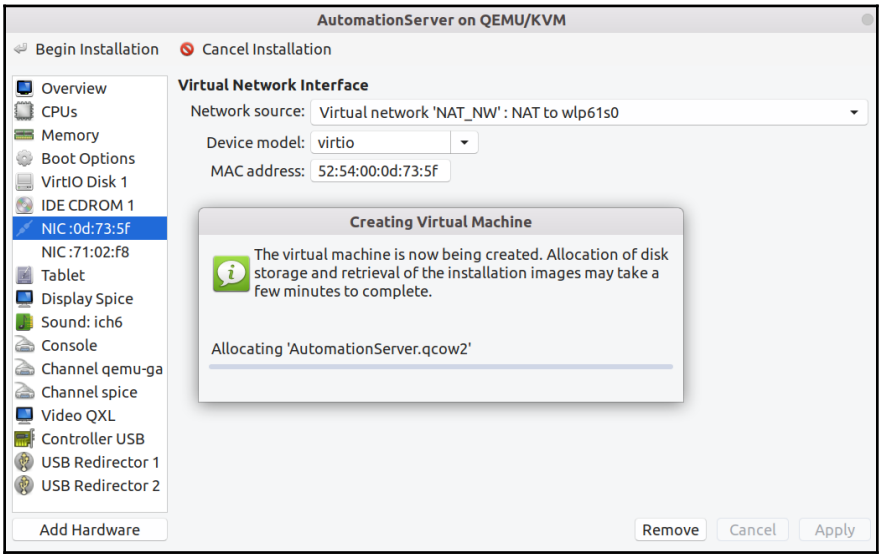
Another window is open, which contains all of the specs for the machine. Click on **Add Hardware,** then choose the **Network**:



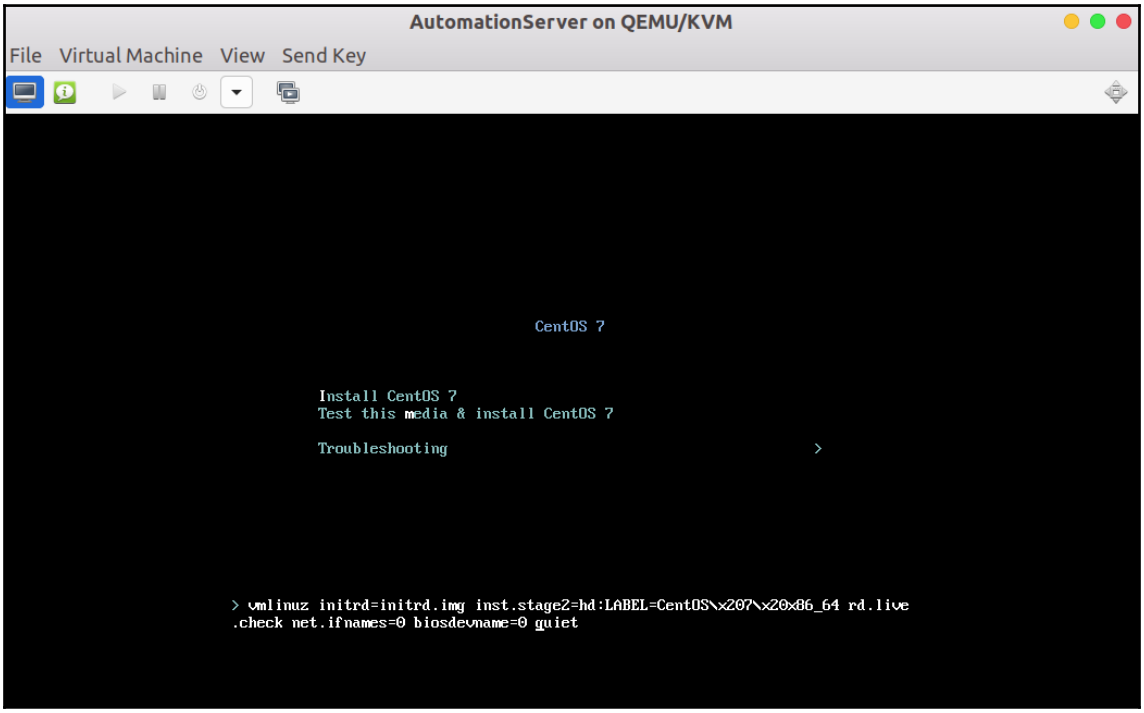
We will add another network interface to communicate with the clients. The first network interface is using NAT to connect to the internet through the physical wireless NIC:



Finally, click on **Begin Installation** on the main window so that the KVM will start allocating the hard disk and attaching the ISO image to the virtual machine:



Once it has finished, you will see the following screen:



Complete the CentOS/Ubuntu installation steps as usual.

Getting started with Cobbler

Cobbler is a piece of open source software, used for unattended network-based installation. It leverages multiple tools, such as DHCP, FTP, PXE, and other open source tools (we will explain them later), so that you will have a one-stop shop for automating the OS installation. The target machine (bare metal or a virtual machine) has to support booting from a network on its **network interface card** (**NIC**). This function enables the machine to send a DHCP request that hits the Cobbler server, which will take care of the rest.

You can read more about the project on its GitHub page (`https://github.com/cobbler/cobbler`).

Understanding how Cobbler works

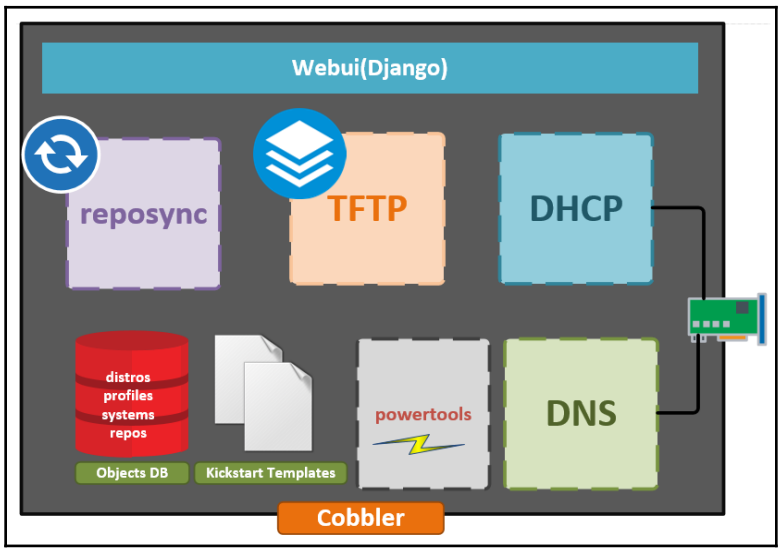
Cobbler depends on multiple tools to provide the **Preboot eXecution Environment** (**PXE**) functionality to clients. First, it depends on the DHCP service that receives the DHCP broadcast message from the client upon powering on; then, it replies with an IP address, a subnet mask, the next server (TFTP), and finally, the `pxeLinux.0`, which is the loader filename that the client is requesting when it initially sends the DHCP message to the server.

The second tool is the TFTP server that hosts `pxeLinux.0` and different distribution images.

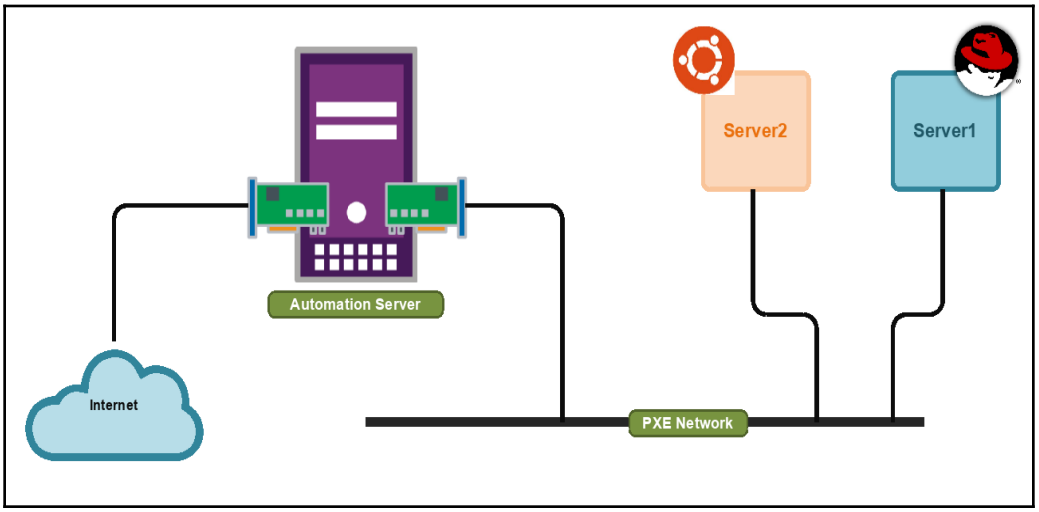
The third tool is the template rendering utility. Cobbler uses `cheetah`, which is an open source template engine developed in Python and has its own DSL (domain specific language) format. We will use it to generate the `kickstart` files.

Kickstart files are used to automate the installation of Red Hat based distributions, like CentOS, Red Hat, and Fedora. It also has limited support for rendering the `Anaconda` files used for installing Debian-based systems.

There are also additional tools. `reposync` is used to mirror an online repository from the internet to a local directory inside of Cobbler, making it available to the client. `ipmitools` remotely manages powering different server hardware on and off:



In the following topology, Cobbler is hosted on the automation server installed previously, and will connect to a couple of servers. We will install Ubuntu and Red Hat on them, through Cobbler. The automation server has another interface that connects directly to the internet, in order to download some additional packages that are required by Cobbler, as we will see in the next section:



Server	**IP Address**
Automation Server (with cobbler installed)	`10.10.10.130`
Server1 (CentOS Machine)	IP from range `10.10.10.5-10.10.10.10`
Server 2 (Ubuntu Machine)	IP from range `10.10.10.5-10.10.10.10`

Installing Cobbler on an automation server

We will start by installing some essential packages, such as `vim`, `tcpudump` , `wget`, and `net-tools`, on our automation server (either CentOS or Ubuntu). Then, we will install the `cobbler` package from the `epel` repository. Please note that these packages are not required for Cobbler, but we will use them to understand how Cobbler really works.

For CentOS, use the following command:

```
yum install vim vim-enhanced tcpdump net-tools wget git -y
```

For Ubuntu, use the following command:

```
sudo apt install vim tcpdump net-tools wget git -y
```

Then, we need to disable the firewall. Cobbler doesn't play well with SELinux policies, and it's recommended to disable it, especially if you are unfamiliar with them. Also, we will disable `iptables` and `firewalld`, since we are in a lab, not production.

For CentOS, use the following command:

```
# Disable firewalld service
systemctl disable firewalld
systemctl stop firewalld

# Disable IPTables service
systemctl disable iptables.service
systemctl stop iptables.service

# Set SELinux to permissive instead of enforcing
sed -i s/^SELinux=.*$/SELinux=permissive/ /etc/seLinux/config
setenforce 0
```

For Ubuntu, use the following command:

```
# Disable ufw service
sudo ufw disable

# Disable IPTables service
sudo iptables-save > $HOME/BeforeCobbler.txt
sudo iptables -X
sudo iptables -t nat -F
sudo iptables -t nat -X
sudo iptables -t mangle -F
sudo iptables -t mangle -X
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT

# Set SELinux to permissive instead of enforcing
sed -i s/^SELinux=.*$/SELinux=permissive/ /etc/seLinux/config
setenforce 0
```

Finally, reboot the automation server machine for the changes to take effect:

```
reboot
```

Now, we will install the `cobbler` package. The software is available in the `epel` repository (but we need to install it first) in the case of CentOS. Ubuntu doesn't have the software available in upstream repositories, so we will download the source code and compile it on the platform.

For CentOS, use the following command:

```
# Download and Install EPEL Repo
yum install epel-release -y

# Install Cobbler
yum install cobbler -y

#Install cobbler Web UI and other dependencies
yum install cobbler-web dnsmasq fence-agents bind xinetd pykickstart -y
```

The current version of Cobbler, at the time of writing this book, is 2.8.2, which was released on September 16, 2017. For Ubuntu, we will clone the latest package from the GIT repository and build it from the source:

```
#install the dependencies as stated in
(http://cobbler.github.io/manuals/2.8.0/2/1_-_Prerequisites.html)

sudo apt-get install createrepo apache2 mkisofs libapache2-mod-wsgi mod_ssl
python-cheetah python-netaddr python-simplejson python-urlgrabber python-
yaml rsync sysLinux atftpd yum-utils make python-dev python-setuptools
python-django -y

#Clone the cobbler 2.8 from the github to your server (require internet)
git clone https://github.com/cobbler/cobbler.git
cd cobbler

#Checkout the release28 (latest as the developing of this book)
git checkout release28

#Build the cobbler core package
make install

#Build cobbler web
make webtest
```

After successfully installing Cobbler on our machine, we will need to customize it to change the default settings to adapt to our network environment. We will need to change the following:

- Choose either the `bind` or `dnsmasq` module to manage DNS queries
- Choose either the `isc` or `dnsmaasq` module to serve incoming DHCP requests from clients
- Configure the TFTP Cobbler IP address (it will usually be a static address in Linux).
- Provide the DHCP range that serves the clients
- Restart the services to apply the configuration

Let's take a step-by-step look at the configuration:

1. Choose `dnsmasq` as the DNS server:

```
vim /etc/cobbler/modules.conf
[dns]
module = manage_dnsmasq
vim /etc/cobbler/settings
manage_dns: 1
restart_dns: 1
```

2. Choose `dnsmasq` for managing the DHCP service:

```
vim /etc/cobbler/modules.conf

[dhcp]
module = manage_dnsmasq
vim /etc/cobbler/settings
manage_dhcp: 1
restart_dhcp: 1
```

3. Configure the Cobbler IP address as the TFTP server:

```
vim /etc/cobbler/settings
server: 10.10.10.130
next_server: 10.10.10.130
vim /etc/xinetd.d/tftp
 disable                 = no
```

   Also, enable PXE boot loop prevention by setting the `pxe_just_once` to `0`:

```
pxe_just_once: 0
```

4. Add the client `dhcp-range` in the `dnsmasq` service template:

```
vim /etc/cobbler/dnsmasq.template
dhcp-range=10.10.10.5,10.10.10.10,255.255.255.0
```

   Note the line that says `dhcp-option=66,$next_server`. This means that Cobbler will pass `next_server`, previously configured in the settings as the TFTP boot server, to any clients requesting an IP address through the DHCP service provided by `dnsmasq`.

5. Enable and restart the services:

```
systemctl enable cobblerd
systemctl enable httpd
systemctl enable dnsmasq

systemctl start cobblerd
systemctl start httpd
systemctl start dnsmasq
```

Provisioning servers through Cobbler

We are now a few steps away from having our first server up and running through Cobbler. Basically, we need to tell Cobbler our clients' MAC addresses and which operating systems they have:

1. Import the Linux ISO. Cobbler will automatically analyze the image and create a profile for it:

```
cobbler import --arch=x86_64 --path=/mnt/cobbler_images --
name=CentOS-7-x86_64-Minimal-1708

task started: 2018-03-28_132623_import
task started (id=Media import, time=Wed Mar 28 13:26:23 2018)
Found a candidate signature: breed=redhat, version=rhel6
Found a candidate signature: breed=redhat, version=rhel7
Found a matching signature: breed=redhat, version=rhel7
Adding distros from path /var/www/cobbler/ks_mirror/CentOS-7-
x86_64-Minimal-1708-x86_64:
creating new distro: CentOS-7-Minimal-1708-x86_64
trying symlink: /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64 -> /var/www/cobbler/links/CentOS-7-
Minimal-1708-x86_64
creating new profile: CentOS-7-Minimal-1708-x86_64
associating repos
```

```
checking for rsync repo(s)
checking for rhn repo(s)
checking for yum repo(s)
starting descent into /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64 for CentOS-7-Minimal-1708-x86_64
processing repo at : /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64
need to process repo/comps: /var/www/cobbler/ks_mirror/CentOS-7-
x86_64-Minimal-1708-x86_64
looking for /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64/repodata/*comps*.xml
Keeping repodata as-is :/var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64/repodata
*** TASK COMPLETE ***
```

You may need to mount the Linux ISO image before importing it to a mount point, by using `mount -O loop /root/<image_iso> /mnt/cobbler_images/`.

You can run the `cobbler profile report` command to check the created profile:

```
cobbler profile report

Name                           : CentOS-7-Minimal-1708-x86_64
TFTP Boot Files                : {}
Comment                        :
DHCP Tag                       : default
Distribution                   : CentOS-7-Minimal-1708-x86_64
Enable gPXE?                   : 0
Enable PXE Menu?               : 1
Fetchable Files                : {}
Kernel Options                 : {}
Kernel Options (Post Install)  : {}
Kickstart                      :
/var/lib/cobbler/kickstarts/sample_end.ks
Kickstart Metadata             : {}
Management Classes             : []
Management Parameters          : <<inherit>>
Name Servers                   : []
Name Servers Search Path       : []
Owners                         : ['admin']
Parent Profile                 :
Internal proxy                 :
Red Hat Management Key         : <<inherit>>
Red Hat Management Server      : <<inherit>>
```

```
Repos                          : []
Server Override                : <<inherit>>
Template Files                 : {}
Virt Auto Boot                 : 1
Virt Bridge                    : xenbr0
Virt CPUs                      : 1
Virt Disk Driver Type          : raw
Virt File Size(GB)             : 5
Virt Path                      :
Virt RAM (MB)                  : 512
Virt Type                      : kvm
```

You can see that the `import` command filled many fields automatically, such as `Kickstart`, `RAM`, `operating system`, and the `initrd/kernel` file locations.

2. Add any additional repositories to the profile (optional):

```
cobbler repo add --
mirror=https://dl.fedoraproject.org/pub/epel/7/x86_64/ --name=epel-
local --priority=50 --arch=x86_64 --breed=yum

cobbler reposync
```

Now, edit the profile, and add the created repository to the list of available repositories:

```
cobbler profile edit --name=CentOS-7-Minimal-1708-x86_64 --
repos="epel-local"
```

3. Add a client MAC address and link it to the created profile:

```
cobbler system add --name=centos_client --profile=CentOS-7-
Minimal-1708-x86_64  --mac=00:0c:29:4c:71:7c --ip-
address=10.10.10.5 --subnet=255.255.255.0 --static=1 --
hostname=centos-client  --gateway=10.10.10.1 --name-servers=8.8.8.8
--interface=eth0
```

The `--hostname` field corresponds to the local system name and configures the client networking using the `--ip-address`, `--subnet`, and `--gateway` options. This will make Cobbler generate a `kickstart` file with these options.

If you need to customize the server and add additional packages, configure firewall, ntp, and configure partitions and hard disk layout then you can add these settings to the `kickstart` file. Cobbler provide sample file under `/var/lib/cobbler/kickstarts/sample.ks`, which you can copy to another folder and provide in the `--kickstart` parameter in the previous command.

You can integrate Ansible inside the `kickstart` file by running Ansible in pull mode (instead the default push mode). Ansible will download the playbook from an online GIT repository (such as GitHub or GitLab) and will execute it after that.

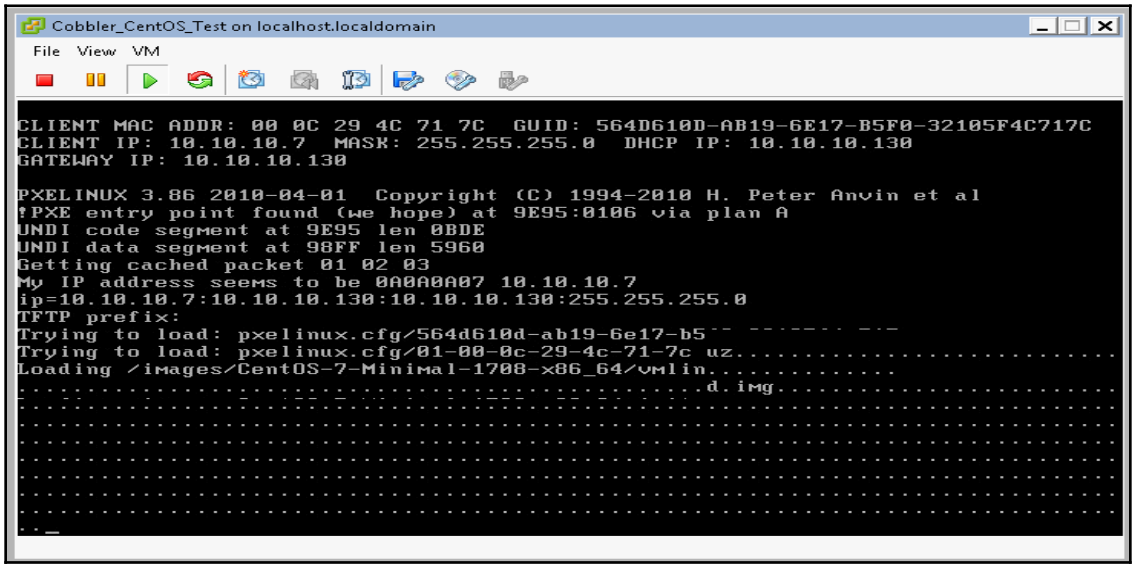
4. Instruct Cobbler to generate the configuration files required to serve our client and to update the internal database with the new information by using the following commands:

```
#cobbler sync

task started: 2018-03-28_141922_sync
task started (id=Sync, time=Wed Mar 28 14:19:22 2018)
running pre-sync triggers
cleaning trees
removing: /var/www/cobbler/images/CentOS-7-Minimal-1708-x86_64
removing: /var/www/cobbler/images/Ubuntu_Server-x86_64
removing: /var/www/cobbler/images/Ubuntu_Server-hwe-x86_64
removing: /var/lib/tftpboot/pxeLinux.cfg/default
removing: /var/lib/tftpboot/pxeLinux.cfg/01-00-0c-29-4c-71-7c
removing: /var/lib/tftpboot/grub/01-00-0C-29-4C-71-7C
removing: /var/lib/tftpboot/grub/efidefault
removing: /var/lib/tftpboot/grub/grub-x86_64.efi
removing: /var/lib/tftpboot/grub/images
removing: /var/lib/tftpboot/grub/grub-x86.efi
removing: /var/lib/tftpboot/images/CentOS-7-Minimal-1708-x86_64
removing: /var/lib/tftpboot/images/Ubuntu_Server-x86_64
removing: /var/lib/tftpboot/images/Ubuntu_Server-hwe-x86_64
removing: /var/lib/tftpboot/s390x/profile_list
copying bootloaders
trying hardlink /var/lib/cobbler/loaders/grub-x86_64.efi ->
/var/lib/tftpboot/grub/grub-x86_64.efi
trying hardlink /var/lib/cobbler/loaders/grub-x86.efi ->
/var/lib/tftpboot/grub/grub-x86.efi
copying distros to tftpboot
copying files for distro: Ubuntu_Server-x86_64
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/netboot/ubuntu-installer/amd64/Linux ->
/var/lib/tftpboot/images/Ubuntu_Server-x86_64/Linux
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
```

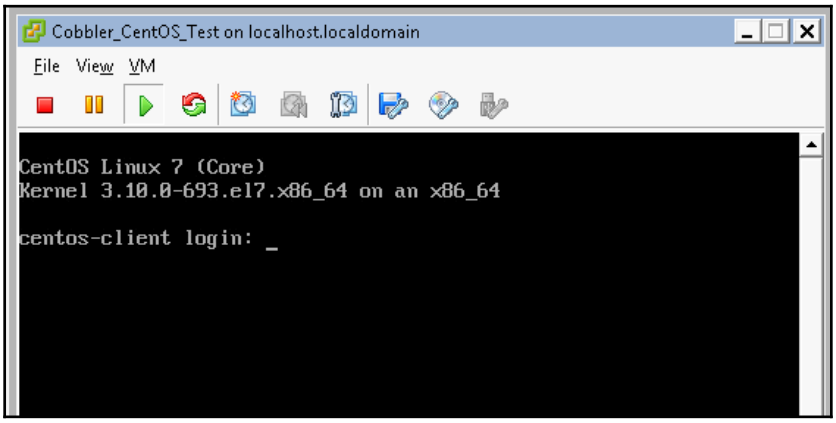
```
x86_64/install/netboot/ubuntu-installer/amd64/initrd.gz ->
/var/lib/tftpboot/images/Ubuntu_Server-x86_64/initrd.gz
copying files for distro: Ubuntu_Server-hwe-x86_64
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/hwe-netboot/ubuntu-installer/amd64/Linux ->
/var/lib/tftpboot/images/Ubuntu_Server-hwe-x86_64/Linux
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/hwe-netboot/ubuntu-installer/amd64/initrd.gz ->
/var/lib/tftpboot/images/Ubuntu_Server-hwe-x86_64/initrd.gz
copying files for distro: CentOS-7-Minimal-1708-x86_64
trying hardlink /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64/images/pxeboot/vmlinuz ->
/var/lib/tftpboot/images/CentOS-7-Minimal-1708-x86_64/vmlinuz
trying hardlink /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64/images/pxeboot/initrd.img ->
/var/lib/tftpboot/images/CentOS-7-Minimal-1708-x86_64/initrd.img
copying images
generating PXE configuration files
generating: /var/lib/tftpboot/pxeLinux.cfg/01-00-0c-29-4c-71-7c
generating: /var/lib/tftpboot/grub/01-00-0C-29-4C-71-7C
generating PXE menu structure
copying files for distro: Ubuntu_Server-x86_64
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/netboot/ubuntu-installer/amd64/Linux ->
/var/www/cobbler/images/Ubuntu_Server-x86_64/Linux
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/netboot/ubuntu-installer/amd64/initrd.gz ->
/var/www/cobbler/images/Ubuntu_Server-x86_64/initrd.gz
Writing template files for Ubuntu_Server-x86_64
copying files for distro: Ubuntu_Server-hwe-x86_64
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/hwe-netboot/ubuntu-installer/amd64/Linux ->
/var/www/cobbler/images/Ubuntu_Server-hwe-x86_64/Linux
trying hardlink /var/www/cobbler/ks_mirror/Ubuntu_Server-
x86_64/install/hwe-netboot/ubuntu-installer/amd64/initrd.gz ->
/var/www/cobbler/images/Ubuntu_Server-hwe-x86_64/initrd.gz
Writing template files for Ubuntu_Server-hwe-x86_64
copying files for distro: CentOS-7-Minimal-1708-x86_64
trying hardlink /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64/images/pxeboot/vmlinuz ->
/var/www/cobbler/images/CentOS-7-Minimal-1708-x86_64/vmlinuz
trying hardlink /var/www/cobbler/ks_mirror/CentOS-7-x86_64-
Minimal-1708-x86_64/images/pxeboot/initrd.img ->
/var/www/cobbler/images/CentOS-7-Minimal-1708-x86_64/initrd.img
Writing template files for CentOS-7-Minimal-1708-x86_64
rendering DHCP files
rendering DNS files
rendering TFTPD files
```

```
generating /etc/xinetd.d/tftp
processing boot_files for distro: Ubuntu_Server-x86_64
processing boot_files for distro: Ubuntu_Server-hwe-x86_64
processing boot_files for distro: CentOS-7-Minimal-1708-x86_64
cleaning link caches
running post-sync triggers
running python triggers from /var/lib/cobbler/triggers/sync/post/*
running python trigger cobbler.modules.sync_post_restart_services
running: service dnsmasq restart
received on stdout:
received on stderr: Redirecting to /bin/systemctl restart
dnsmasq.service

running shell triggers from /var/lib/cobbler/triggers/sync/post/*
running python triggers from /var/lib/cobbler/triggers/change/*
running python trigger cobbler.modules.scm_track
running shell triggers from /var/lib/cobbler/triggers/change/*
*** TASK COMPLETE ***
```

Once you have started the CentOS client, you will notice that it goes to the PXE process and sends a DHCP message over `PXE_Network`. Cobbler will respond with an IP address, a `PXELinux0` file, and the required image assigned to that MAC address:



After Cobbler finishes the CentOS installation, you will see the hostname correctly configured in the machine:



You can go through the same steps for an Ubuntu machine.

Summary

In this chapter, you learned how to prepare a lab environment by installing two Linux machines (CentOS and Ubuntu) over a hypervisor. We then explored automation options, and sped up server deployment by installing Cobbler.

In the next chapter, you will learn how to send commands from a Python script directly to an operating system shell and investigate the output returned.

9
Using the Subprocess Module

Running and spawning a new system process can be useful to system administrators who want to automate specific operating system tasks or execute a few commands within their scripts. Python provides many libraries to call external system utilities, and it interacts with the data produced. The first library that was created is the `OS` module, which provides some useful tools to invoke external processes, such as `os.system`, `os.spwan`, and `os.popen*`. It lacks some essential functions, however, so Python developers have introduced a new library, `subprocess`, which can spawn new processes, send and receive from the processes, and handle error and return codes. Currently, the official Python documentation recommends the `subprocess` module for accessing system commands, and Python actually intends to replace the older modules with it.

The following topics will be covered in this chapter:

- The `Popen()` Subprocess
- Reading `stdin`, `stdout`, and `stderr`
- The subprocess call suite

The popen() subprocess

The `subprocess` module implements only one class: `popen()`. The primary use of this class is to spawn a new process on the system. This class can accept additional arguments for the running process, along with additional arguments for `popen()` itself:

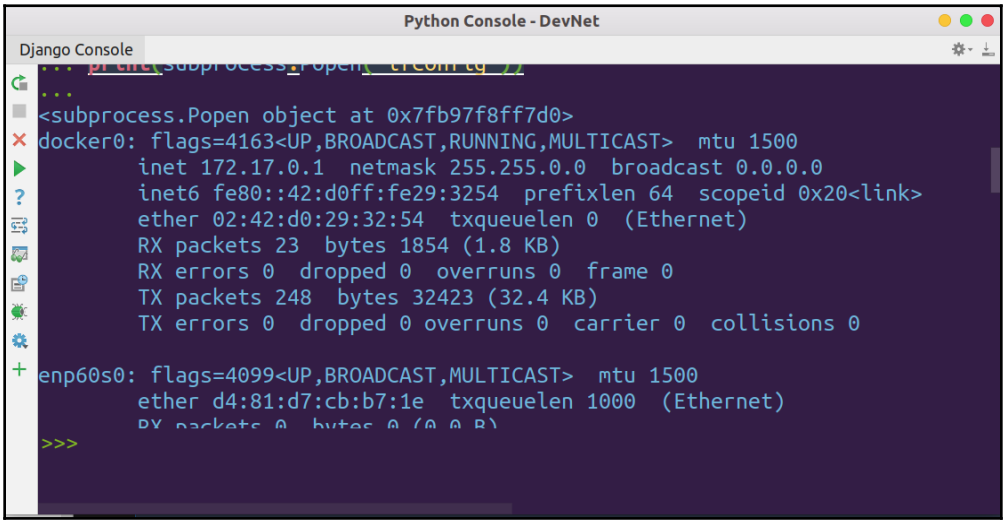
Arguments	**Meaning**
`args`	A string, or a sequence of program arguments.
`bufsize`	It is supplied as the buffering argument to the `open()` function when creating the `stdin`/`stdout`/`stderr` pipe file objects.
`executable`	A replacement program to execute.

`stdin`, `stdout`, `stderr`	These specify the executed program's standard input, standard output, and standard error file handles, respectively.
`shell`	If `True`, the command will be executed through the shell (the default is `False`). In Linux, this means calling the `/bin/sh` before running the child process.
`cwd`	Sets the current directory before the child is executed.
`env`	Defines the environmental variables for the new process.

Now, let us focus on `args`. The `popen()` command can take a Python list as an input, with the first element treated as the command and the subsequent elements as the command `args`, as shown in the following code snippet:

```
import subprocess
print(subprocess.Popen("ifconfig"))
```

Script output



The output returned from the command is printed directly to your Python Terminal.

The `ifconfig` is a Linux utility used to return the network interface information. For Windows users, you can get similar output by using the `ipconfig` command on cmd.

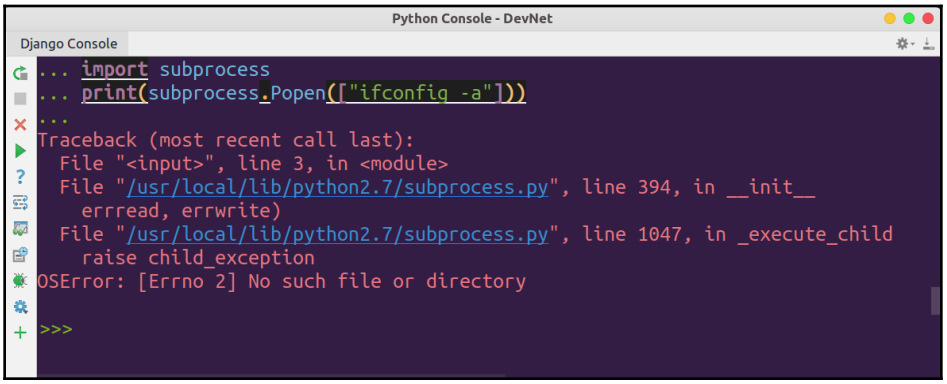
We can rewrite the preceding code and use a list instead of a string, as seen in the following code snippet:

```
print(subprocess.Popen(["ifconfig"]))
```

Using this approach allows you to add additional arguments to the main command as list items:

```
print(subprocess.Popen(["sudo", "ifconfig", "enp60s0:0", "10.10.10.2",
"netmask", "255.255.255.0", "up"]))

enp60s0:0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.10.10.2  netmask 255.255.255.0  broadcast 10.10.10.255
        ether d4:81:d7:cb:b7:1e  txqueuelen 1000  (Ethernet)
        device interrupt 16
```

Note that if you provide the previous command as a string not as a list, as we did in the first example, the command will fail as shown in below screenshot. The subprocess `Popen()` expects an executable name in each list element and not any other arguments.



On the other hand, if you want to use the string method instead of a list, you can set the `shell` argument to `True`. This will instruct `Popen()` to append `/bin/sh` before the command; hence, the command will be executed with all of the arguments after it:

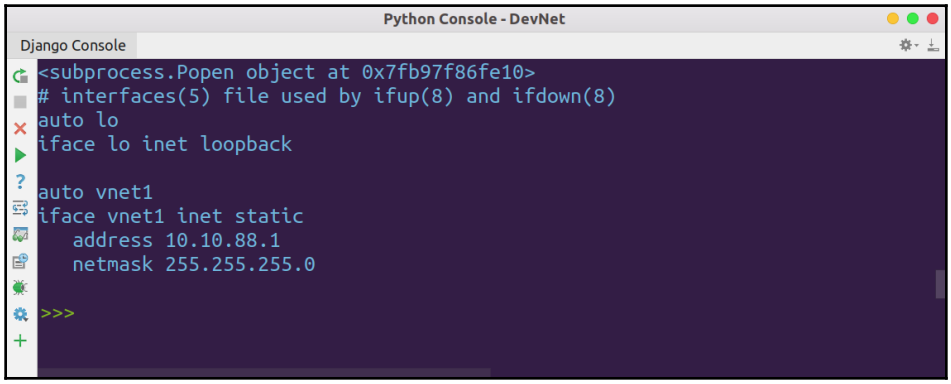
```
print(subprocess.Popen("sudo ifconfig enp60s0:0 10.10.10.2 netmask
255.255.255.0 up", shell=True))
```

You can think about `shell=True` as you spawn a shell process and pass the command with an argument to it. This could save you a few lines of code through using `split()`, in case you receive the command from an external system and want to run it directly.

The default shell used by `subprocess` is `/bin/sh`. If you're using other shells, like `tch` or `csh`, you can define them in the `executable` argument. Also notice running the command as a shell can be a security issue and allow *security injection*. A user who instructs your code to run the script can add `"; rm -rf /"` and cause terrible things to happen.

Also, you can change the directory to a specific one before running the command by using the `cwd` argument. This is useful when you need to list the contents of the directory before operating on it:

```
import subprocess
print(subprocess.Popen(["cat", "interfaces"], cwd="/etc/network"))
```



Ansible has a similar flag called `chdir:`. This argument will be used inside a playbook task to change a directory before the execution.

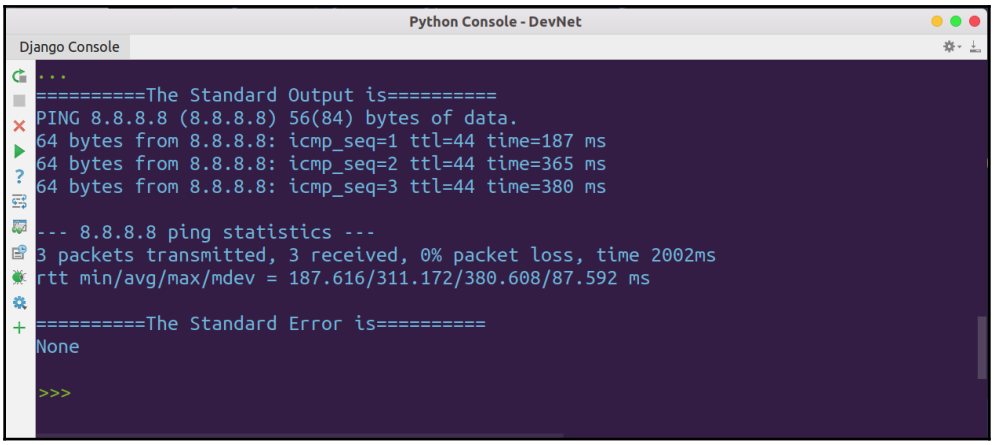
Reading stdin, stdout, and stderr

The spawned processes can communicate with the operating system in three channels:

1. Standard input (`stdin`)
2. Standard output (`stdout`)
3. Standard error (`stderr`)

In subprocess, `Popen()` can interact with the three channels and redirect each stream to an external file, or to a special value called `PIPE`. An additional method, called `communicate()`, is used to read from the `stdout` and write on the `stdin`.
The `communicate()` method can take input from the user and return both the standard output and the standard error, as shown in the following code snippet:

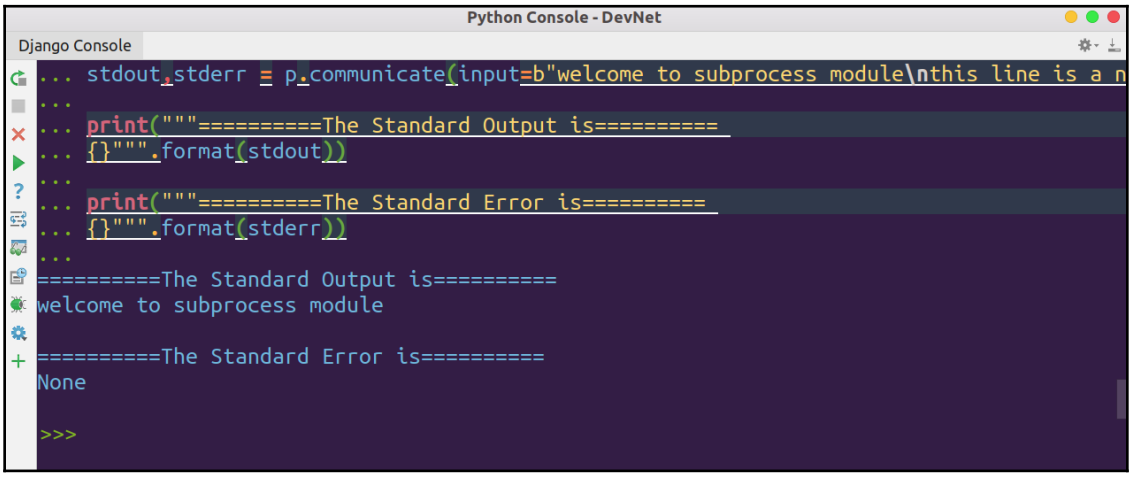
```
import subprocess
p = subprocess.Popen(["ping", "8.8.8.8", "-c", "3"], stdin=subprocess.PIPE,
stdout=subprocess.PIPE)
stdout, stderr = p.communicate()
print("""==========The Standard Output is==========
{}""".format(stdout))

print("""==========The Standard Error is==========
{}""".format(stderr))
```



Similarly, you can send data and write to the process using the input argument inside `communicate()`:

```
import subprocess
p = subprocess.Popen(["grep", "subprocess"], stdout=subprocess.PIPE,
stdin=subprocess.PIPE)
stdout,stderr = p.communicate(input=b"welcome to subprocess module\nthis
line is a new line and doesnot contain the require string")

print("""==========The Standard Output is==========
{}""".format(stdout))

print("""==========The Standard Error is==========
{}""".format(stderr))
```

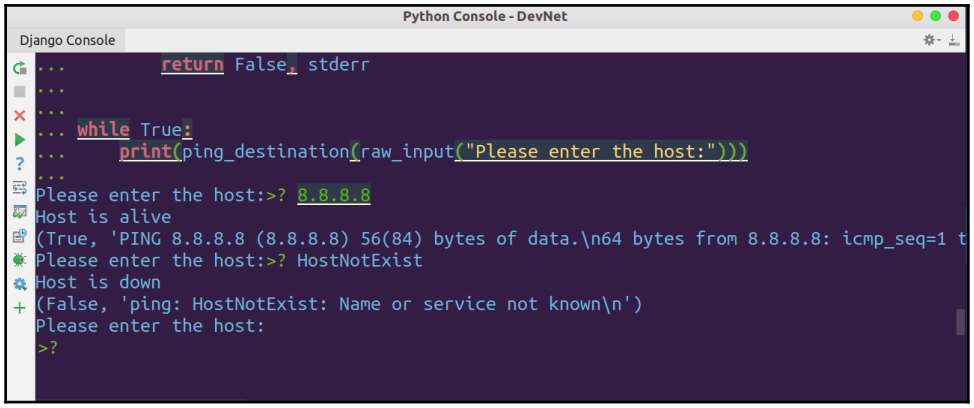
In the script, we used the `input` argument inside `communicate()`,which will send the data to the other child process, which will search for the subprocess keyword using the `grep` command. The returned output will be stored inside the `stdout` variable:



Another approach to validate the successful execution of the process is to use the return code. When the command has successfully executed without errors, the return code will be `0`; otherwise, it will be an integer value larger than `0`:

```
import subprocess

def ping_destination(ip):
    p = subprocess.Popen(['ping', '-c', '3'],
                         stdout=subprocess.PIPE,
                         stderr=subprocess.PIPE)
    stdout, stderr = p.communicate(input=ip)
    if p.returncode == 0:
        print("Host is alive")
        return True, stdout
    else:
        print("Host is down")
        return False, stderr

while True:
    print(ping_destination(raw_input("Please enter the host:")))
```

The script will ask the user to enter an IP address, and will then call the `ping_destination()` function, which will execute the `ping` command against the IP address. The result of the `ping` command (either `success` or `failed`) will return in the standard output, and the `communicate()` function will populate the return code with the result:



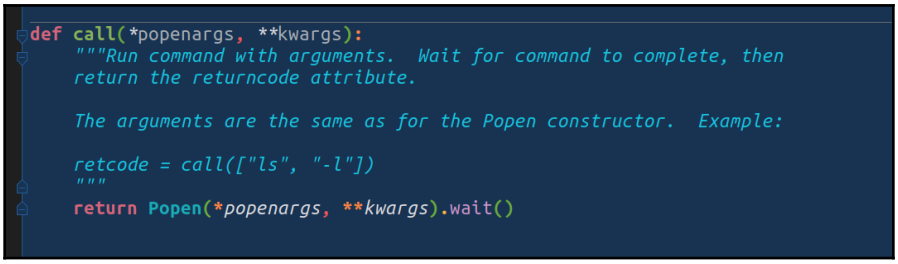
First, we tested the Google DNS IP address. The host is alive, and the command will be successfully executed with the return code =0. The function will return `True` and print `Host is alive`. Second, we tested with the `HostNotExist` string. The function will return `False` to the main program and print `Host is down`. Also, it will print the command standard output returned to subprocess which is (`Name or service not known`).

You can use `echo $?` to check the return code (sometimes called the exit code) of the previously executed command.

The subprocess call suite

The subprocess module provides another function that makes process spawning a safer operation than using `Popen()`. The subprocess `call()` function waits for the called command/program to finish reading the output. It supports the same arguments as the `Popen()` constructor, such as `shell`, `executable`, and `cwd`, but this time, your script will wait for the program to complete and populate the return code without the need to `communicate()`.

If you inspect the `call()` function, you will see that it's actually a wrapper around the `Popen()` class, but with a `wait()` function that waits until the end of the command before returning the output:



```
def call(*popenargs, **kwargs):
    """Run command with arguments.  Wait for command to complete, then
    return the returncode attribute.

    The arguments are the same as for the Popen constructor.  Example:

    retcode = call(["ls", "-l"])
    """
    return Popen(*popenargs, **kwargs).wait()
```

```
import subprocess
subprocess.call(["ifconfig", "docker0"], stdout=subprocess.PIPE,
stderr=None, shell=False)
```

If you want more protection for your code, you can use the `check_call()` function. It's the same as `call()`, but adds another check to the return code. If it is equal to `0` (meaning that the command has successfully executed), then the output will be returned. Otherwise, it will raise an exception with the returned exit code. This will allow you to handle the exception in your program flow:

```
import subprocess

try:
    result = subprocess.check_call(["ping", "HostNotExist", "-c", "3"])
except subprocess.CalledProcessError:
    print("Host is not found")
```

A downside of using the `call()` function is that you can't use `communicate()` to send the data to process, like we did with `Popen()`.

Summary

In this chapter, we learned how to run and spawn new processes in the system, and we learned about how these spawned processes communicate with the operating system. We also discussed the subprocess module and the `subprocess` call.

In the next chapter, we will see how to run and execute commands on remote hosts.

10
Running System Administration Tasks with Fabric

In the previous chapter, we used the `subprocess` module to run and spawn a system process inside the machine that hosted our Python script, and to return the output back to the Terminal. However, many automation tasks require access to remote servers to execute commands, which is not easy to do using a sub-process. This becomes a piece of cake with the use of another Python module: `Fabric`. The library makes connections to remote hosts and executes different tasks, such as uploading and downloading files, running commands with specific user IDs, and prompting users for input. The `Fabric` Python module is a robust tool for administrating dozens of Linux machines from a central point.

The following topics will be covered in this chapter:

- What is Fabric?
- Executing your first Fabric file
- Other useful Fabric features

Technical requirements

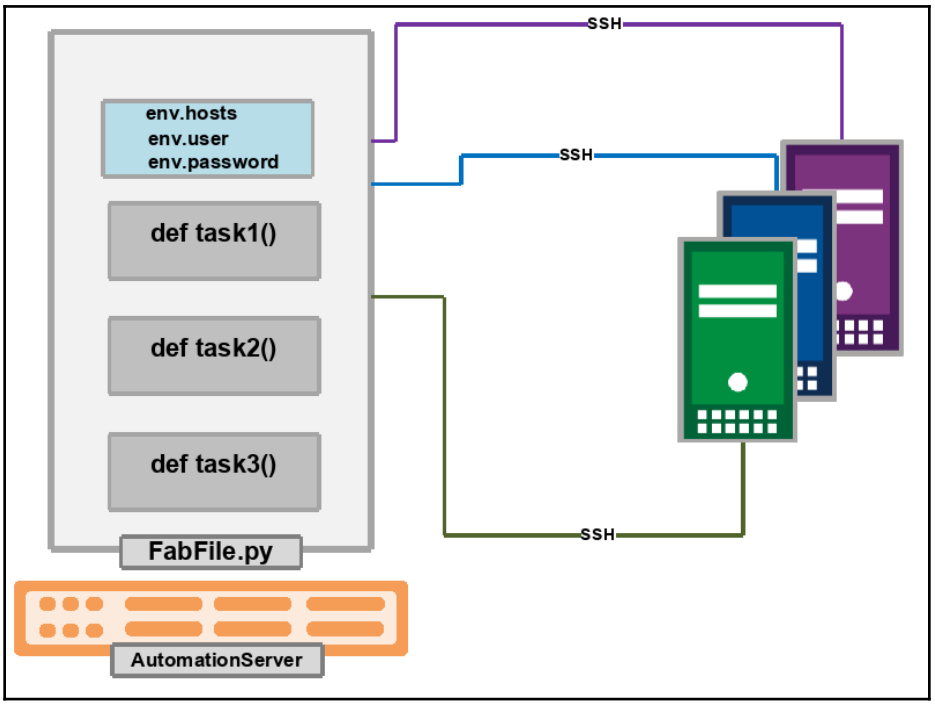
The following tools should be installed and available in your environment:

- Python 2.7.1x.
- PyCharm Community or Pro Edition.
- EVE-NG topology. Please refer to Chapter 8, *Preparing a Lab Environment*, for how to install and configure system servers.

You can find the full scripts developed in this chapter at the following GitHub URL: `https://github.com/TheNetworker/EnterpriseAutomation.git`.

What is Fabric?

Fabric (`http://www.fabfile.org/`) is a high-level Python library that is used to connect to remote servers (through the paramiko library) and execute predefined tasks on them. It runs a tool called **fab** on the machine that hosts the fabric module. This tool will look for a `fabfile.py` file, located in the same directory that you run the tool in.
The `fabfile.py` file contains your tasks, defined as a Python function that is called from the command line to start the execution on the servers. The Fabric tasks themselves are just normal Python functions, but they contain special methods that are used to execute commands on remote servers. Also, at the beginning of `fabfile.py`, you need to define some environmental variables, such as the remote hosts, username, password, and any other variables needed during execution:



Installation

Fabric requires Python 2.5 to 2.7. You can install Fabric and all of its dependencies using `pip`, or you can use a system package manager, such as `yum` or `apt`. In both cases, you will have the `fab` utility ready and executable from your operating system.

To install `fabric` using `pip`, run the following command on your automation server:

```
pip install fabric
```

```
[root@AutomationServer ~]#
[root@AutomationServer ~]# pip install fabric
Collecting fabric
  Downloading Fabric-1.14.0-py2-none-any.whl (92kB)
    100% |████████████████████████████████| 102kB 738kB/s
Collecting paramiko<3.0,>=1.10 (from fabric)
  Downloading paramiko-2.4.1-py2.py3-none-any.whl (194kB)
    100% |████████████████████████████████| 194kB 1.4MB/s
Collecting pyasn1>=0.1.7 (from paramiko<3.0,>=1.10->fabric)
  Downloading pyasn1-0.4.2-py2.py3-none-any.whl (71kB)
    100% |████████████████████████████████| 71kB 3.2MB/s
Collecting bcrypt>=3.1.3 (from paramiko<3.0,>=1.10->fabric)
  Downloading bcrypt-3.1.4-cp27-cp27mu-manylinux1_x86_64.whl (57kB)
    100% |████████████████████████████████| 61kB 3.3MB/s
Collecting cryptography>=1.5 (from paramiko<3.0,>=1.10->fabric)
  Downloading cryptography-2.2.2-cp27-cp27mu-manylinux1_x86_64.whl (2.2MB)
    100% |████████████████████████████████| 2.2MB 353kB/s
Collecting pynacl>=1.0.1 (from paramiko<3.0,>=1.10->fabric)
  Downloading PyNaCl-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl (696kB)
    100% |████████████████████████████████| 706kB 918kB/s
Requirement already satisfied (use --upgrade to upgrade): six>=1.4.1 in /usr/lib/python2.7/si
te-packages (from bcrypt>=3.1.3->paramiko<3.0,>=1.10->fabric)
Collecting cffi>=1.1 (from bcrypt>=3.1.3->paramiko<3.0,>=1.10->fabric)
  Downloading cffi-1.11.5-cp27-cp27mu-manylinux1_x86_64.whl (407kB)
    100% |████████████████████████████████| 409kB 1.4MB/s
Collecting enum34; python_version < "3" (from cryptography>=1.5->paramiko<3.0,>=1.10->fabric)
  Downloading enum34-1.1.6-py2-none-any.whl
```

Notice that Fabric requires `paramiko`, which is a popular Python library that is used for establishing SSH connections.

You can validate the Fabric installation with two steps. First, make sure that you have the `fab` command available in your system:

```
[root@AutomationServer ~]# which fab
/usr/bin/fab
```

The second step for verification is to open Python and try to import the `fabric` library. If there's no error thrown, then Fabric has successfully installed:

```
[root@AutomationServer ~]# python
Python 2.7.5 (default, Aug  4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from fabric.api import *
>>>
```

Fabric operations

There are many operations available in the `fabric` tool. These operations act as a functions inside the tasks in fabfile (there will be more about tasks later), but the following is a summary of the most important operations inside the `fabric` library.

Using run operation

The syntax for the `run` operation in Fabric is as follows:

```
run(command, shell=True, pty=True, combine_stderr=True, quiet=False,
warn_only=False, stdout=None, stderr=None)
```

This will execute the command on a remote host, while the `shell` argument controls whether a shell (such as `/bin/sh`) should be created before execution (the same parameter also exists in the sub-process).

After the command execution, Fabric will populate `.succeeded` or `.failed`, depending on the command output. You can check whether the command succeeded or failed by calling the following:

```
def run_ops():
    output = run("hostname")
```

Using get operation

The syntax for the Fabric `get` operation is as follows:

```
get(remote_path, local_path)
```

This will download the files from the remote host to the machine running the `fabfile`, using either `rsync` or `scp` . This is commonly used when you need to gather log files to the server, for example:

```
def get_ops():
    try:
        get("/var/log/messages","/root/")
    except:
        pass
```

Using put operation

The syntax for the Fabric `put` operation is as follows:

```
put(local_path, remote_path, use_sudo=False, mirror_local_mode=False,
mode=None)
```

This operation will upload the file from the machine running the `fabfile` (local) to the remote host. Using `use_sudo` will solve the permissions issue when you upload to the root directory. Also, you can keep the current file permissions on both the local and remote server, or you can set new permissions:

```
def put_ops():
    try:
        put("/root/VeryImportantFile.txt","/root/")
    except:
        pass
```

Using sudo operation

The syntax for the Fabric `sudo` operation is as follows:

```
sudo(command, shell=True, pty=True, combine_stderr=True, user=None,
quiet=False, warn_only=False, stdout=None, stderr=None, group=None)
```

This operation can be considered another wrapper around the `run()` command. However, the `sudo` operation will run the command with the root username by default regardless of the username used to execute the `fabfile`. Also it contains a user argument which could be used to run the command with a different username. Also, the `user` argument executes the command with a specific UID, while the `group` argument defines the GID:

```
def sudo_ops():
    sudo("whoami") #it should print the root even if you use another
account
```

Using prompt operation

The syntax for the Fabric `prompt` operation is as follows:

```
prompt(text, key=None, default='', validate=None)
```

The user can provide a specific value for the task by using the `prompt` operation, and the input will be stored inside of a variable and used by tasks. Please note that you will be prompted for each host inside of the `fabfile`:

```
def prompt_ops():
    prompt("please supply release name", default="7.4.1708")
```

Using reboot operation

The syntax for the Fabric `reboot` operation is as follows:

```
reboot(wait=120)
```

This is a simple operation that reboots the host by default. Fabric will wait for 120 seconds before attempting to reconnect, but you can change this value to another one by using the `wait` argument:

```
def reboot_ops():
    reboot(wait=60, use_sudo=True)
```

For a full list of other supported operations, please check `http://docs.fabfile.org/en/1.14/api/core/operations.html`. You can also check them directly from PyCharm, by looking at all of the autocomplete functions that pop up when you type *Ctrl + spacebar*. From `fabric.operations` import *<ctrl+space>* under `fabric.operations`:

Executing your first Fabric file

We now know how the operation works, so we will put it inside `fabfile` and create a full automation script that can work with remote machines. The first step for `fabfile` is to import the required classes. Most of them are located in `fabric.api`, so we will globally import all of them to our Python script:

```
from fabric.api import *
```

The next part is to define the remote machine IP addresses, usernames, and passwords. In the case of our environment, we have two machines (besides the automation server) that run Ubuntu 16.04 and CentOS 7.4, respectively, with the following details:

Machine Type	**IP Address**	**Username**	**Password**
Ubuntu 16.04	`10.10.10.140`	`root`	`access123`
CentOS 7.4	`10.10.10.193`	`root`	`access123`

We will include them inside the Python script, as shown in the following snippet:

```
env.hosts = [
    '10.10.10.140',  # ubuntu machine
    '10.10.10.193',  # CentOS machine
]

env.user = "root"
env.password = "access123"
```

Notice that we use the variable called `env`, which is inherited from the `_AttributeDict` class. Inside of this variable, we can set the username and password from the SSH connection. You can also use the SSH keys stored in your `.ssh` directory by setting `env.use_ssh_config=True`; Fabric will use the keys to authenticate the connection.

The last step is to define your tasks as a Python function. Tasks can use the preceding operations to execute commands.

The following is the full script:

```
from fabric.api import *

env.hosts = [
    '10.10.10.140',  # ubuntu machine
    '10.10.10.193',  # CentOS machine
]

env.user = "root"
env.password = "access123"

def detect_host_type():
    output = run("uname -s")
    if output.failed:
        print("something wrong happen, please check the logs")
    elif output.succeeded:
        print("command executed successfully")
```

```
def list_all_files_in_directory():
    directory = prompt("please enter full path to the directory to list",
default="/root")
    sudo("cd {0} ; ls -htlr".format(directory))


def main_tasks():
    detect_host_type()
    list_all_files_in_directory()
```

In the preceding example, the following applies:

- We defined two tasks. The first one will execute the `uname -s` command and return the output, then verify whether the command executed successfully or not. The task uses the `run()` operation to accomplish it.
- The second task will use two operations: `prompt()` and `sudo()`. The first operation will ask the user to enter the full path to the directory, while the second operation will list all of the content in the directory.
- The final task, `main_tasks()`, will actually group the preceding two methods into one task, so that we can call it from the command line.

In order to run the script, we will upload the file to the automation server and use the `fab` utility to run it:

```
fab -f </full/path/to/fabfile>.py <task_name>
```

The `-f` switch in the previous command is not mandatory if your filename is `fabfile.py`. If it is not, you will need to provide the name to the `fab` utility. Also, `fabfile` should be in the current directory; otherwise, you will need to provide the full path.

Now we will run the `fabfile` by executing the following command:

```
fab -f fabfile_first.py main_tasks
```

The first task will be executed, and will return the output to the Terminal:

```
[10.10.10.140] Executing task 'main_tasks'
[10.10.10.140] run: uname -s
[10.10.10.140] out: Linux
[10.10.10.140] out:

command executed successfully
```

Now, we will enter `/var/log/` to list the contents:

```
please enter full path to the directory to list [/root] /var/log/
[10.10.10.140] sudo: cd /var/log/ ; ls -htlr
[10.10.10.140] out: total 1.7M
[10.10.10.140] out: drwxr-xr-x 2 root   root 4.0K Dec  7 23:54 lxd
[10.10.10.140] out: drwxr-xr-x 2 root   root 4.0K Dec 11 15:47 sysstat
[10.10.10.140] out: drwxr-xr-x 2 root   root 4.0K Feb 22 18:24 dist-upgrade
[10.10.10.140] out: -rw------- 1 root   utmp    0 Feb 28 20:23 btmp
[10.10.10.140] out: -rw-r----- 1 root   adm    31 Feb 28 20:24 dmesg
[10.10.10.140] out: -rw-r--r-- 1 root   root  57K Feb 28 20:24
bootstrap.log
[10.10.10.140] out: drwxr-xr-x 2 root   root 4.0K Apr  4 08:00 fsck
[10.10.10.140] out: drwxr-xr-x 2 root   root 4.0K Apr  4 08:01 apt
[10.10.10.140] out: -rw-r--r-- 1 root   root  32K Apr  4 08:09 faillog
[10.10.10.140] out: drwxr-xr-x 3 root   root 4.0K Apr  4 08:09 installer


command executed successfully
```

The same applies if you need to list the configuration files under the `network-scripts` directory in the CentOS machine:

```
please enter full path to the directory to list [/root]
/etc/sysconfig/network-scripts/
[10.10.10.193] sudo: cd /etc/sysconfig/network-scripts/ ; ls -htlr
[10.10.10.193] out: total 232K
[10.10.10.193] out: -rwxr-xr-x. 1 root root 1.9K Apr 15  2016 ifup-TeamPort
[10.10.10.193] out: -rwxr-xr-x. 1 root root 1.8K Apr 15  2016 ifup-Team
[10.10.10.193] out: -rwxr-xr-x. 1 root root 1.6K Apr 15  2016 ifdown-
TeamPort
[10.10.10.193] out: -rw-r--r--. 1 root root  31K May  3  2017 network-
functions-ipv6
[10.10.10.193] out: -rw-r--r--. 1 root root  19K May  3  2017 network-
functions
[10.10.10.193] out: -rwxr-xr-x. 1 root root 5.3K May  3  2017 init.ipv6-
global
[10.10.10.193] out: -rwxr-xr-x. 1 root root 1.8K May  3  2017 ifup-wireless
[10.10.10.193] out: -rwxr-xr-x. 1 root root 2.7K May  3  2017 ifup-tunnel
[10.10.10.193] out: -rwxr-xr-x. 1 root root 3.3K May  3  2017 ifup-sit
[10.10.10.193] out: -rwxr-xr-x. 1 root root 2.0K May  3  2017 ifup-routes
[10.10.10.193] out: -rwxr-xr-x. 1 root root 4.1K May  3  2017 ifup-ppp
[10.10.10.193] out: -rwxr-xr-x. 1 root root 3.4K May  3  2017 ifup-post
[10.10.10.193] out: -rwxr-xr-x. 1 root root 1.1K May  3  2017 ifup-plusb

<output omitted for brevity>
```

Finally, Fabric will disconnect from the two machines:

```
[10.10.10.193] out:


Done.
Disconnecting from 10.10.10.140... done.
Disconnecting from 10.10.10.193... done.
```

More about the fab tool

The `fab` tool itself supports many operations. It can be used to list the different tasks inside `fabfile`. It can also set the `fab` environment during execution. For example, you can define the host that will run the commands on it by using the `-H` or `--hosts` switches, without the need to specify it inside `fabfile`. This actually sets the `env.hosts` variable inside `fabfile` during execution:

```
fab -H srv1,srv2
```

On the other hand, you can define the command that you want to run by using the `fab` tool. This is something like Ansible ad hoc mode (we will discuss this in detail in `Chapter 13`, *Ansible for System Administration*):

```
fab -H srv1,srv2 -- ifconfig -a
```

If you don't want to store the password in clear text inside of the `fabfile` script, then you have two options. The first one is to use the SSH identity file (`private-key`) with the `-i` option, which loads the file during connection.

The other option is to force Fabric to prompt you for the session password before connecting to the remote machine by using the `-I` option.

Note that this option will overwrite the `env.password` parameter, if specified inside `fabfile`.

The `-D` switch will disable the known hosts and force Fabric not to load the `known_hosts` file from the `.ssh` directory. You can make Fabric reject connections to the hosts not defined in the `known_hosts` file with the `-r` or `--reject-unknown-hosts` options.

Also, you can list all of the supported tasks inside of the fabfile by using `-l` or `--list`, providing the fabfile name to the `fab` tool. For example, applying that to the previous script will generate the following output:

```
# fab -f fabfile_first.py -l
Available commands:

    detect_host_type
    list_all_files_in_directory
    main_tasks
```

You can see all of the available options and arguments for the `fab` command line with the `-h` switch, or at `http://docs.fabfile.org/en/1.14/usage/fab.html`.

Discover system health using Fabric

In this use case, we will utilize Fabric to develop a script that executes multiple commands on remote machines. The goal of the script is to gather two types of output: the `discovery` command and the `health` command. The `discovery` command gathers the uptime, hostname, kernel release, and both private and public IP addresses, while the `health` command gathers the used memory, CPU utilization, number of spawned processes, and disk usage. We will design `fabfile` so that we can scale our script and add more commands to it:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from fabric.api import *
from fabric.context_managers import *
from pprint import pprint

env.hosts = [
    '10.10.10.140',  # Ubuntu Machine
    '10.10.10.193',  # CentOS Machine
]

env.user = "root"
env.password = "access123"


def get_system_health():
```

```
    discovery_commands = {
        "uptime": "uptime | awk '{print $3,$4}'",
        "hostname": "hostname",
        "kernel_release": "uname -r",
        "architecture": "uname -m",
        "internal_ip": "hostname -I",
        "external_ip": "curl -s ipecho.net/plain;echo",


    }
    health_commands = {
        "used_memory": "free  | awk '{print $3}' | grep -v free | head -
n1",
        "free_memory": "free  | awk '{print $4}' | grep -v shared | head -
n1",
        "cpu_usr_percentage": "mpstat | grep -A 1 '%usr' | tail -n1 | awk
'{print $4}'",
        "number_of_process": "ps -A --no-headers | wc -l",
        "logged_users": "who",
        "top_load_average": "top -n 1 -b | grep 'load average:' | awk
'{print $10 $11 $12}'",
        "disk_usage": "df -h| egrep 'Filesystem|/dev/sda*|nvme*'"

    }

    tasks = [discovery_commands,health_commands]

    for task in tasks:
        for operation,command in task.iteritems():
print("============================{0}============================".forma
t(operation))
            output = run(command)
```

Notice that we created two dictionaries: `discover_commands` and `health_commands`. Each one of them contains the Linux commands as a key-value pair. The key represents the operation, while the value represents the actual Linux command. Then, we created a `tasks` list to group both dictionaries.

Finally, we created a nested `for` loop. The outer loop is used to iterate over the list items. The inner `for` loop is to iterate over the key-value pairs. Use the Fabric `run()` operation to send the command to the remote hosts:

```
# fab -f fabfile_discoveryAndHealth.py get_system_health
[10.10.10.140] Executing task 'get_system_health'
============================uptime============================
[10.10.10.140] run: uptime | awk '{print $3,$4}'
[10.10.10.140] out: 3:26, 2
```

```
[10.10.10.140] out:

==============================kernel_release==============================
[10.10.10.140] run: uname -r
[10.10.10.140] out: 4.4.0-116-generic
[10.10.10.140] out:

==============================external_ip==============================
[10.10.10.140] run: curl -s ipecho.net/plain;echo
[10.10.10.140] out: <Author_Masked_The_Output_For_Privacy>
[10.10.10.140] out:

==============================hostname==============================
[10.10.10.140] run: hostname
[10.10.10.140] out: ubuntu-machine
[10.10.10.140] out:

==============================internal_ip==============================
[10.10.10.140] run: hostname -I
[10.10.10.140] out: 10.10.10.140
[10.10.10.140] out:

==============================architecture==============================
[10.10.10.140] run: uname -m
[10.10.10.140] out: x86_64
[10.10.10.140] out:

==============================disk_usage==============================
[10.10.10.140] run: df -h| egrep 'Filesystem|/dev/sda*|nvme*'
[10.10.10.140] out: Filesystem                                Size  Used Avail
Use% Mounted on
[10.10.10.140] out: /dev/sda1                                 472M   58M  390M
13% /boot
[10.10.10.140] out:

==============================used_memory==============================
[10.10.10.140] run: free  | awk '{print $3}' | grep -v free | head -n1
[10.10.10.140] out: 75416
[10.10.10.140] out:

==============================logged_users==============================
[10.10.10.140] run: who
[10.10.10.140] out: root     pts/0        2018-04-08 23:36 (10.10.10.130)
[10.10.10.140] out: root     pts/1        2018-04-08 21:23 (10.10.10.1)
[10.10.10.140] out:

==============================top_load_average==============================
[10.10.10.140] run: top -n 1 -b | grep 'load average:' | awk '{print $10
```

```
$11 $12}'
[10.10.10.140] out: 0.16,0.03,0.01
[10.10.10.140] out:

==============================cpu_usr_percentage==============================
=
[10.10.10.140] run: mpstat | grep -A 1 '%usr' | tail -n1 | awk '{print $4}'
[10.10.10.140] out: 0.02
[10.10.10.140] out:

==============================number_of_process==============================
[10.10.10.140] run: ps -A --no-headers | wc -l
[10.10.10.140] out: 131
[10.10.10.140] out:

==============================free_memory==============================
[10.10.10.140] run: free  | awk '{print $4}' | grep -v shared | head -n1
[10.10.10.140] out: 5869268
[10.10.10.140] out:
```

The same task (`get_system_health`) will also be executed on the second server, and will return the output to the Terminal:

```
[10.10.10.193] Executing task 'get_system_health'
==============================uptime==============================
[10.10.10.193] run: uptime | awk '{print $3,$4}'
[10.10.10.193] out: 3:26, 2
[10.10.10.193] out:

==============================kernel_release==============================
[10.10.10.193] run: uname -r
[10.10.10.193] out: 3.10.0-693.el7.x86_64
[10.10.10.193] out:

==============================external_ip==============================
[10.10.10.193] run: curl -s ipecho.net/plain;echo
[10.10.10.193] out: <Author_Masked_The_Output_For_Privacy>
[10.10.10.193] out:

==============================hostname==============================
[10.10.10.193] run: hostname
[10.10.10.193] out: controller329
[10.10.10.193] out:

==============================internal_ip==============================
[10.10.10.193] run: hostname -I
```

```
[10.10.10.193] out: 10.10.10.193
[10.10.10.193] out:

==============================architecture==============================
[10.10.10.193] run: uname -m
[10.10.10.193] out: x86_64
[10.10.10.193] out:

==============================disk_usage==============================
[10.10.10.193] run: df -h| egrep 'Filesystem|/dev/sda*|nvme*'
[10.10.10.193] out: Filesystem               Size  Used Avail Use% Mounted
on
[10.10.10.193] out: /dev/sda1                488M   93M  360M  21% /boot
[10.10.10.193] out:

==============================used_memory==============================
[10.10.10.193] run: free  | awk '{print $3}' | grep -v free | head -n1
[10.10.10.193] out: 287048
[10.10.10.193] out:

==============================logged_users==============================
[10.10.10.193] run: who
[10.10.10.193] out: root     pts/0        2018-04-08 23:36 (10.10.10.130)
[10.10.10.193] out: root     pts/1        2018-04-08 21:23 (10.10.10.1)
[10.10.10.193] out:

==============================top_load_average==============================
[10.10.10.193] run: top -n 1 -b | grep 'load average:' | awk '{print $10
$11 $12}'
[10.10.10.193] out: 0.00,0.01,0.02
[10.10.10.193] out:

==============================cpu_usr_percentage==============================
=
[10.10.10.193] run: mpstat | grep -A 1 '%usr' | tail -n1 | awk '{print $4}'
[10.10.10.193] out: 0.00
[10.10.10.193] out:

==============================number_of_process==============================
[10.10.10.193] run: ps -A --no-headers | wc -l
[10.10.10.193] out: 190
[10.10.10.193] out:

==============================free_memory==============================
[10.10.10.193] run: free  | awk '{print $4}' | grep -v shared | head -n1
[10.10.10.193] out: 32524912
[10.10.10.193] out:
```

Finally, the `fabric` module will terminate the established SSH session and disconnect from the two machines after executing all of the tasks:

```
Disconnecting from 10.10.10.140... done.
Disconnecting from 10.10.10.193... done.
```

Note that we could redesign the previous script and make the `discovery_commands` and `health_commands` a Fabric task, then include them within `get_system_health()`. When we execute the `fab` command, we will call `get_system_health()`, which will execute the other two functions; we will get the same output as before. The following is a modified sample script:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from fabric.api import *
from fabric.context_managers import *
from pprint import pprint

env.hosts = [
    '10.10.10.140',  # Ubuntu Machine
    '10.10.10.193',  # CentOS Machine
]

env.user = "root"
env.password = "access123"


def discovery_commands():
    discovery_commands = {
        "uptime": "uptime | awk '{print $3,$4}'",
        "hostname": "hostname",
        "kernel_release": "uname -r",
        "architecture": "uname -m",
        "internal_ip": "hostname -I",
        "external_ip": "curl -s ipecho.net/plain;echo",


    }
    for operation, command in discovery_commands.iteritems():
print("============================{0}============================".forma
t(operation))
        output = run(command)

def health_commands():
    health_commands = {
```

```
        "used_memory": "free  | awk '{print $3}' | grep -v free | head -
n1",
        "free_memory": "free  | awk '{print $4}' | grep -v shared | head -
n1",
        "cpu_usr_percentage": "mpstat | grep -A 1 '%usr' | tail -n1 | awk
'{print $4}'",
        "number_of_process": "ps -A --no-headers | wc -l",
        "logged_users": "who",
        "top_load_average": "top -n 1 -b | grep 'load average:' | awk
'{print $10 $11 $12}'",
        "disk_usage": "df -h| egrep 'Filesystem|/dev/sda*|nvme*'"

    }
    for operation, command in health_commands.iteritems():
print("============================{0}============================".forma
t(operation))
        output = run(command)

def get_system_health():
    discovery_commands()
    health_commands()
```

Other useful features in Fabric

Fabric has other useful features, such as roles and context managers.

Fabric roles

Fabric can define roles for hosts, and run only the tasks to role members. For example, we might have a bunch of database servers on which we need to validate whether the MySql service is up, and other web servers on which we need to validate whether the Apache service is up. We can group these hosts into roles, and execute functions based on those roles:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from fabric.api import *

env.hosts = [
    '10.10.10.140',  # ubuntu machine
    '10.10.10.193',  # CentOS machine
```

```
    '10.10.10.130',
]

env.roledefs = {
    'webapps': ['10.10.10.140','10.10.10.193'],
    'databases': ['10.10.10.130'],
}

env.user = "root"
env.password = "access123"

@roles('databases')
def validate_mysql():
    output = run("systemctl status mariadb")


@roles('webapps')
def validate_apache():
    output = run("systemctl status httpd")
```

In the preceding example, we used the Fabric decorator `roles` (imported from `fabric.api`) when setting `env.roledef`. Then, we will assign either webapp or databases roles to each server (think of the role assignment as tagging a server). This will give us flexibility to execute the `validate_mysql` function on servers with database role only:

```
# fab -f fabfile_roles.py validate_mysql:roles=databases
[10.10.10.130] Executing task 'validate_mysql'
[10.10.10.130] run: systemctl status mariadb
[10.10.10.130] out: ● mariadb.service - MariaDB database server
[10.10.10.130] out:    Loaded: loaded
(/usr/lib/systemd/system/mariadb.service; enabled; vendor preset: disabled)
[10.10.10.130] out:    Active: active (running) since Sat 2018-04-07
19:47:35 EET; 1 day 2h ago
<output omitted>
```

Fabric context managers

In our first Fabric script, `fabfile_first.py`, we have a task that prompts the user for the directory, then switches to it and prints its content. This is done by using `;`, which appends two Linux commands together. However, running the same won't always work on other operating systems. That's where the Fabric context manager comes into the picture.

The context manager maintains the directory state when executing commands. It usually runs with Python by using `with-statement`, and, inside the block, you can write any of the previous Fabric operations. Let's look at an example to explain the idea:

```
from fabric.api import *
from fabric.context_managers import *

env.hosts = [
    '10.10.10.140',  # ubuntu machine
    '10.10.10.193',  # CentOS machine
]

env.user = "root"
env.password = "access123"

def list_directory():
    with cd("/var/log"):
        run("ls")
```

In the preceding example, first, we globally imported everything inside `fabric.context_managers`; then, we used the `cd` context manager to switch to the specific directory. We used the Fabric `run()` operation to execute the `ls` on that directory. This is the same as writing `cd /var/log ; ls` on the SSH session, but it provides a more Pythonic way to develop your code.

The `with` statement can be nested. For example, we can rewrite the preceding code with the following:

```
def list_directory_nested():
    with cd("/var/"):
        with cd("log"):
            run("ls")
```

Another useful context manager is the **local change directory** (**LCD**). This is the same as the `cd` context manager in the previous example, but it works on the local machine that runs `fabfile`. We can use it to change the context to a specific directory (for example, to upload or download a file to/from the remote machine, then change back to the execution directory automatically):

```
def uploading_file():
    with lcd("/root/"):
        put("VeryImportantFile.txt")
```

The `prefix` context manager will accept a command as input and execute it before any other commands, inside the `with` block. For example, you can source a file or a Python virtual `env` wrapper script before running each command to set up your virtual environment:

```
def prefixing_commands():
    with prefix("source ~/env/bin/activate"):
        sudo('pip install wheel')
        sudo("pip install -r requirements.txt")
        sudo("python manage.py migrate")
```

This is actually equivalent to writing the following command in the Linux shell:

```
source ~/env/bin/activate && pip install wheel
source ~/env/bin/activate && pip install -r requirements.txt
source ~/env/bin/activate && python manage.py migrate
```

The final context manager is `shell_env(new_path, behavior='append')`, which can alter the shell environmental variables for wrapped commands; so, any calls inside of that block will take the modified path into consideration:

```
def change_shell_env():
    with shell_env(test1='val1', test2='val2', test3='val3'):
        run("echo $test1") #This command run on remote host
        run("echo $test2")
        run("echo $test3")
        local("echo $test1") #This command run on local host
```

Note that after the operation is done, Fabric will restore the old environments back to the original one.

Summary

Fabric is a fantastic and powerful tool that automates tasks, usually in remote machines. It integrates well with Python scripts, providing easy access to the SSH suite. You can develop many fab files for different tasks and integrate them together to create an automation workflow that includes deploying, restarting, and stopping servers or processes.

In the next chapter, we will learn about collecting data and generating recurring reports for system monitoring.

11
Generating System Reports and System Monitoring

Collecting data and generating recurring system reports are essential tasks for any system administrator, and automating these tasks can help us to discover issues early on, in order to provide solutions for them. In this chapter, we will see some proven methods for automating data collection from servers and generating that data into formal reports. We will learn how to manage new and existing users, using Python and Ansible. Also, we will dive into log analysis and monitoring the system **Key Performance Indicators** (**KPIs**). You can schedule the monitoring scripts to run on a regular basis.

The following topics will be covered in this chapter:

- Collecting data from Linux
- Managing users in Ansible

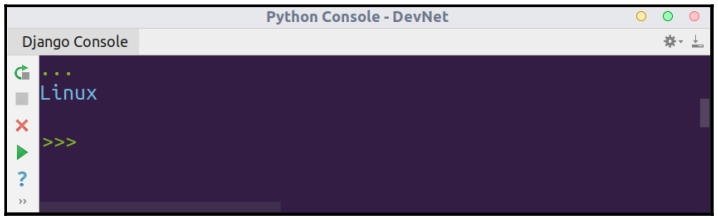
Collecting data from Linux

Native Linux commands provide useful data about the current system status and health. However, each one of those Linux commands and utilities are focused on getting data from only one aspect of the system. We need to leverage Python modules to get those details back to the administrator and generate useful system reports.

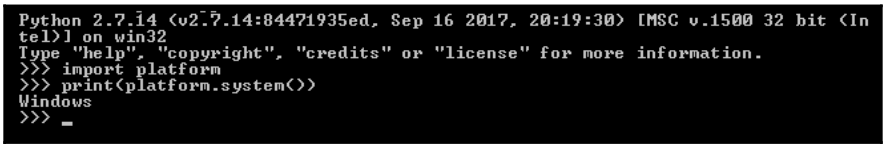
We will divide the reports into two parts. The first one is getting general information about the system by using the `platform` module, while the second part is exploring the hardware resources in terms of the CPU and memory.

We will start by leveraging the `platform` module, which is a built-in library inside of Python. The `platform` module contains many methods that can be used to get details about the system that Python operates on:

```
import platform
system = platform.system()
print(system)
```

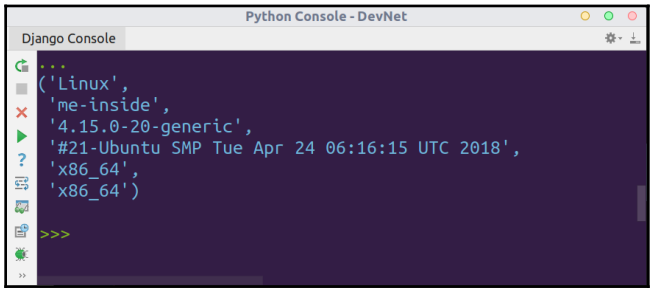


Running the same script on a Windows machine will result in different outputs, reflecting the current system. So, when we run it on a Windows PC, we will get `Windows` as the output from the script:



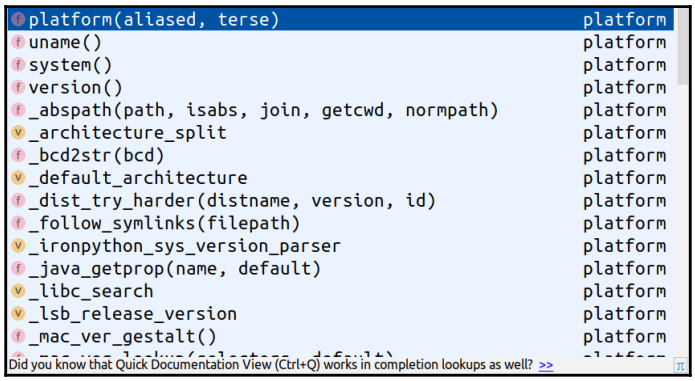
Another useful function is `uname()`, which does the same job as the Linux command (`uname -a`): retrieving the machine's hostname, architecture, and kernel, but in a structured format, so that you can match any value by referring to its index:

```
import platform
from pprint import pprint
uname = platform.uname()
pprint(uname)
```



The first value is the system type, which we get using the `system()` method, and the second value is the hostname of the current machine.

You can explore and list all of the available functions inside the `platform` module by using autocomplete in PyCharm; you can check the documentation for each function by pressing *CTRL* + *Q*:



The second part of designing our script is using the information made available by the Linux files to explore the hardware configuration in the Linux machine. Remember that the CPU, memory, and network information could be accessible from under `/proc/`; we will read this information and access it using standard `open()` function in Python. You can get more information about the available resources by reading and exploring `/proc/`.

Script:

This is the first step for importing the `platform` module. It's needed only for this task:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import platform
```

This snippet contains the functions used in this exercise; we will design two functions - `check_feature()` and `get_value_from_string()`:

```
def check_feature(feature,string):
    if feature in string.lower():
        return True
    else:
        return False
```

```
def get_value_from_string(key,string):
    value = "NONE"
    for line in string.split("\n"):
        if key in line:
            value = line.split(":")[1].strip()
    return value
```

Finally, the following is the main body of the Python script, which contains the Python logic to get the required information:

```
cpu_features = []
with open('/proc/cpuinfo') as cpus:
    cpu_data = cpus.read()
    num_of_cpus = cpu_data.count("processor")
    cpu_features.append("Number of Processors: {0}".format(num_of_cpus))
    one_processor_data = cpu_data.split("processor")[1]
    print one_processor_data
    if check_feature("vmx",one_processor_data):
        cpu_features.append("CPU Virtualization: enabled")
    if check_feature("cpu_meltdown",one_processor_data):
        cpu_features.append("Known Bugs: CPU Metldown ")
    model_name = get_value_from_string("model name ",one_processor_data)
    cpu_features.append("Model Name: {0}".format(model_name))

    cpu_mhz = get_value_from_string("cpu MHz",one_processor_data)
    cpu_features.append("CPU MHz: {0}".format((cpu_mhz)))

memory_features = []
with open('/proc/meminfo') as memory:
    memory_data = memory.read()
    total_memory = get_value_from_string("MemTotal",memory_data).replace("
kB","")
    free_memory = get_value_from_string("MemFree",memory_data).replace("
kB","")
    swap_memory = get_value_from_string("SwapTotal",memory_data).replace("
kB","")
    total_memory_in_gb = "Total Memory in GB:
{0}".format(int(total_memory)/1024)
    free_memory_in_gb = "Free Memory in GB:
{0}".format(int(free_memory)/1024)
    swap_memory_in_gb = "SWAP Memory in GB:
{0}".format(int(swap_memory)/1024)
    memory_features =
[total_memory_in_gb,free_memory_in_gb,swap_memory_in_gb]
```

This part is used to print the information obtained from the previous section:

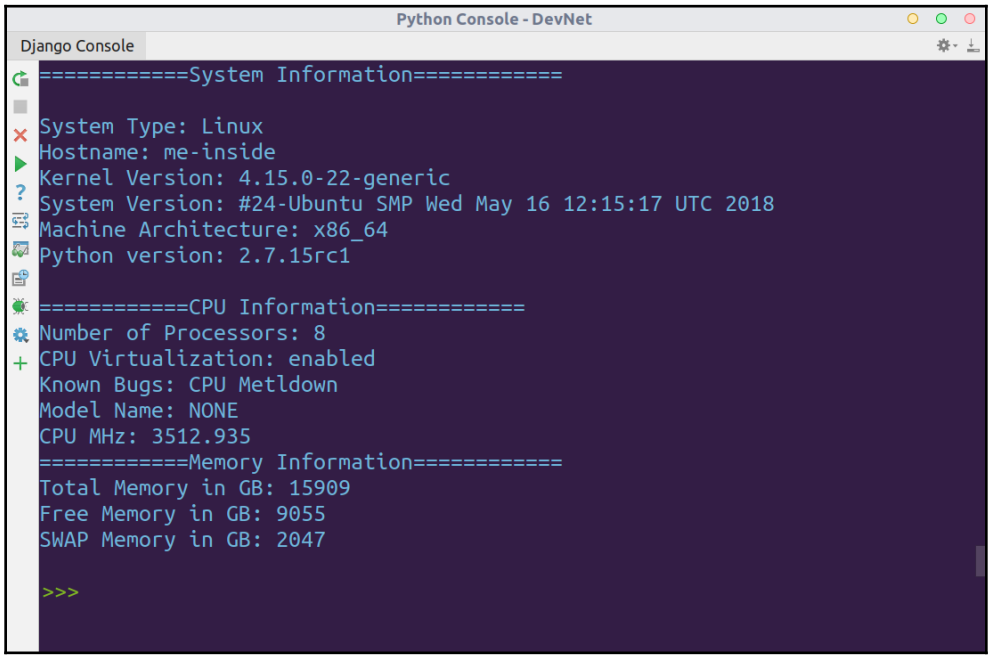
```
print("============System Information============")

print("""
System Type: {0}
Hostname: {1}
Kernel Version: {2}
System Version: {3}
Machine Architecture: {4}
Python version: {5}
""".format(platform.system(),
           platform.uname()[1],
           platform.uname()[2],
           platform.version(),
           platform.machine(),
           platform.python_version()))


print("============CPU Information============")
print("\n".join(cpu_features))


print("============Memory Information============")
print("\n".join(memory_features))
```

In the preceding example, the following steps were performed:

1. First, we opened `/proc/cpuinfo` and read its contents, then stored the result in `cpu_data`.
2. The number of processors inside the file could be found by counting the keyword `processor` using the `count()` String function.
3. Then, we needed to get the options and features available for each processor. For that, we got only one processor entry (since they're usually identical to each other) and passed it the `check_feature()` function. This method accepts the feature that we want to search in one argument, and the other is the processor data, which will return `True` if the feature is available in the processor data.
4. The processor data is available in key-value pairs. So, we designed the `get_value_from_string()` method, which accepts the key name and will search for its corresponding value by iterating over the processor data; then, we will split on the `:` delimiter for every returned key value pair to get the value only.

5. All of these values are added to the `cpu_feature` list using the `append()` method.
6. We then repeated the same operation with the memory information to get the total, free, and swap memory.
7. Next, we used the platform's built-in methods, such as `system()`, `uname()`, and `python_version()`, to get information about the system.
8. At the end, we printed the report that contains the preceding information.

The script output can be seen in the following screenshot:



Another way to represent the generated data is to leverage the `matplotlib` library that we used in `Chapter 5`, *Extracting Useful Data for Network Devices*, to visualize data over time.

Sending generated data through email

The report generated in the previous section provides a good overview of the resources currently on the system. However, we can tweak the script and extend its functionality to send us an email with all of the details. This is very useful for a **Network Operation Center** (**NoC**) team, which can receive emails from a monitored system based on specific incidents (HDD failure, high CPU, or dropped packets). Python has a built-in library called `smtplib`, where it leverages the **Simple Mail Transfer Protocol** (**SMTP**) that is responsible for sending and receiving emails from mail servers.

This requires that you have local email servers on your machine, or that you use one of the free online email services, such as Gmail or Outlook. For this example, we will log in to `http://www.gmail.com` using the SMTP and send email with our data.

Without further ado, we will modify our script and add the SMTP support to it.

We will import the required modules into Python. Again, `smtplib` and `platform` are needed for this task:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import smtplib
imp        ort platform
```

This is the part of the function that contains both the `check_feature()` and `get_value_from_string()` functions:

```
def check_feature(feature,string):
    if feature in string.lower():
        return True
    else:
        return False

def get_value_from_string(key,string):
    value = "NONE"
    for line in string.split("\n"):
        if key in line:
            value = line.split(":")[1].strip()
    return value
```

Finally, the main body of the Python script is as follows, containing the Python logic to get the required information:

```
cpu_features = []
with open('/proc/cpuinfo') as cpus:
    cpu_data = cpus.read()
    num_of_cpus = cpu_data.count("processor")
    cpu_features.append("Number of Processors: {0}".format(num_of_cpus))
    one_processor_data = cpu_data.split("processor")[1]
    if check_feature("vmx",one_processor_data):
        cpu_features.append("CPU Virtualization: enabled")
    if check_feature("cpu_meltdown",one_processor_data):
        cpu_features.append("Known Bugs: CPU Metldown ")
    model_name = get_value_from_string("model name ",one_processor_data)
    cpu_features.append("Model Name: {0}".format(model_name))

    cpu_mhz = get_value_from_string("cpu MHz",one_processor_data)
    cpu_features.append("CPU MHz: {0}".format((cpu_mhz)))

memory_features = []
with open('/proc/meminfo') as memory:
    memory_data = memory.read()
    total_memory = get_value_from_string("MemTotal",memory_data).replace("
kB","")
    free_memory = get_value_from_string("MemFree",memory_data).replace("
kB","")
    swap_memory = get_value_from_string("SwapTotal",memory_data).replace("
kB","")
    total_memory_in_gb = "Total Memory in GB:
{0}".format(int(total_memory)/1024)
    free_memory_in_gb = "Free Memory in GB:
{0}".format(int(free_memory)/1024)
    swap_memory_in_gb = "SWAP Memory in GB:
{0}".format(int(swap_memory)/1024)
    memory_features =
[total_memory_in_gb,free_memory_in_gb,swap_memory_in_gb]

Data_Sent_in_Email = ""
Header = """From: PythonEnterpriseAutomationBot <basim.alyy@gmail.com>
To: To Administrator <basim.alyy@gmail.com>
Subject: Monitoring System Report

"""
Data_Sent_in_Email += Header
Data_Sent_in_Email +="============System Information============"

Data_Sent_in_Email +="""
```

```
System Type: {0}
Hostname: {1}
Kernel Version: {2}
System Version: {3}
Machine Architecture: {4}
Python version: {5}
""".format(platform.system(),
           platform.uname()[1],
           platform.uname()[2],
           platform.version(),
           platform.machine(),
           platform.python_version())


Data_Sent_in_Email +="============CPU Information============\n"
Data_Sent_in_Email +="\n".join(cpu_features)


Data_Sent_in_Email +="\n============Memory Information============\n"
Data_Sent_in_Email +="\n".join(memory_features)
```

At the end, we need to populate the variables with some values to properly connect to the `gmail` server:

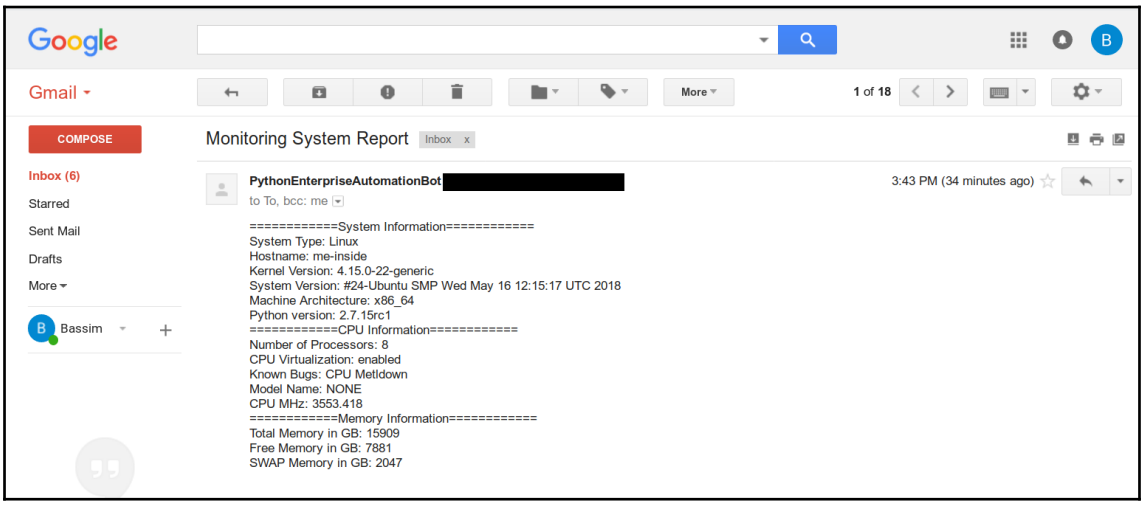
```
fromaddr = 'yyyyyyyyyyy@gmail.com'
toaddrs  = 'basim.alyy@gmail.com'
username = 'yyyyyyyyyyy@gmail.com'
password = 'xxxxxxxxxx'
server = smtplib.SMTP('smtp.gmail.com:587')
server.ehlo()
server.starttls()
server.login(username,password)

server.sendmail(fromaddr, toaddrs, Data_Sent_in_Email)
server.quit()
```

In the preceding example, the following applies:

1. The first part is the same as the original example, but instead of printing the data to the terminal, we add it to the `Data_Sent_in_Email` variable.
2. The `Header` variable represents the email header containing the sender's address, the recipient's address, and the email's subject.
3. We use the `SMTP()` class inside of the `smtplib` module to connect to the public Gmail SMTP server and negotiate the TTLS connection. This is the default method when connecting to Gmail servers. We hold the SMTP connection in the `server` variable.

4. Now, we log in to the server by using the `login()` method, and finally, we use the `sendmail()` function to send the email. `sendmail()` accepts three arguments: the sender, the recipient, and the email body.
5. Finally, we close the connection with the server:

Script output



Using the time and date modules

Great; so far, we have been able to send custom data generated from our servers through email. However, there might be a difference in time between the generated data and the email's delivery time, due to network congestion or a failure in the mail system, or any other reason. So, we can't depend on the email to correlate the delivery time with the actual event time.

For that reason, we will use the Python `datetime` module to follow the current time on the monitored system. This module can format the time in many attributes, such as year, month, day, hour, and minute.

Aside from that, the `datetime` instance from the `datetime` module is actually a standalone object in Python (like int, string, boolean, and so on); hence, it has its own attributes inside of Python.

To convert the `datetime` object to a string, you can use the `strftime()` method, which is available as an attribute inside of the created object. Also, it provides a method for formatting the time by using the following directives:

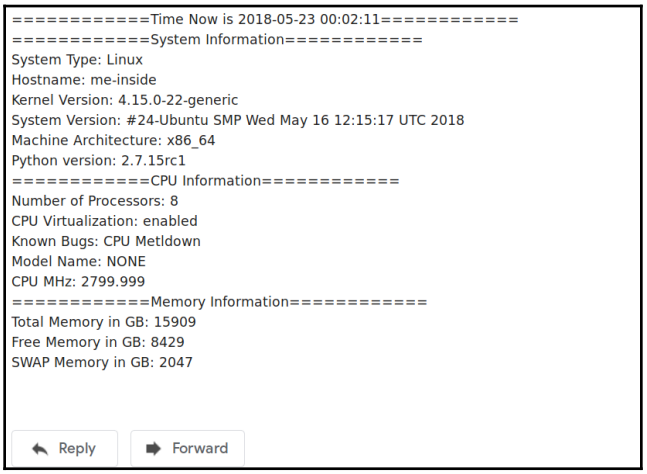
Directive	**Meaning**
`%Y`	Returns the year, from 0001 to 9999
`%m`	Returns the month number
`%d`	Returns the day of the month
`%H`	Returns the hour number, 0-23
`%M`	Returns the minutes, 0-59
`%S`	Returns the seconds,0-59

So, we will tweak our script and add the following snippet to the code:

```
from datetime import datetime
time_now = datetime.now()
time_now_string = time_now.strftime("%Y-%m-%d %H:%M:%S")
Data_Sent_in_Email += "====Time Now is {0}====\n".format(time_now_string)
```

First, we imported the `datetime` class from the `datetime` module. Then, we created the `time_now` object using the `datetime` class and the `now()` function, which returns the current time on the running system. Finally, we used `strftime()`, with a directive, to format the time in a specific format and convert it to a string for printing (remember, the object has a `datetime` object).

The script's output is as follows:



Running the script on a regular basis

A final step in the script is to schedule the script to run at a time interval. This can be daily, weekly, hourly, or at a specific time. This can be done using the `cron` job on Linux systems. `cron` is used to schedule a repeated event, such as cleaning up directories, backing up databases, rotating logs, or anything else you can think of.

To view the current jobs scheduled, use the following command:

```
crontab -l
```

To edit `crontab`, use the `-e` switch. If this is the first time you are running `cron`, you will be prompted to use your favorite editor (`nano` or `vi`).

A typical `crontab` consists of five stars, each one representing a time entry:

Field	Values
Minutes	0-59
Hours	0-23
Day of the month	1-31
Month	1-12
Day of the week	0-6 (Sunday - Saturday)

For example, if you need to schedule a job to run every Friday at 9:00 P.M. you will use the following entry:

```
0 21 * * 5 /path/to/command
```

If you need to have a command every day at 12:00 A.M. (a backup, for example), use the following `cron` job:

```
0 0 * * * /path/to/command
```

Also, you can schedule the `cron` to run at *every* specific interval. For example, if you need to run a job every `5` minutes, use this `cron` job:

```
*/5 * * * * /path/to/command
```

Back to our script; we can schedule it to run every day at 7:30 AM:

```
30 7 * * * /usr/bin/python /root/Send_Email.py
```

Finally, remember to save the `cron` job before exiting.

It's better to provide a full command path to Linux, rather than a relative path, to avoid any potential issues.

Managing users in Ansible

Now, we will discuss how to manage users in different systems.

Linux systems

Ansible provides powerful user management modules to manage different tasks on a system. We have a chapter dedicated to discussing Ansible (`Chapter 13`, *Ansible for System Administration*), but in this chapter, we will explore its power for managing user accounts across a company's infrastructure.

Sometimes, companies allow root access to all users, to get rid of the headache of user management; this is not a good solution in terms of security and auditing. It's the best practice to give the right permissions to the right users, and to revoke them once users leave the company.

Ansible provides an unmatched way to manage users across multiple servers, through either password or password-less (SSH key) access.

There are a few other things that need to be taken into consideration when creating users in a Linux system. The user must have a shell (such as Bash, CSH, ZSH, and so on) in order to log in to the server. Also, the user should have a home directory (usually under `/home`). Finally, the user must be in a group that determines its privileges and permissions.

Our first example will be creating a user with an SSH key in the remote server, using the ad hoc command. The key source is at the `ansible` tower, while we execute the command on `all` servers:

```
ansible all –m copy –a "src=~/id_rsa dest=~/.ssh/id_rsa mode=0600"
```

The second example is creating a user using the Playbook:

```
---
- hosts: localhost
  tasks:
    - name: create a username
```

```
    user:
      name: bassem
      password: "$crypted_value$"
      groups:
        - root
      state: present
      shell: /bin/bash
      createhome: yes
      home: /home/bassem
```

Let's look at the task's parameters:

- In our tasks, we use a user module that contains several parameters, such as `name`, that set the username for the user.
- The second parameter is `password`, where we set the user's password, but in a crypted format. You need to use the `mkpasswd` command, which prompts you for the password and will generate the hash value.
- `groups` is a list of groups that the user belongs to; hence, the user will inherit the permissions. You can use comma-separated values in this field.
- `state` is used to tell Ansible whether the user will be created or deleted.
- You can define the user shell used for remote access in the `shell` parameter.
- `createhome` and `home` are parameters used to specify the user's home location.

Another parameter is `ssh_key_file`, which specifies the SSH filename. Also, the `ssh_key_passphrase` will specify the passphrase for the SSH key.

Microsoft Windows

Ansible provides the `win_user` module to manage local Windows user accounts. This is very useful when creating users on active directory domains or Microsoft SQL databases (`mssql`), or when creating default accounts on normal PCs. The following example will create a user called `bassem` and give it the password `access123`. The difference here is that the password is given in plain text and not in the crypted value, as in the Unix-based system:

```
- hosts: localhost
  tasks:
    - name: create user on windows machine
      win_user:
        name: bassem
        password: 'access123'
        password_never_expires: true
```

```
account_disabled: no
account_locked: no
password_expired: no
state: present
groups:
  - Administrators
  - Users
```

The `password_never_expires` parameter will prevent Windows from expiring the password after a specific time; this is useful when creating admin and default accounts. On the other hand, `password_expired`, if set to `yes`, will require the user to enter a new password and change it upon first login.

The `groups` parameter will add the user from a listed value or comma-separated list of groups. This will depend on the `groups_action` parameter, and could be `add`, `replace`, or `remove`.

Finally, the state will tell Ansible what should be done to the user. This parameter could be `present`, `absent`, or `query`.

Summary

In this chapter, we learned about collecting data and reports from Linux machines and alerting through email using time and date modules. We also learned how to manage users in Ansible.

In the next chapter, we will learn how to interact with DBMS using Python connectors.

12
Interacting with the Database

In previous chapters, we generated several different reports, using many Python utilities and tools. In this chapter, we will utilize Python libraries to connect to external databases and submit the data we have generated. This data can then be accessed by external applications to get information.

Python provides a wide range of libraries and modules that cover managing and working on popular **Database Management Systems** (**DBMSes**), such as MySQL, PostgreSQL, and Oracle. In this chapter, we will learn how to interact with a DBMS and fill it with our own data.

The following topics will be covered in this chapter:

- Installing MySQL on an automation server
- Accessing the MySQL database from Python

Installing MySQL on an automation server

The first thing that we need to do is set up a database. In the following steps, we will cover how to install the MySQL database on our automation server, which we created in `Chapter 8`, *Preparing a Lab Environment*. Basically, you will need a Linux-based machine (CentOS or Ubuntu) with an internet connection to download the SQL packages. MySQL is an open source DBMS that uses a relational database and the SQL language to interact with data. In CentOS 7, MySQL is replaced with another, forked version, called MariaDB; both have the same source code, with some enhancements in MariaDB.

Follow these steps to install MariaDB:

1. Use the `yum` package manager (or `apt`, in the case of Debian-based systems) to download the `mariadb-server` package, as shown in the following snippet:

```
yum install mariadb-server -y
```

2. Once the installation has completed successfully, start the `mariadb` daemon. Also, we need to enable it at the operating system startup using the `systemd` command:

```
systemctl enable mariadb ; systemctl start mariadb

Created symlink from /etc/systemd/system/multi-
user.target.wants/mariadb.service to
/usr/lib/systemd/system/mariadb.service.
```

3. Validate the database status by running the following command, and make sure that the output contains `Active:active (running)`:

```
systemctl status mariadb

● mariadb.service - MariaDB database server
   Loaded: loaded (/usr/lib/systemd/system/mariadb.service;
enabled; vendor preset: disabled)
   Active: active (running) since Sat 2018-04-07 19:47:35 EET; 1min
34s ago
```

Securing the installation

The next, logical step after installation is securing it. MariaDB includes a security script that changes the options inside the MySQL configuration files, like creating the root password for accessing the database and allowing remote access. Run the following commands to launch the script:

```
mysql_secure_installation
```

The first prompt asks you to provide the root password. This root password is not the Linux root username, but the root password for the MySQL database; since this is a fresh installation, we have not set it yet, so we will simply press *Enter* to go to the next step:

```
Enter current password for root (enter for none): <PRESS_ENTER>
```

The script will suggest setting the password for the root. We will accept the suggestion by pressing `Y` and entering the new password:

```
Set root password? [Y/n] Y
New password:EnterpriseAutomation
Re-enter new password:EnterpriseAutomation
Password updated successfully!
Reloading privilege tables..
 ... Success!
```

The following prompts will suggest removing the anonymous users from administrating and accessing the database, which is highly recommended:

```
Remove anonymous users? [Y/n] y
 ... Success!
```

You can run SQL commands from a remote machine to the database hosted in your automation servers; this requires you to give a special privilege to root users, so they can access the database remotely:

```
Disallow root login remotely? [Y/n] n
 ... skipping.
```

Finally, we will remove the testing database, which anyone can access, and reload the privileges tables to ensure that all changes will take effect immediately:

```
Remove test database and access to it? [Y/n] y
 - Dropping test database...
 ... Success!
 - Removing privileges on test database...
 ... Success!

Reload privilege tables now? [Y/n] y
 ... Success!

Cleaning up...

All done!  If you've completed all of the above steps, your MariaDB
installation should now be secure.

Thanks for using MariaDB!
```

We have finished securing the installation; now, let's validate it.

Verifying the database installation

The first step after MySQL installation is to validate it. We need to verify that the `mysqld` daemon has started and is listening to port `3306`. We will do that by running the `netstat` command and `grep` on the listening port:

```
netstat -antup | grep -i 3306
tcp   0   0 0.0.0.0:3306      0.0.0.0:*         LISTEN      3094/mysqld
```

This means that the `mysqld` service can accept incoming connections from any IP over the port `3306`.

If you have `iptables` running on your machine, you need to add a rule to `INPUT` a chain, in order to allow remote hosts to connect to the MySQL database. Also, validate that `SELINUX` has the proper policies.

The second verification is through connecting to the database using the `mysqladmin` utility. This tool is included in MySQL clients and allows you to execute commands remotely (or locally) on the MySQL database:

```
mysqladmin -u root -p ping
Enter password:EnterpriseAutomation

mysqld is alive
```

Switch Name	Meaning
`-u`	Specifies the username.
`-p`	Makes MySQL prompt you with the username's password.
`ping`	Operation name to validate whether the MySQL database is alive or not.

The output indicates that the MySQL installation has completed successfully, and we're ready to move to the next step.

Accessing the MySQL database from Python

The Python developer creates the `MySQLdb` module, which provides a utility to interact and manage the database from a Python script. This module can be installed using Python's `pip`, or with an operating system package manager, such as `yum` or `apt`.

To install the package, use the following command:

```
yum install MySQL-python
```

Verify the installation as follows:

```
[root@AutomationServer ~]# python
Python 2.7.5 (default, Aug  4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import MySQLdb
>>>
```

Since the module has imported without any errors, we know that the Python module has successfully installed.

We will now access the database through the console and create a simple database called `TestingPython`, with one table inside it. We will then connect to it from Python:

```
[root@AutomationServer ~]# mysql -u root -p
Enter password: EnterpriseAutomation
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 12
Server version: 5.5.56-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

MariaDB [(none)]> CREATE DATABASE TestingPython;
Query OK, 1 row affected (0.00 sec)
```

In the preceding statements, we connected to the database using the MySQL utility, then used the SQL `CREATE` command to create a blank, new database.

You can verify the newly created database by using the following commands:

```
MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| TestingPython      |
| mysql              |
```

```
| performance_schema |
+--------------------+
4 rows in set (0.00 sec)
```

It's not mandatory to write SQL commands in uppercase; however, it's a best practice, in order to differentiate them from variables and other operations.

We need to switch to the new database:

```
MariaDB [(none)]> use TestingPython;
Database changed
```

Now, execute the following command to create a new table inside the database:

```
MariaDB [TestingPython]> CREATE TABLE TestTable (id INT PRIMARY KEY, fName
VARCHAR(30), lname VARCHAR(20), Title VARCHAR(10));
Query OK, 0 rows affected (0.00 sec)
```

When you're creating a table, you should specify the column type. For example, `fname` is a string with a maximum of 30 characters, while `id` is an integer.

Verify the table creation as follows:

```
MariaDB [TestingPython]> SHOW TABLES;
+--------------------------+
| Tables_in_TestingPython |
+--------------------------+
| TestTable                |
+--------------------------+
1 row in set (0.00 sec)

MariaDB [TestingPython]> describe TestTable;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | int(11)     | NO   | PRI | NULL    |       |
| fName | varchar(30) | YES  |     | NULL    |       |
| lname | varchar(20) | YES  |     | NULL    |       |
| Title | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

Querying the database

At this point, our database is ready for some Python script. Let's create a new Python file and provide database parameters:

```
import MySQLdb
SQL_IP ="10.10.10.130"
SQL_USERNAME="root"
SQL_PASSWORD="EnterpriseAutomation"
SQL_DB="TestingPython"

sql_connection = MySQLdb.connect(SQL_IP,SQL_USERNAME,SQL_PASSWORD,SQL_DB)
print sql_connection
```

The parameters provided (`SQL_IP`, `SQL_USERNAME`, `SQL_PASSWORD`, and `SQL_DB`) are needed to establish the connection and authenticate against the database on port `3306`.

The following table mentions the parameters and their meaning:

Parameter	Meaning
`host`	The server IP address that has the `mysql` installation.
`user`	The username with administrative privileges over the connected database.
`passwd`	The password created using the `mysql_secure_installation` script.
`db`	The database name.

The output will be as follows:

```
<_mysql.connection open to '10.10.10.130' at 1cfd430>
```

The returned object indicates that the connection has successfully opened to the database. Let's use this object to create the SQL cursor that is used to execute the actual commands:

```
cursor = sql_connection.cursor()
cursor.execute("show tables")
```

You can have many cursors associated with a single connection, and any change in one cursor will be immediately reported to other ones, as you have the same connection opened.

The cursor has two main methods: `execute()` and `fetch*()`.

The `execute()` method is used to send commands to the database and return the query results, while the `fetch*()` method has three flavors:

Method Name	Description
`fetchone()`	Fetches only one record from the output, regardless of the number of returned rows.
`fetchmany(num)`	Returns the number of records specified inside the method.
`fetchall()`	Returns all records.

Since `fetchall()` is a generic method that fetches either one record or all records, we will use it:

```
output = cursor.fetchall()
print(output)

# python mysql_simple.py
(('TestTable',),)
```

Inserting records into the database

The `MySQLdb` module allows us to insert records into the database using the same cursor operation. Remember that the `execute()` method can be used for both insertion and query. Without further ado, we will change our script a bit and provide the following `insert` commands:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import MySQLdb

SQL_IP ="10.10.10.130"
SQL_USERNAME="root"
SQL_PASSWORD="EnterpriseAutomation"
SQL_DB="TestingPython"

sql_connection = MySQLdb.connect(SQL_IP,SQL_USERNAME,SQL_PASSWORD,SQL_DB)

employee1 = {
    "id": 1,
    "fname": "Bassim",
    "lname": "Aly",
    "Title": "NW_ENG"
}
```

```
employee2 = {
    "id": 2,
    "fname": "Ahmed",
    "lname": "Hany",
    "Title": "DEVELOPER"
}

employee3 = {
    "id": 3,
    "fname": "Sara",
    "lname": "Mosaad",
    "Title": "QA_ENG"
}

employee4 = {
    "id": 4,
    "fname": "Aly",
    "lname": "Mohamed",
    "Title": "PILOT"
}

employees = [employee1,employee2,employee3,employee4]

cursor = sql_connection.cursor()

for record in employees:
    SQL_COMMAND = """INSERT INTO TestTable(id,fname,lname,Title) VALUES
({0},'{1}','{2}','{3}')""".format(record['id'],record['fname'],record['lnam
e'],record['Title'])

    print SQL_COMMAND
    try:
        cursor.execute(SQL_COMMAND)
        sql_connection.commit()
    except:
        sql_connection.rollback()

sql_connection.close()
```

In the preceding example, the following applies:

- We defined four employee records as a dictionary. Each one has an `id`, `fname`, `lname`, and `title`, in keys, with different values for each.
- Then, we grouped them using `employees`, which is a variable of the `list` type.

- A `for` loop was created to iterate over the `employees` list and, inside the loop, we formatted the `insert` SQL command and used the `execute()` method to push the data to the SQL database. Notice that it's not required to add a semicolon (`;`) after the command inside the `execute` function, as it will be added automatically.
- After each successful execution of the SQL command, the `commit()` operation will be used to force the database engine to commit the data; otherwise, the connection will be rolled back.
- Finally, use the `close()` function to terminate the established SQL connection.

Closing the database connection means that all the cursors are sent to Python garbage collectors and will be unusable. Also, note that when you close the connection without committing the changes, it will make the database engine immediately roll back all transactions.

The script's output is as follows:

```
# python mysql_insert.py
INSERT INTO TestTable(id,fname,lname,Title) VALUES
(1,'Bassim','Aly','NW_ENG')
INSERT INTO TestTable(id,fname,lname,Title) VALUES
(2,'Ahmed','Hany','DEVELOPER')
INSERT INTO TestTable(id,fname,lname,Title) VALUES
(3,'Sara','Mosad','QA_ENG')
INSERT INTO TestTable(id,fname,lname,Title) VALUES
(4,'Aly','Mohamed','PILOT')
```

You can query the database through the MySQL console to verify that the data has been submitted to the database:

```
MariaDB [TestingPython]> select * from TestTable;
+----+--------+---------+-----------+
| id | fName  | lname   | Title     |
+----+--------+---------+-----------+
|  1 | Bassim | Aly     | NW_ENG    |
|  2 | Ahmed  | Hany    | DEVELOPER |
|  3 | Sara   | Mosaad  | QA_ENG    |
|  4 | Aly    | Mohamed | PILOT     |
+----+--------+---------+-----------+
```

Now, returning to our Python code, we can use the `execute()` function again; this time, we use it to select all the data that we inserted inside the `TestTable`:

```
import MySQLdb

SQL_IP ="10.10.10.130"
SQL_USERNAME="root"
SQL_PASSWORD="EnterpriseAutomation"
SQL_DB="TestingPython"

sql_connection = MySQLdb.connect(SQL_IP,SQL_USERNAME,SQL_PASSWORD,SQL_DB)
# print sql_connection

cursor = sql_connection.cursor()
cursor.execute("select * from TestTable")

output = cursor.fetchall()
print(output)
```

The script's output is as follows:

```
python mysql_show_all.py
((1L, 'Bassim', 'Aly', 'NW_ENG'), (2L, 'Ahmed', 'Hany', 'DEVELOPER'), (3L,
'Sara', 'Mosaa    d', 'QA_ENG'), (4L, 'Aly', 'Mohamed', 'PILOT'))
```

The `L` character after the `id` value in the previous example can be resolved by converting the data to integer again (in Python), using the `int()` function.

Another useful attribute inside of the cursor is `.rowcount`. This attribute will indicate how many rows are returned as a result of the last `.execute()` method.

Summary

In this chapter, we learned how to interact with a DBMS using Python connectors. We installed a MySQL database on our automation server, and then verified it. Then, we accessed the MySQL DB using a Python script, and performed operations on it.

In the next chapter, we will learn how to use Ansible for system administration.

13
Ansible for System Administration

In this chapter, we will explore one of the popular automation frameworks used by thousands of network and system engineers called *Ansible*, Ansible is used to administrate servers and network devices over multiple transport protocols such as SSH, Netconf, and API in order to deliver a reliable infrastructure.

We will start first by learning the terminologies used in ansible, how to construct an inventory file that contains the infrastructure access details, Building a robust Ansible playbook using features like conditions, loops, and template rendering.

Ansible belongs to the configuration management class of software; it is used to manage the configuration life cycle on multiple different devices and servers, making sure that the same steps are applied on all of them and help to create Infrastructure as a code (IaaC) environment.

The following topics will be covered in this chapter:

- Ansible and its terminology
- Installing Ansible on Linux
- Using Ansible in ad hoc mode
- Create your first playbook
- Understanding Ansible conditions, handlers, and loops
- Working with Ansible facts
- Working with the Ansible template

Ansible terminology

Ansible is an automation tool and a complete framework that provides an abstraction layer based on Python tools. Originally, it was designed to handle task automation. This task might be executed on a single server or on thousands of servers and ansible will handle them without any problem; later, Ansible's scope extended to network devices and cloud providers. Ansible follows the concept of `idempotency`, wherein Ansible instructions can run the same task multiple times and always give the same configuration on all devices at the end, reaching a desired state with minimal changes. For example, if we run Ansible to upload a file to a specific group of servers, then run it again, Ansible will first validate if the file already exist in the remote destination as a result a previous execution or not. if it exist, then the ansible won't upload it

again. This feature called **idempotency**.

Another aspect of Ansible is that it is agentless. Ansible doesn't require any agents to be installed in the servers before it runs tasks. It leverages the SSH connection and Python standard libraries to execute tasks on remote servers and return the output to the Ansible server. Also, it doesn't create a database to store remote machine information, but depends on a flat text file called `inventory` to store all required server information, such as IP addresses, credentials, and infrastructure categorization. The following is an example of a simple inventory file:

```
[all:children]
web-servers
db-servers

[web-servers]
web01 Ansible_ssh_host=192.168.10.10


[db-servers]
db01 Ansible_ssh_host=192.168.10.11
db02 Ansible_ssh_host=192.168.10.12

[all:vars]
Ansible_ssh_user=root
Ansible_ssh_pass=access123

[db-servers:vars]
Ansible_ssh_user=root
Ansible_ssh_pass=access123
```

```
[local]
127.0.0.1 Ansible_connection=local
Ansible_python_interpreter="/usr/bin/python"
```

Notice that we group together servers that perform the same functions in our infrastructure (such as database servers, in a group called `[db-servers]`; the same goes for `[web-servers]`). Then, we define a special group, called `[all]`, that combines both groups, in case we have a task targeted to all of our servers.

The keyword `children`, in `[all:children]`, means that the entries inside the group are also groups that contain hosts.

Ansible's **ad hoc** mode allows users to execute tasks directly from the Terminal, towards the remote servers. Let's suppose that you want to update specific packages on specific types of servers, such as databases or web backend servers, to resolve a new bug. At the same time, you don't want to go all the way to developing a complex playbook to execute a simple task. By leveraging the ad hoc mode in Ansible, you can execute any command on the remote servers by typing it on the Ansible host Terminal. Even some modules can be executed in the Terminal; we will see that in the *Using Ansible in ad hoc mode* section.

Installing Ansible on Linux

The Ansible package is available on all major Linux distributions. In this section, we will install it onto both Ubuntu and CentOS machines. The Ansible 2.5 release was used at the time of developing this book, and it provides support for both Python 2.6 and Python 2.7. Also, starting from version 2.2, Ansible provides a tech preview for Python 3.5+.

On RHEL and CentOS

You will need to have the EPEL repository installed and enabled before installing Ansible. To do so, use the following command:

```
sudo yum install epel-release
```

Then, proceed with the Ansible package installation, as shown in the following command:

```
sudo yum install Ansible
```

Ubuntu

First, make sure that your system is up to date, and add the Ansible channel. Finally, install the Ansible package itself, as shown in the following snippet:

```
$ sudo apt-get update
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:Ansible/Ansible
$ sudo apt-get update
$ sudo apt-get install Ansible
```

For more installation flavors, you can check the official Ansible website (`http://docs.Ansible.com/Ansible/latest/installation_guide/intro_installation.html?#installing-the-control-machine`).

You can validate your installation by running `Ansible --version` to check the installed version:

```
bassim@me-inside:~$ ansible --version
ansible 2.5.1
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/bassim/.ansible/plugins/modules', u'/usr/sha
re/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/dist-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.14 (default, Sep 23 2017, 22:06:14) [GCC 7.2.0]
bassim@me-inside:~$
```

The Ansible configuration files are usually stored in `/etc/Ansible`, with the filename `Ansible.cfg`.

Using Ansible in ad hoc mode

Ansible ad hoc mode is used when you need to execute simple operations on remote machines, without creating complex and persistent tasks. This is where a user usually starts when they first work on Ansible, before performing advanced tasks in a playbook.

Executing the ad-hoc command requires two things. First, you will need the host or group from the inventory file; secondly, you will need the Ansible module that you want to execute towards the target machine:

1. First, let's define our hosts and add the CentOS and Ubuntu machines in a separate group:

```
[all:children]
centos-servers
ubuntu-servers

[centos-servers]
centos-machine01 Ansible_ssh_host=10.10.10.193

[ubuntu-servers]
ubuntu-machine01 Ansible_ssh_host=10.10.10.140

[all:vars]
Ansible_ssh_user=root
Ansible_ssh_pass=access123

[centos-servers:vars]
Ansible_ssh_user=root
Ansible_ssh_pass=access123

[ubuntu-servers:vars]
Ansible_ssh_user=root
Ansible_ssh_pass=access123

[routers]
gateway ansible_ssh_host = 10.10.88.110 ansible_ssh_user=cisco
ansible_ssh_pass=cisco

[local]
127.0.0.1 Ansible_connection=local
Ansible_python_interpreter="/usr/bin/python"
```

2. Save this file as `hosts`, under `/root/` or your home directory in the `AutomationServer`.
3. Then, run the `Ansible` command with the `ping` module:

```
# Ansible -i hosts all -m ping
```

The `-i` argument will accept the inventory file that we added, while the `-m` argument will specify the name of the Ansible module.

After running the command, you will get the following output, indicating a failure in connecting to the remote machine:

```
ubuntu-machine01 | FAILED! => {
    "msg": "Using a SSH password instead of a key is not possible because
Host Key checking is enabled and sshpass does not support this.  Please add
this host's fingerprint to your known_hosts file to manage this host."
}
centos-machine01 | FAILED! => {
    "msg": "Using a SSH password instead of a key is not possible because
Host Key checking is enabled and sshpass does not support this.  Please add
this host's fingerprint to your known_hosts file to manage this host."
}
```

This happened because the remote machines are not inside of the `known_hosts` of the Ansible server; it can be solved through two methods.

The first method is SSHing to them manually, which will add the host fingerprint to the server. Or, you can completely disable host key checking in the Ansible configuration, as shown in the following snippet:

```
sed -i -e 's/#host_key_checking = False/host_key_checking = False/g'
/etc/Ansible/Ansible.cfg

sed -i -e 's/#   StrictHostKeyChecking ask/   StrictHostKeyChecking no/g'
/etc/ssh/ssh_config
```

Rerun the `Ansible` command, and you should get successful output from the three machines:

```
127.0.0.1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ubuntu-machine01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
centos-machine01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

The `ping` module in Ansible does not perform the ICMP operation against the device. It actually tries to log in to the device by using the SSH with provided credentials; if the login succeeds, it will return the `pong` keyword to the Ansible host.

Another useful module is `apt`, or `yum`, which is used to manage the package on either an Ubuntu or CentOS server. The following example will install the `apache2` package on the Ubuntu machines:

```
# Ansible –i hosts ubuntu-servers –m apt –a "name=apache2 state=present"
```

The state in the `apt` module can have the following values:

State	Action
`absent`	Removes the package from the system.
`present`	Makes sure that the package is installed on the system.
`latest`	Ensures that the package is in the latest version.

You can access the Ansible module documentation by running `Ansible-doc <module_name>`; you will see the full options, with examples, for the module.

The `service` module is used to manage operation and current status of the service. You can change the service status to either `started`, `restarted` or `stopped` in the `state` option and ansible will run the appropriate command to change the status. In the meantime, you can configure whether service is enabled at boot time or disabled by configuring the `enabled` .

```
#Ansible –i hosts centos-servers –m service –a "name=httpd state=stopped,
enabled=no"
```

Also, you can restart the service by providing the service name, with the `state` set as `restarted`:

```
#Ansible –i hosts centos-servers –m service –a "name=mariadb
state=restarted"
```

The other way to run Ansible in ad hoc mode is to pass the command directly to Ansible, using not the built-in modules but the `-a` argument:

```
#Ansible –i hosts all –a "ifconfig"
```

You can even reboot the servers by running the `reboot` command; but this time, we will only run it against the CentOS servers:

```
#Ansible -i hosts centos-servers -a "reboot"
```

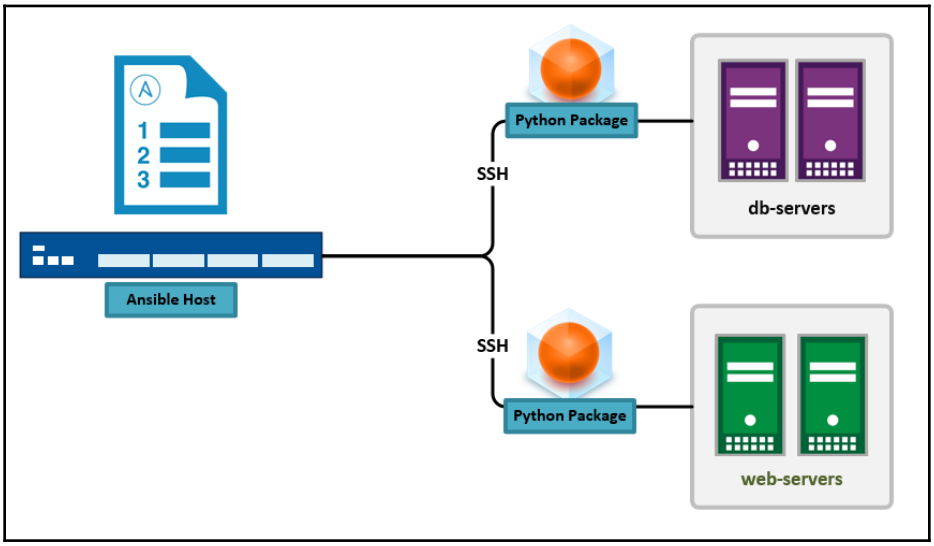
Sometimes, you will need to run the command (or the module) using a different user. This will be useful when you run a script on a remote server with specific permissions assigned to a user different than the SSH user. In that case, we will add the `-u`, `--become`, and `--ask-become-pass` (`-K`) switches. This will make Ansible run the command with the provided username and prompt you for the user's password:

```
#Ansible -i hosts ubuntu-servers --become-user bassim  --ask-become-pass -a
"cat /etc/sudoers"
```

How Ansible actually works

Ansible is basically written in Python, However it use it's own DSL (Domain Specific Language). You can write using this DSL and ansible will convert it to Python on remote machines to execute tasks. So, it first validates the task syntax and copies the module from the Ansible host to the remote server, and then executes it on the machine itself over SSH.

The result from the execution is returned back to the Ansible host in a `json` format, so you can match any returned values by knowing its key:



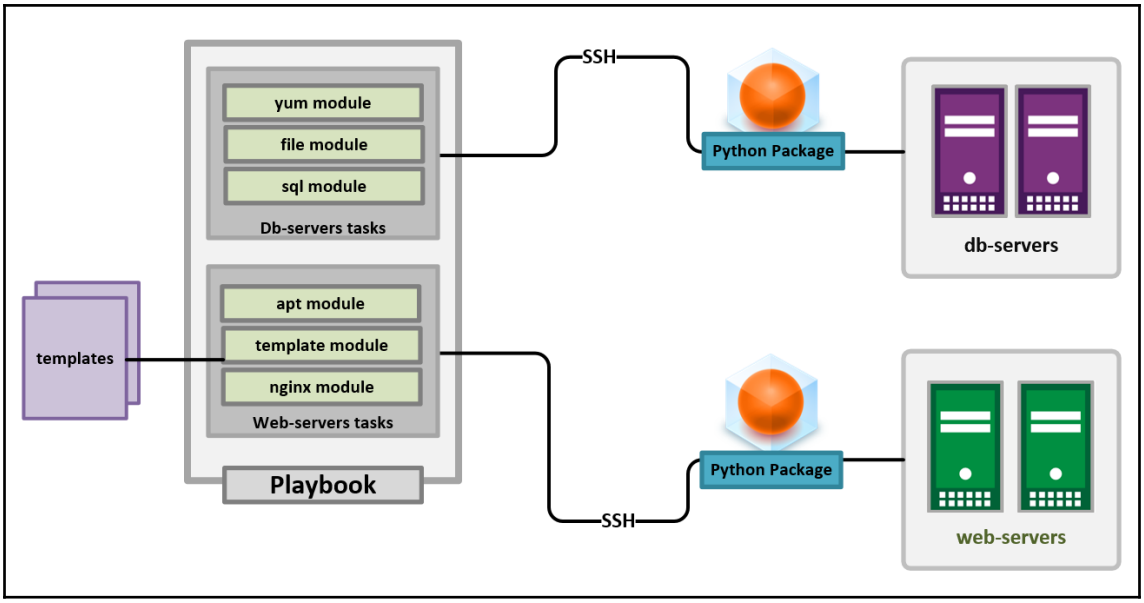
In the case of network devices where Python is installed on the **Network Operating System** (**NOS**), Ansible uses either an API or `netconf`, if the network device supports it (such as Juniper and Cisco Nexus); or, it just executes commands using the paramiko `exec_command()` function, and returns the output to the Ansible host. This can be done by using the `raw` module, as shown in the following snippet:

```
# Ansible -i hosts routers -m raw -a "show arp"
gateway | SUCCESS | rc=0 >>

Sat Apr 21 01:33:58.391 CAIRO

Address         Age        Hardware Addr   State      Type  Interface
85.54.41.9        -          45ea.2258.d0a9  Interface  ARPA
TenGigE0/2/0/0
10.88.18.1      -          d0b7.428b.2814  Satellite  ARPA  TenGigE0/2/0/0
192.168.100.1   -          00a7.5a3b.4193  Interface  ARPA
GigabitEthernet100/0/0/9
192.168.100.2   02:08:03   fc5b.3937.0b00  Dynamic    ARPA  \
```

Creating your first playbook

Now the magic party can begin. An Ansible playbook is a set of commands (called tasks) that need to be executed in order, and it describes the desired state of the hosts after execution finishes. Think of a playbook as a manual that contains a set of instructions for how to change the state of an infrastructure; each instruction depends on many built-in Ansible modules to perform the tasks. For example, you may have a playbook that is used to build web applications that consist of SQL servers, to act as backend databases and nginx web servers. The playbook will have a list of tasks to perform against each group of servers, to change their states from `No-Exist` to `Present`, or to `Restarted` or `Absent`, if you want to delete the web app.

The power of having the playbook, over the ad hoc commands is that you can use it to configure and set up your infrastructure everywhere. The same procedure used to create the dev environment will be used in the production environment. A playbook is used to create the automation workflow that runs on your infrastructure:



Playbooks are written with YAML, which we discussed in `Chapter 6`, *Configuration Generator with Python and Jinja2*. A playbook consists of multiple plays, executed against a set of hosts that are defined in the inventory file. The hosts will be converted to a Python `list`, and each item inside the list will be called a `play`. In the preceding example, the `db-servers` tasks are some of the plays, and are executed against the `db-servers` only. During playbook execution, you can decide to run all of the plays in the file, only a specific play, or tasks with specific tags, regardless of which play they belong to.

Now, let's look at our first playbook, to get the look and feel of it:

```
- hosts: centos-servers
  remote_user: root

  tasks:
    - name: Install openssh
      yum: pkg=openssh-server state=installed

    - name: Start the openssh
      service: name=sshd state=started enabled=yes
```

This is a simple playbook, with a single `play` that contains two `tasks`:

1. Install `openssh-server`.
2. Start the `sshd` service after installation, and make sure that it's available at the boot time.

Now, we need to apply this to a specific host (or a group of hosts). So, we set the `hosts` to be `CentOS-servers`, defined previously in the inventory file, and we also set the `remote_user` to be the root, to ensure that the tasks after it will be executed with root permissions.

The tasks will consist of the names and the Ansible modules. The name is used to describe the task. It's not mandatory to provide names for your tasks, but it's recommended, in case you need to start the execution from a specific task.

The second part is the Ansible module, which is mandatory. In our example, we used the core module `yum` to install the `openssh-server` package onto the target servers. The second task has the same structure, but this time, we will use another core module, called `service`, to start and enable the `sshd` daemon.

A final note is to watch the indentation for different components inside of Ansible. For example, the names of the tasks should be on the same level, while the `tasks` should align with the `hosts` on the same line.

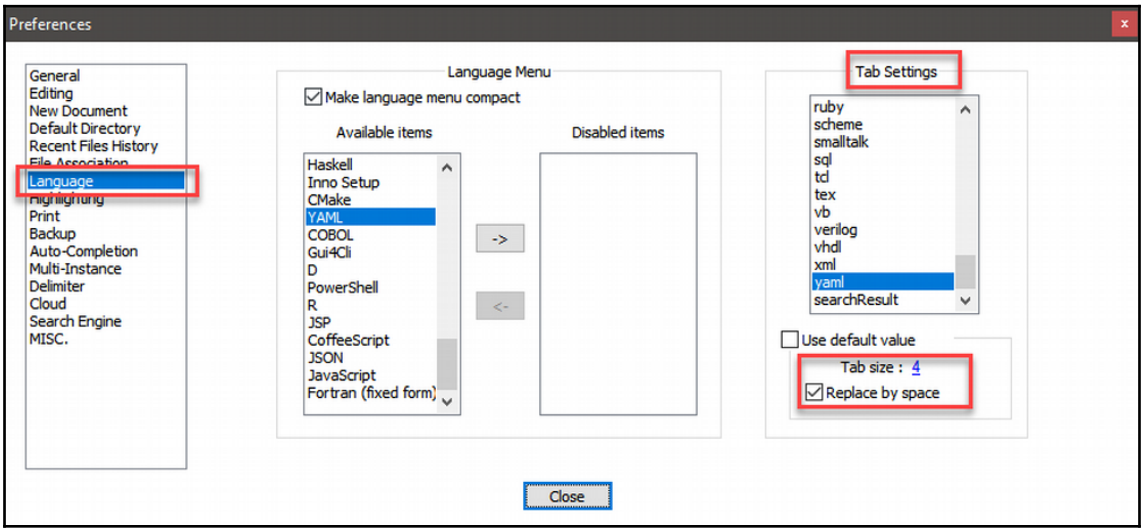
Let's run the playbook in our automation server and check the output:

```
#Ansible-playbook -i hosts first_playbook.yaml

PLAY [centos-servers]
*********************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [centos-machine01]

TASK [Install openssh]
*********************************************************************
ok: [centos-machine01]

TASK [Start the openssh]
*********************************************************************
ok: [centos-machine01]
```

```
PLAY RECAP
*********************************************************************
******
centos-machine01           : ok=3    changed=0    unreachable=0    failed=0
```

You can see that the playbook is executed on `centos-machine01`, and the tasks are executed sequentially, as defined in the playbook.

YAML requires that you preserve the indentation level and don't mix between the tabs and spaces; otherwise, it will give an error. Many text editors and IDEs will convert the tab to a set of white spaces. An example of that option is shown in the following screenshot, in the notepad++ editor preferences:



Understanding Ansible conditions, handlers, and loops

In this part of the chapter, we will look at some of the advanced features in the Ansible playbook.

Designing conditions

An Ansible playbook can execute tasks (or skip them) based on the results of specific conditions inside the task—for example, when you want to install packages on a specific family of operating systems (Debian or CentOS), or when the operating system is a particular version, or even when the remote hosts are virtual, not bare metal. This can be done by using the `when` clause inside of the task.

Let's enhance the previous playbook and limit the `openssh-server` installation to only CentOS based systems, so that it does not give an error when it hits an Ubuntu server that uses the `apt` module, not `yum`.

First, we will add the following two sections to our `inventory` file, to group the CentOS and Ubuntu machines in the `infra` section:

```
[infra:children]
centos-servers
ubuntu-servers


[infra:vars]
Ansible_ssh_user=root
Ansible_ssh_pass=access123
```

Then, we will redesign the tasks inside of the playbook to have the `when` clause, which limits task execution to only CentOS based machines. This should read as `if the remote machine is CentOS based, then I will execute the task; otherwise, skip`:

```
- hosts: infra
  remote_user: root

  tasks:
    - name: Install openssh
      yum: pkg=openssh-server state=installed
      when: Ansible_distribution == "CentOS"

    - name: Start the openssh
      service: name=sshd state=started enabled=yes
      when: Ansible_distribution == "CentOS"
```

Let's run the playbook:

```
# Ansible-playbook -i hosts using_when.yaml

PLAY [infra]
*********************************************************************
****

TASK [Gathering Facts]
*********************************************************************
ok: [centos-machine01]
ok: [ubuntu-machine01]

TASK [Install openssh]
*********************************************************************
skipping: [ubuntu-machine01]
ok: [centos-machine01]

TASK [Start the openssh]
*********************************************************************
skipping: [ubuntu-machine01]
ok: [centos-machine01]

PLAY RECAP
*********************************************************************
******
centos-machine01           : ok=3    changed=0    unreachable=0    failed=0
ubuntu-machine01           : ok=1    changed=0    unreachable=0    failed=0
```

Notice that the playbook first gathers the facts about the remote machines (we will discuss that later in this chapter), and then checks the operating system. The task will be skipped when it hits an `ubuntu-machine01`, and it will run normally on the CentOS.

You can also have multiple conditions that need to be true in order to run the task. For example, you can have the following playbook, which validates two things—first, that the machine is based on Debian, and second, that it is a virtual machine, not a baremetal:

```
- hosts: infra
  remote_user: root

  tasks:
    - name: Install openssh
      apt: pkg=open-vm-tools state=installed
      when:
        - Ansible_distribution == "Debian"
        - Ansible_system_vendor == "VMware, Inc."
```

Running this playbook will result in the following output:

```
# Ansible-playbook -i hosts using_when_1.yaml

PLAY [infra]
*********************************************************************
****

TASK [Gathering Facts]
*********************************************************************
ok: [centos-machine01]
ok: [ubuntu-machine01]

TASK [Install openssh]
*********************************************************************
skipping: [centos-machine01]
ok: [ubuntu-machine01]

PLAY RECAP
*********************************************************************
******
centos-machine01           : ok=1    changed=0    unreachable=0    failed=0
ubuntu-machine01           : ok=2    changed=0    unreachable=0    failed=0
```

The Ansible `when` clause also accepts expressions. For example, you can check whether a specific keyword exists in the returned output (that you saved using the register flag), and, based on that, execute the task.

The following playbook will validate the OSPF neighbor status. The first task will execute `show ip ospf neighbor` on the routers and register the output in a variable called `neighbors`. The next task will check for `EXSTART` or `EXCHANGE` in the returned output; if found, it will print a message back to the console:

```
hosts: routers

tasks:
  - name: "show the ospf neighbor status"
    raw: show ip ospf neighbor
    register: neighbors

  - name: "Validate the Neighbors"
    debug:
      msg: "OSPF neighbors stuck"
    when: ('EXSTART' in neighbors.stdout) or ('EXCHANGE' in
neigbnors.stdout)
```

You can check the facts commonly used in the `when` clause at `http://docs.Ansible.com/Ansible/latest/user_guide/playbooks_conditionals.html#commonly-used-facts`.

Creating loops in ansible

Ansible provides many ways to repeat the same task inside a play, but with a different value each time. For example, when you want to install multiple packages on a server, you don't need to create a task for each package. Rather, you can create a task that installs a package and provides a list of package names to the task, and Ansible will iterate over them until it finishes the installation. To accomplish this, we will need the `with_items` flag inside of the task that contains a list, and the variable `{{ item }}`, which serves as a placeholder for the items in the list. The playbook will leverage the `with_items` flag to iterate over a set of packages and provide them to the `yum` module, which requires the name and state of the package:

```
- hosts: infra
  remote_user: root

  tasks:
    - name: "Modifying Packages"
      yum: name={{ item.name }} state={{ item.state }}
      with_items:
        - { name: python-keyring-5.0-1.el7.noarch, state: absent }
        - { name: python-django, state: absent }
        - { name: python-django-bash-completion, state: absent }
        - { name: httpd, state: present }
        - { name: httpd-tools, state: present }
        - { name: python-qpid, state: present }
      when: Ansible_distribution == "CentOS"
```

You can hardcode the value of the state to be `present`; in that case, all of the packages will be installed. However, in the previous case, `with_items` will provide the two elements to the `yum` module.

The playbook's output is as follows:

```
PLAY [infra] *******************************************************************************

TASK [Gathering Facts] *********************************************************************
ok: [centos-machine01]
ok: [ubuntu-machine01]

TASK [Modifying Packages] ******************************************************************
skipping: [ubuntu-machine01] => (item={u'state': u'absent', u'name': u'python-keyring-5.0-1.
el7.noarch'})
skipping: [ubuntu-machine01] => (item={u'state': u'absent', u'name': u'python-django'})
skipping: [ubuntu-machine01] => (item={u'state': u'absent', u'name': u'python-django-bash-co
mpletion'})
skipping: [ubuntu-machine01] => (item={u'state': u'present', u'name': u'httpd'})
skipping: [ubuntu-machine01] => (item={u'state': u'present', u'name': u'httpd-tools'})
skipping: [ubuntu-machine01] => (item={u'state': u'present', u'name': u'python-qpid'})
ok: [centos-machine01] => (item={u'state': u'absent', u'name': u'python-keyring-5.0-1.el7.no
arch'})
ok: [centos-machine01] => (item={u'state': u'absent', u'name': u'python-django'})
ok: [centos-machine01] => (item={u'state': u'absent', u'name': u'python-django-bash-completi
on'})
changed: [centos-machine01] => (item={u'state': u'present', u'name': u'httpd'})
ok: [centos-machine01] => (item={u'state': u'present', u'name': u'httpd-tools'})
changed: [centos-machine01] => (item={u'state': u'present', u'name': u'python-qpid'})

PLAY RECAP *********************************************************************************
centos-machine01           : ok=2    changed=1    unreachable=0    failed=0
ubuntu-machine01           : ok=1    changed=0    unreachable=0    failed=0
```

Trigger tasks with handlers

Okay; you have installed and removed a series of packages in your system. You have copied files to/from your server. And you have changed many things in the server by using an Ansible playbook. Now, you need to restart a few other services, or add some lines to the files, to complete the configuration of the service. So, you should add a new task, right? Yes, that's correct. However, Ansible provides another great option, called **handlers**, which will not automatically execute when it hits (unlike tasks), but will rather be executed only when it is called. This provides you with the flexibility to call them upon the execution of tasks inside the play.

Handlers have the same alignment as the hosts and tasks, and are located at the bottom of each play. When you need to call a handler, you use the `notify` flag inside of the original task, to determine which handler will be executed; Ansible will link them together.

Let's look at an example. We will write a playbook that installs and configures the KVM on the CentOS servers. The KVM requires a few changes after installation, such as loading the `sysctl`, enabling the `kvm` and `802.1q` modules, and loading the `kvm` at `boot`:

```
- hosts: centos-servers
  remote_user: root

  tasks:
    - name: "Install KVM"
      yum: name={{ item.name }} state={{ item.state }}
      with_items:
        - { name: qemu-kvm, state: installed }
        - { name: libvirt, state: installed }
        - { name: virt-install, state: installed }
        - { name: bridge-utils, state: installed }

      notify:
        - load sysctl
        - load kvm at boot
        - enable kvm

  handlers:
    - name: load sysctl
      command: sysctl -p

    - name: enable kvm
      command: "{{ item.name }}"
      with_items:
        - {name: modprobe -a kvm}
        - {name: modprobe 8021q}
        - {name: udevadm trigger}

    - name: load kvm at boot
      lineinfile: dest=/etc/modules state=present create=True line={{
item.name }}
      with_items:
        - {name: kvm}
```

Notice the usage of `notify` after the installation task. When the task runs, it will notify three handlers in sequence, so that they will execute. The handlers will run after the task has successfully executed. That means that if the task has failed to run (for example, the `kvm` package was not found, or there's no internet connection to download it), there will be no changes to your system, and `kvm` will not be enabled.

Another awesome feature of the handler is that it's only run when there's a change in the task. For example, if you rerun the task, Ansible won't install the `kvm` package since it's already installed; it won't call any handlers, as it doesn't detect any changes in the system.

We will add a final note about two modules: `lineinfile` and `command`. The first module is actually inserting or deleting lines from configuration files by using regular expressions; we used it in order to insert the `kvm` into `/etc/modules`, to automatically boot the KVM when the machine starts. The second module, `command`, is used to execute a shell command directly on the device and return the output to the Ansible host.

Working with Ansible facts

Ansible is not only used to deploy and configure remote hosts. It can be used to gather all kinds of information and facts about them. The facts collection can take significant amount of time to collect everything from a busy system, but will provide a full view of the target machine.

The facts that are gathered can be used inside the playbook later, to design a task condition. For example, we used the `when` clause to limit the `openssh` installation to only CentOS-based systems:

```
when: Ansible_distribution == "CentOS"
```

You can enable/disable fact gathering in the Ansible plays by configuring `gather_facts` on the same level as hosts and tasks:

```
- hosts: centos-servers
  gather_facts: yes
  tasks:
    <your tasks go here>
```

Another way to gather facts and print them in Ansible is to use the `setup` module in the ad hoc mode. The returned results are in the form of nested dictionaries and lists, to describe the remote target facts, such as the server architecture, memory, networking settings, OS version, and so on:

```
#Ansible -i hosts ubuntu-servers -m setup | less
```


```
ubuntu-machine01 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.10.10.140"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::20c:29ff:feef:a88c"
        ],
        "ansible_apparmor": {
            "status": "enabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "09/17/2015",
        "ansible_bios_version": "6.00",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/vmlinuz-4.4.0-116-generic",
            "ro": true,
            "root": "/dev/mapper/ubuntu--machine--vg-root"
        },
        "ansible_date_time": {
            "date": "2018-04-26",
            "day": "26",
            "epoch": "1524699626",
            "hour": "01",
            "iso8601": "2018-04-25T23:40:26Z",
            "iso8601_basic": "20180426T014026018841",
            "iso8601_basic_short": "20180426T014026",
```


You can get to a specific value from the facts by using either a dot notation or square brackets. For example, to get the IPv4 address for `eth0`, you can use either `Ansible_eth0["ipv4"]["address"]` or `Ansible_eth0.ipv4.address`.

Working with the Ansible template

The last piece of working with Ansible is understanding how it handles the template. Ansible uses the Jinja2 template, which we discussed in `Chapter 6`, *Configuration Generator with Python and Jinja2*. It fills the parameters with either Ansible facts or the static values provided in the `vars` section, or even with the result of a task stored using the `register` flag.

In the following example, we will build an Ansible playbook that gathers the previous three cases. First, we define a variable called `Header` in the `vars` section, holding a welcome message as a static value. Then, we enable the `gather_facts` flag, to get all possible information from the target machine. Finally, we execute the `date` command, to get the current date in the server and store the output in the `date_now` variable:

```
- hosts: centos-servers
  vars:
    - Header: "Welcome to Server facts page generated from Ansible
playbook"
  gather_facts: yes
  tasks:
    - name: Getting the current date
      command: date
      register: date_now
    - name: Setup webserver
      yum: pkg=nginx state=installed
      when: Ansible_distribution == "CentOS"

      notify:
        - enable the service
        - start the service

    - name: Copying the index page
      template: src=index.j2 dest=/usr/share/nginx/html/index.html

  handlers:
    - name: enable the service
      service: name=nginx enabled=yes

    - name: start the service
      service: name=nginx state=started
```

The template module that was used in the preceding playbook will accept a Jinja2 file named `index.j2`, located in the same directory of the playbook; it will then provide all of the values for the jinj2 variables from the three sources we discussed previously. Then, the rendered file will be stored in a path provided by the `dest` option, inside the template module.

The content of `index.j2` will be as follows. It will be a simple HTML page that leverages the jinja2 language to generate a final HTML page:

```
<html>
<head><title>Hello world</title></head>
<body>
```

```
<font size="6" color="green">{{ Header }}</font>

<br>
<font size="5" color="#ff7f50">Facts about the server</font>
<br>
<b>Date Now is:</b> {{ date_now.stdout }}

<font size="4" color="#00008b">
<ul>
    <li>IPv4 Address: {{ Ansible_default_ipv4['address'] }}</li>
    <li>IPv4 gateway: {{ Ansible_default_ipv4['gateway'] }}</li>
    <li>Hostname: {{ Ansible_hostname }}</li>
    <li>Total Memory: {{ Ansible_memtotal_mb }}</li>
    <li>Operating System Family: {{ Ansible_os_family }}</li>
    <li>System Vendor: {{ Ansible_system_vendor }}</li>
</ul>
</font>
</body>
</html>
```

Running this playbook will result in installing the nginx web server on the CentOS machine, and adding an `index.html` page to it. You can access the page by using the browser:

You can also utilize the template module to generate network device configurations. The jinja2 templates used in `Chapter 6`, *Configuration Generator with Python and Jinja2*, which generated the `day0` and `day1` configurations for the router, can be reused inside of the Ansible playbook.

Summary

Ansible is a very powerful tool, used to automate IT infrastructure. It contains many modules and libraries that cover almost everything in system and network automation, making software deployment, package management, and configuration management very easy. While Ansible can execute a single module in ad hoc mode, the real power of Ansible is in writing and developing playbooks.

14
Creating and Managing VMware Virtual Machines

For a long long time, virtualization has been an important technology in the IT industry as it provides an efficient way for hardware resources and allows us to easily manage application life cycle inside the **Virtual Machine** (**VM**). In 2001, VMware released the first version of the ESXi that could run directly over the **commodity off the shelf** (**COTS**) **server** while converting it to a resource that could be consumed by multiple separate virtual machines. In this chapter, we will explore many options available to automate the building of virtual machine thanks to Python and Ansible.

The following topics will be covered in this chapter:

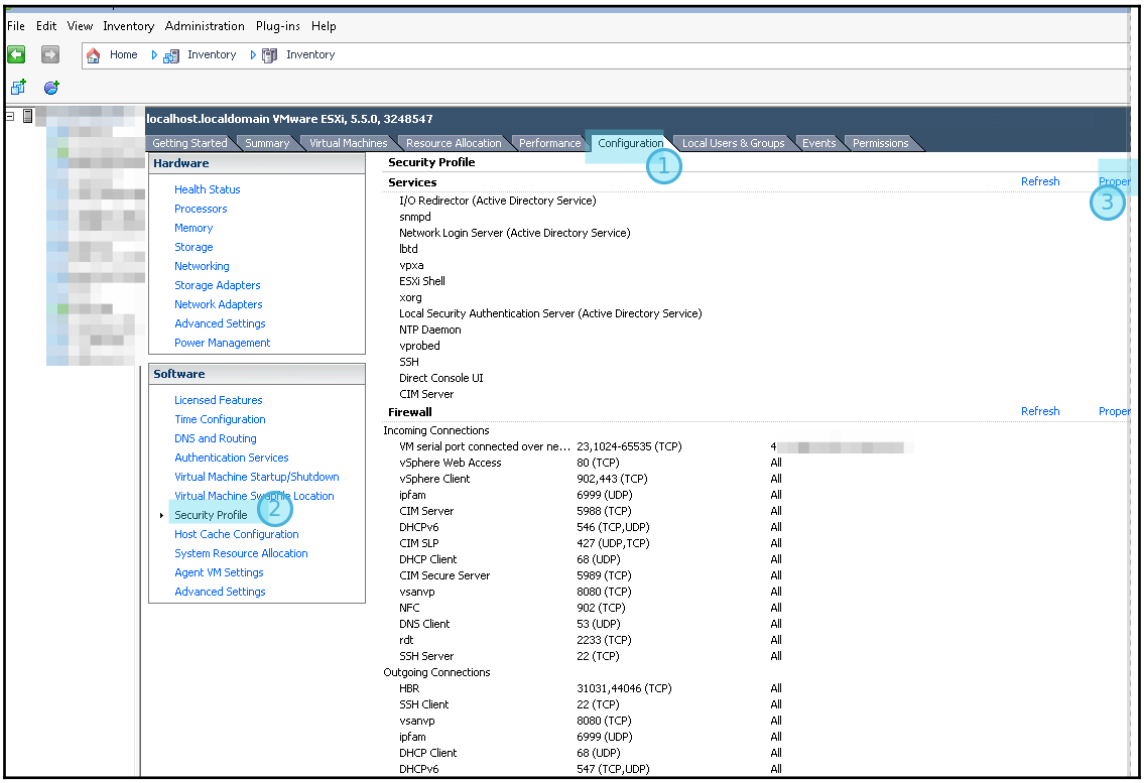
- Setting up the lab environment
- Generating a VMX file using Jinja2
- VMware Python clients
- Using Ansible playbooks to manage instances

Setting up the environment

For this chapter, we will have VMware ESXi version 5.5 installed over a Cisco UCS server and host a few virtual machines. We need to enable a few things in our ESXi server in order to expose some external ports to the outside world:

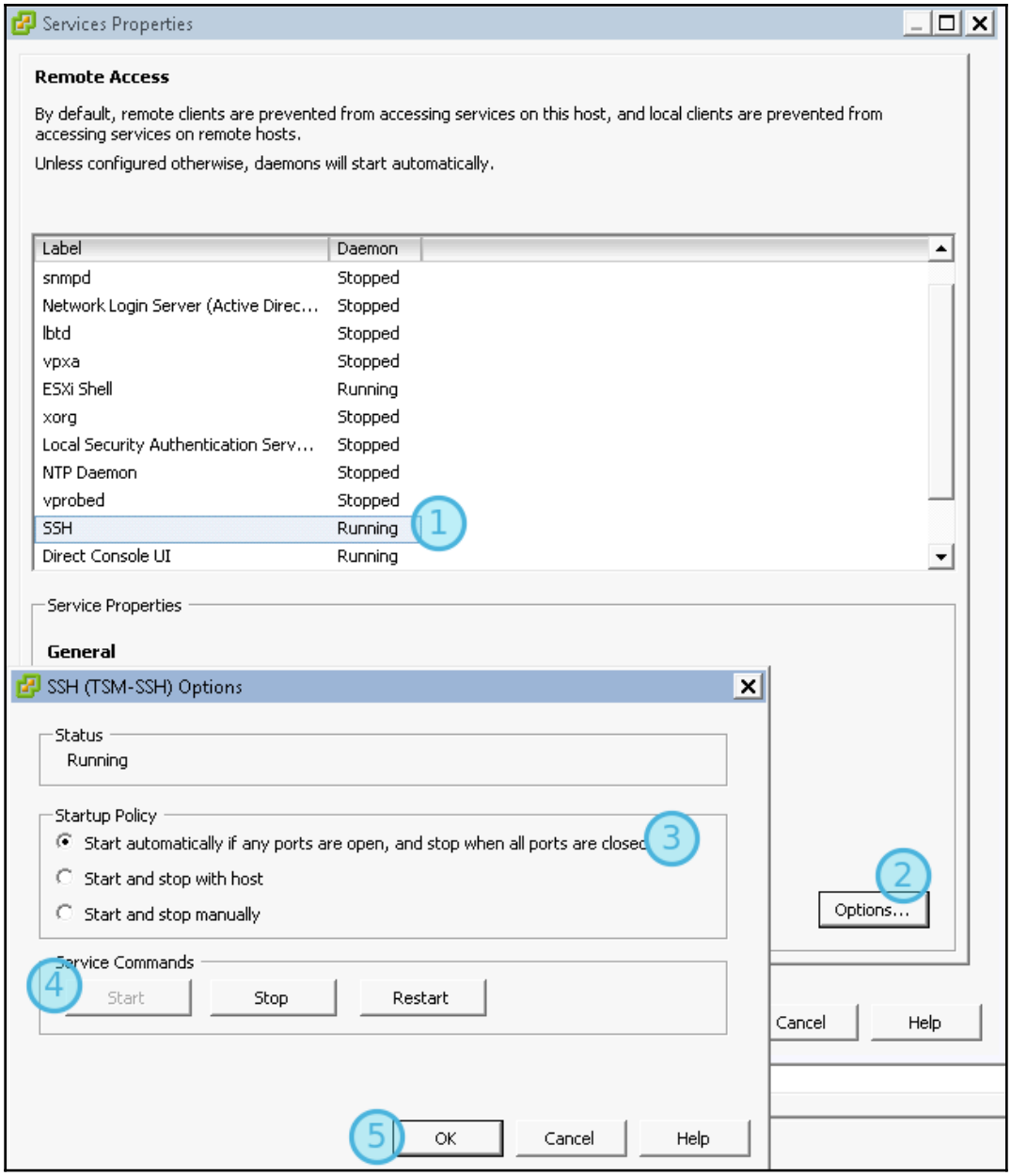
1. The first thing is to enable both Shell and SSH access to the ESXi console. Basically, ESXi allows you to manage it using the vSphere client (based on C# for the versions before 5.5.x and based on HTML for version 6 and up). Once we enable the Shell and SSH access, this will give us the ability to use the CLI to manage virtual infrastructure and to perform tasks such as creating, deleting, and customizing the virtual machine.

2. Access the ESXi vSphere client and go to **Configuration**, then choose **Security Profiles** from the left tab, and finally click on **Properties**:

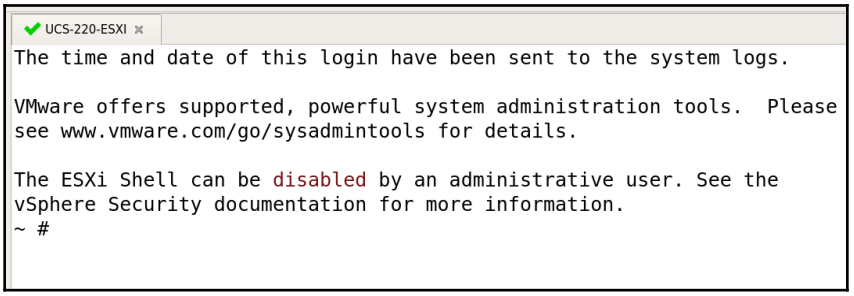


A pop-up window will be opened that contains a list of services, statuses, and various options that can be applied:

3. Select **SSH** service and then click on **Option**. Another pop-up window will be opened.
4. Choose the first option that reads **Start automatically if any ports are open, and stop when all ports are closed** under the **Startup Policy**.
5. Also, click on **Start** under **Service Commands** and hit **OK**:


Services Properties
Remote Access
By default, remote clients are prevented from accessing services on this host, and local clients are prevented from accessing services on remote hosts.
Unless configured otherwise, daemons will start automatically.
Label
Daemon
snmpd Stopped
Network Login Server (Active Direc... Stopped
lbtd Stopped
vpxa Stopped
ESXi Shell Running
xorg Stopped
Local Security Authentication Serv... Stopped
NTP Daemon Stopped
vprobed Stopped
SSH Running
Direct Console UI Running
1
Service Properties
General
Options...
2
SSH (TSM-SSH) Options
Status
Running
Startup Policy
Start automatically if any ports are open, and stop when all ports are closed
Start and stop with host
Start and stop manually
3
Service Commands
4
Start
Stop
Restart
Cancel
Help
5
OK
Cancel
Help

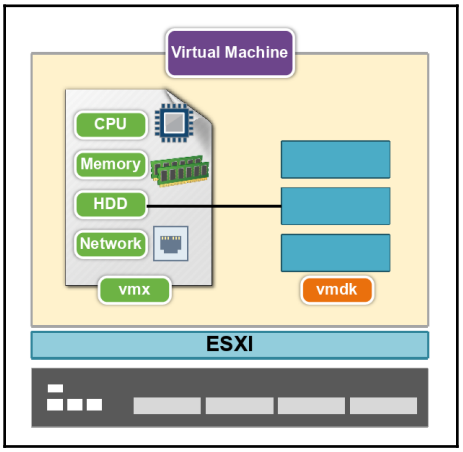
Repeat the same steps again for the **ESXi Shell** service. This will ensure that both services will be started once the ESXi server has started and will be opened and ready to accept the connection. You can test both services, SSH to the ESXi IP address and provide the root credentials as with SSH connection:



Generating a VMX file using Jinja2

The basic unit for a virtual machine (sometimes called a guest machine) is the VMX file. This file contains all the settings needed to build the virtual machine in terms of compute resources, allocated memory, HDD, and networking. Also, it defines the operating system that runs over the machine so the VMware can install some tools to manage the VM powering.

An additional file is needed: VMDK. This file stores the actual contents of the VM and acts as the hard disk for the VM partitions:



These files (VMX and VMDK) should be stored under the `/vmfs/volumes/datastore1` directory in the ESXi Shell and should be inside a directory with the name of the virtual machine.

Building the VMX template

We are now going to create the template file that we will use to build our virtual machine in Python. Here's an example of the final running VMX file that we need to generate with the help of Python and Jinja2:

```
.encoding = "UTF-8"
vhv.enable = "TRUE"
config.version = "8"
virtualHW.version = "8"

vmci0.present = "TRUE"
hpet0.present = "TRUE"
displayName = "test_jinja2"

# Specs
memSize = "4096"
numvcpus = "1"
cpuid.coresPerSocket = "1"


# HDD
scsi0.present = "TRUE"
scsi0.virtualDev = "lsilogic"
scsi0:0.deviceType = "scsi-hardDisk"
scsi0:0.fileName = "test_jinja2.vmdk"
scsi0:0.present = "TRUE"

# Floppy
floppy0.present = "false"

#  CDRom
ide1:0.present = "TRUE"
ide1:0.deviceType = "cdrom-image"
ide1:0.fileName = "/vmfs/volumes/datastore1/ISO Room/CentOS-7-x86_64-
Minimal-1708.iso"

#  Networking
ethernet0.virtualDev = "e1000"
ethernet0.networkName = "network1"
ethernet0.addressType = "generated"
```

```
ethernet0.present = "TRUE"

# VM Type
guestOS = "ubuntu-64"

# VMware Tools
toolScripts.afterPowerOn = "TRUE"
toolScripts.afterResume = "TRUE"
toolScripts.beforeSuspend = "TRUE"
toolScripts.beforePowerOff = "TRUE"
tools.remindInstall = "TRUE"
tools.syncTime = "FALSE"
```

I added some comments inside the file to illustrate the functionality of each block. However, in the actual file, you won't see these comments.

Let's analyze the file and understand the meaning of some fields:

- `vhv.enable`: When set to `True`, the ESXi server will expose the CPU host flags to the guest CPU that allows the running of the VM inside the guest machine (called nested virtualization).
- `displayName`: The name that will be registered in the ESXi and shown in the vSphere client.
- `memsize`: This defines the allocated RAM to the VM and should be provided in megabytes.
- `numvcpus`: This defines the number of physical CPUs allocated to the VM. This flag is used with `cpuid.coresPerSocket` so it can define the total number of vCPU allocated.
- `scsi0.virtualDev`: The type of SCSI controller for the virtual hard drive. It can be one of four values: BusLogic, LSI Logic parallel, LSI Logic SAS, or VMware paravirtual.
- `scsi0:0.fileName`: This defines the name of the `vmdk` (in the same directory) that will store the actual virtual machine settings.
- `ide1:0.fileName`: The image path that contains the installation binaries packaged in ISO format. This will make the ESXi connect the ISO image in the image CD-ROM (IDE device).
- `ethernet0.networkName`: This is the name of the virtual switch in ESXi that should connect to VM NIC. You can add additional instances of this parameter to reflect additional network interfaces.

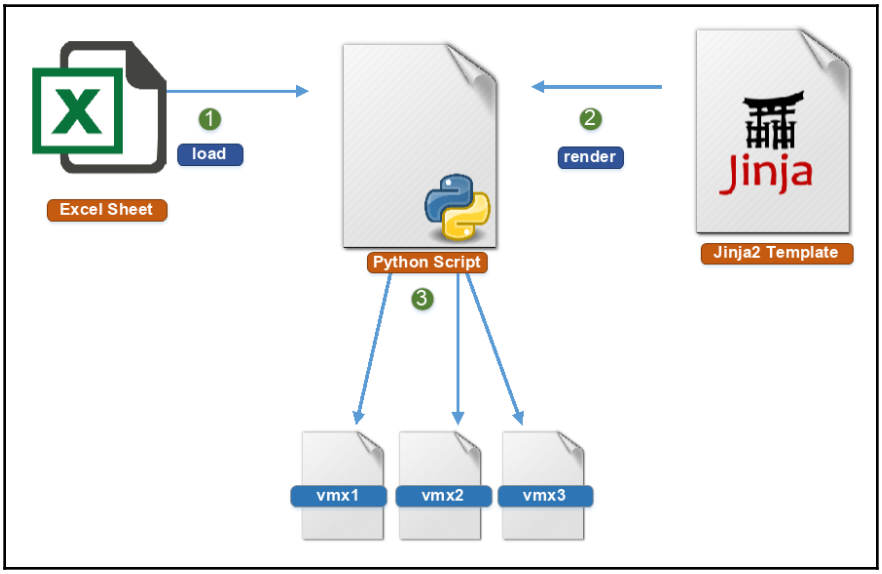
Now we will build the Jinja2 template; you can review `Chapter 6`, *Configuration Generator with Python and Jinja2,* for the basics of templating using the Jinja2 language:

```
.encoding = "UTF-8"
vhv.enable = "TRUE"
config.version = "8"
virtualHW.version = "8"

vmci0.present = "TRUE"
hpet0.present = "TRUE"
displayName = "{{vm_name}}"

# Specs
memSize = "{{ vm_memory_size }}"
numvcpus = "{{ vm_cpu }}"
cpuid.coresPerSocket = "{{cpu_per_socket}}"


# HDD
scsi0.present = "TRUE"
scsi0.virtualDev = "lsilogic"
scsi0:0.deviceType = "scsi-hardDisk"
scsi0:0.fileName = "{{vm_name}}.vmdk"
scsi0:0.present = "TRUE"

# Floppy
floppy0.present = "false"


# CDRom
ide1:0.present = "TRUE"
ide1:0.deviceType = "cdrom-image"
ide1:0.fileName = "/vmfs/volumes/datastore1/ISO Room/{{vm_image}}"


# Networking
ethernet0.virtualDev = "e1000"
ethernet0.networkName = "{{vm_network1}}"
ethernet0.addressType = "generated"
ethernet0.present = "TRUE"

# VM Type
guestOS = "{{vm_guest_os}}" #centos-64 or ubuntu-64

# VMware Tools
toolScripts.afterPowerOn = "TRUE"
toolScripts.afterResume = "TRUE"
```

```
toolScripts.beforeSuspend = "TRUE"
toolScripts.beforePowerOff = "TRUE"
tools.remindInstall = "TRUE"
tools.syncTime = "FALSE"
```

Notice that we removed the static values for the relevant fields, such as `diplayName`, `memsize`, and so on, and replaced them with double curly braces with variable names inside them. During template rendering from Python, these fields will be replaced with actual values to construct a valid VMX file.

Now, let's build the Python script that will render the file. Usually, we use the YAML data serialization in conjunction with Jinja2 to fill in the data of the template. But since we already explain the YAML concept in `Chapter 6`, *Configuration Generator with Python and Jinja2*, we will get our data from another data source, Microsoft Excel:



Handling Microsoft Excel data

Python has some excellent libraries that can handle the data written in an Excel sheet. We already used the Excel sheet in `Chapter 4`, *Using Python to Manage Network Devices*, when we needed to automate the `netmiko` configuration and read the data that described the infrastructure of the Excel file. Now, we will start by installing the Python `xlrd` library inside the Automation Server.

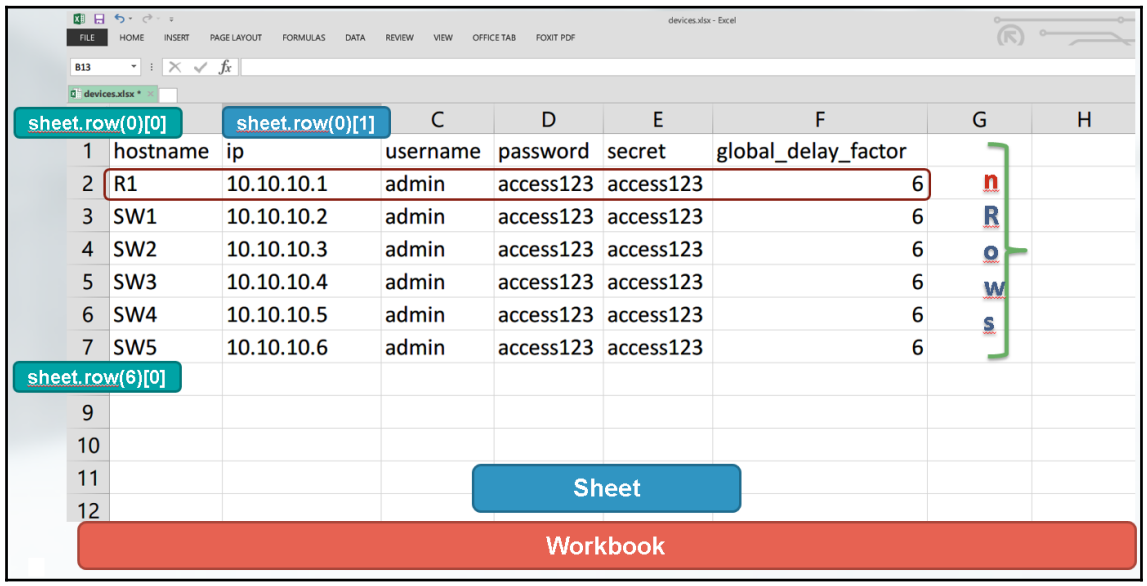
Use the following command to install `xlrd`:

```
pip install xlrd
```

```
[root@AutomationServer ~]# pip install xlrd
Collecting xlrd
  Downloading xlrd-1.1.0-py2.py3-none-any.whl (108kB)
    100% |████████████████████████████████| 112kB 750kB/s
Installing collected packages: xlrd
Successfully installed xlrd-1.1.0
[root@AutomationServer ~]#
```

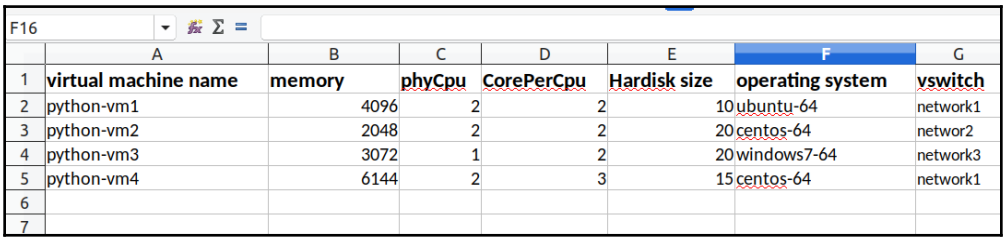
Follow the steps given below:

1. The XLRD module can open the Microsoft workbook and parse the contents using the `open_workbook()` method.
2. Then you can select the sheet that contains your data either by providing the sheet index or the sheet name to the `sheet_by_index()` or `sheet_by_name()` methods respectively.
3. Finally, you can access the row data by providing the row number to the `row()` function which converts the row data into a Python list:



Notice that `nrows` and `ncols` are special variables which will be populated once you open the sheet that counts the number of rows and number of columns inside the sheet. You can iterate over with the `for` loop. The number always start from

Back to the virtual machine example. We will have the following data in the Excel sheet, which reflects the virtual machine settings:



	A	B	C	D	E	F	G
1	**virtual machine name**	**memory**	**phyCpu**	**CorePerCpu**	**Hardisk size**	**operating system**	**vswitch**
2	python-vm1	4096	2	2	10	ubuntu-64	network1
3	python-vm2	2048	2	2	20	centos-64	networ2
4	python-vm3	3072	1	2	20	windows7-64	network3
5	python-vm4	6144	2	3	15	centos-64	network1
6							
7							

To read the data into Python, we will use the following script:

```
import xlrd
workbook =
xlrd.open_workbook(r"/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomat
ionProject/Chapter14_Creating_and_managing_VMware_virtual_machines/vm_inven
tory.xlsx")
sheet = workbook.sheet_by_index(0)
print(sheet.nrows)
print(sheet.ncols)

print(int(sheet.row(1)[1].value))

for row in range(1,sheet.nrows):
    vm_name = sheet.row(row)[0].value
    vm_memory_size = int(sheet.row(row)[1].value)
    vm_cpu = int(sheet.row(row)[2].value)
    cpu_per_socket = int(sheet.row(row)[3].value)
    vm_hdd_size = int(sheet.row(row)[4].value)
    vm_guest_os = sheet.row(row)[5].value
    vm_network1 = sheet.row(row)[6].value
```

In the previous script, we did the following:

1. We imported the `xlrd` module and provided the Excel file to the `open_workbook()` method to read the Excel sheet and save that to the `workbook` variable.
2. Then, we accessed the first sheet using the `sheet_by_index()` method and saved the reference to the `sheet` variable.

3. Now we will iterate over the opened sheet and get each field using the `row()` method. This will allow us to convert the row to a Python list. Since we need only one value inside the row, we will use the list slice to access the index. Remember that the list index always starts with zero. We will store that value into the variable and we will use this variable to populate the Jinja2 template in the next section.

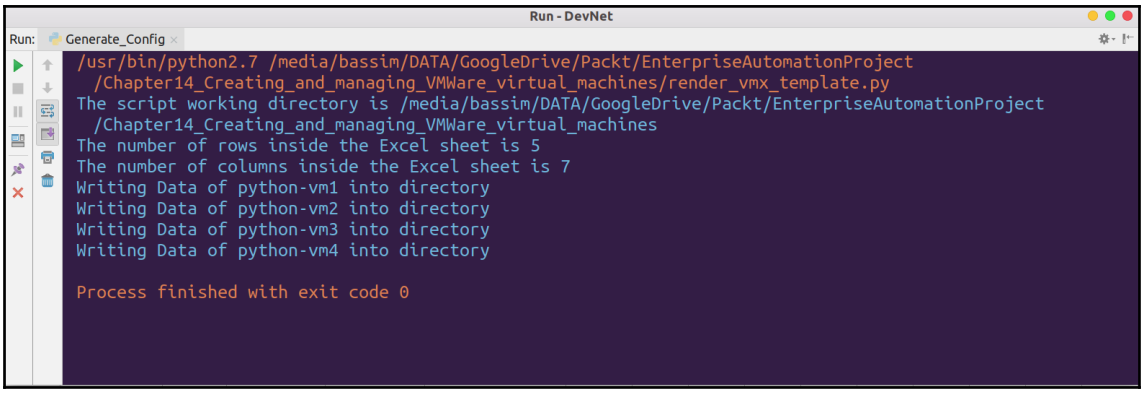
Generating VMX files

The last part is to generate the VMX files from the Jinja2 template. We will read the data from the Excel sheet and add it to the empty dictionary, `vmx_data`. This dictionary will be passed later to the `render()` function inside the Jinja2 template. The Python dictionary key will be the template variable name while the value will be the substituted values that should be in the file. The final part in the script is to open a file in writing mode inside the `vmx_files` directory and write the data into it for each VMX file:

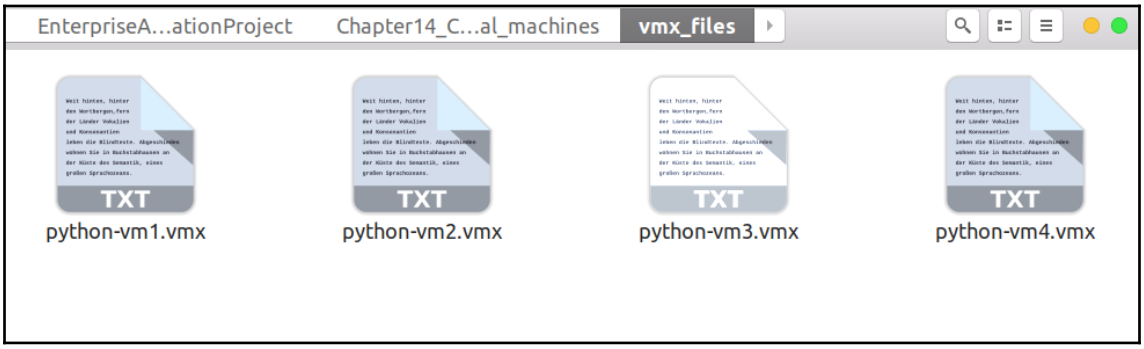
```
from jinja2 import FileSystemLoader, Environment
import os
import xlrd

print("The script working directory is {}"
.format(os.path.dirname(__file__)))
script_dir = os.path.dirname(__file__)

vmx_env = Environment(
    loader=FileSystemLoader(script_dir),
    trim_blocks=True,
    lstrip_blocks= True
)


workbook = xlrd.open_workbook(os.path.join(script_dir,"vm_inventory.xlsx"))
sheet = workbook.sheet_by_index(0)
print("The number of rows inside the Excel sheet is {}"
.format(sheet.nrows))
print("The number of columns inside the Excel sheet is {}"
.format(sheet.ncols))


vmx_data = {}

for row in range(1,sheet.nrows):
    vm_name = sheet.row(row)[0].value
```

```
    vm_memory_size = int(sheet.row(row)[1].value)
    vm_cpu = int(sheet.row(row)[2].value)
    cpu_per_socket = int(sheet.row(row)[3].value)
    vm_hdd_size = int(sheet.row(row)[4].value)
    vm_guest_os = sheet.row(row)[5].value
    vm_network1 = sheet.row(row)[6].value

    vmx_data["vm_name"] = vm_name
    vmx_data["vm_memory_size"] = vm_memory_size
    vmx_data["vm_cpu"] = vm_cpu
    vmx_data["cpu_per_socket"] = cpu_per_socket
    vmx_data["vm_hdd_size"] = vm_hdd_size
    vmx_data["vm_guest_os"] = vm_guest_os
    if vm_guest_os == "ubuntu-64":
        vmx_data["vm_image"] = "ubuntu-16.04.4-server-amd64.iso"

    elif vm_guest_os == "centos-64":
        vmx_data["vm_image"] = "CentOS-7-x86_64-Minimal-1708.iso"

    elif vm_guest_os == "windows7-64":
        vmx_data["vm_image"] = "windows_7_ultimate_sp1_ x86-x64_bg-en_IE10_
April_2013.iso"

    vmx_data["vm_network1"] = vm_network1

    vmx_data = vmx_env.get_template("vmx_template.j2").render(vmx_data)
    with open(os.path.join(script_dir,"vmx_files/{}.vmx".format(vm_name)),
"w") as f:
        print("Writing Data of {} into directory".format(vm_name))
        f.write(vmx_data)
    vmx_data = {}
```

The script output is as follows:



The files are stored under `vmx_files` and each one contains specific information for the virtual machine as configured in the excel sheet:



Now, we will use both `paramiko` and `scp` libraries to connect to the ESXi Shell and upload these files under `/vmfs/volumes/datastore1`. To achieve that, we will first create a function named `upload_and_create_directory()` that accepts `vm name`, `hard disk size`, and VMX `source file`. `paramiko` will connect to the ESXi server and execute the required commands which will create both the directory and VMDK under `/vmfs/volumes/datastore1`. Finally, we will use `SCPClient` from the `scp` module to upload the source files to the previously created directory and run the registry command to add the machine to the vSphere client:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import paramiko
from scp import SCPClient
import time

def upload_and_create_directory(vm_name, hdd_size, source_file):

    commands = ["mkdir /vmfs/volumes/datastore1/{0}".format(vm_name),
                "vmkfstools -c {0}g -a lsilogic -d zeroedthick
/vmfs/volumes/datastore1/{1}/{1}.vmdk".format(hdd_size,
vm_name),]
    register_command = "vim-cmd solo/registervm
/vmfs/volumes/datastore1/{0}/{0}.vmx".format(vm_name)
    ipaddr = "10.10.10.115"
    username = "root"
    password = "access123"
```

```
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    ssh.connect(ipaddr, username=username, password=password,
look_for_keys=False, allow_agent=False)

    for cmd in commands:
        try:
            stdin, stdout, stderr = ssh.exec_command(cmd)
            print " DEBUG: ... Executing the command on ESXi
server".format(str(stdout.readlines()))

        except Exception as e:
            print e
            pass
            print " DEBUG: **ERR....unable to execute command"
        time.sleep(2)
    with SCPClient(ssh.get_transport()) as scp:
        scp.put(source_file,
remote_path='/vmfs/volumes/datastore1/{0}'.format(vm_name))
        ssh.exec_command(register_command)
    ssh.close()
```

We need to define this function *before* we run theJinja2 template and generate the VMX and call the function *after* we save the file to the `vmx_files` directory and pass the required arguments to it.

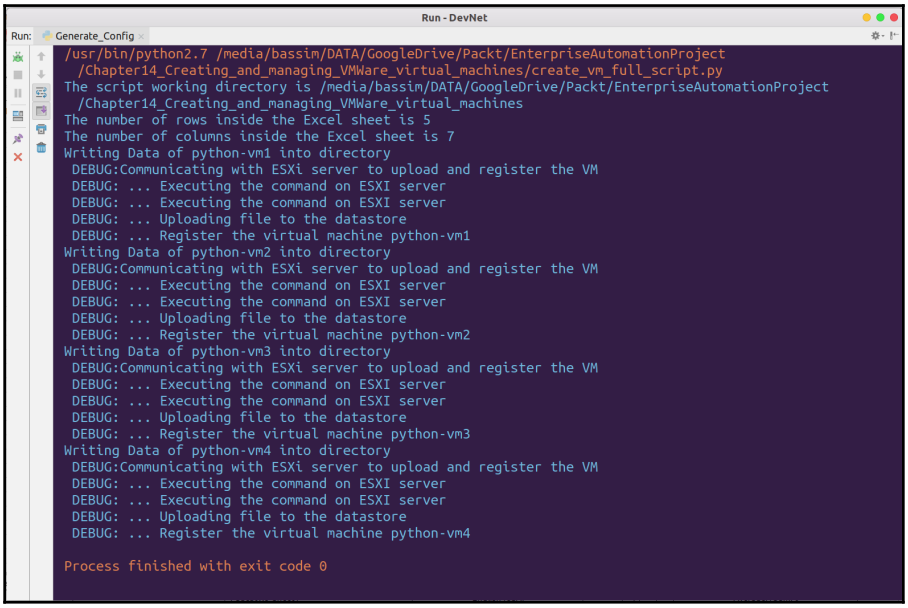
The final code should be as follows:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import paramiko
from scp import SCPClient
import time
from jinja2 import FileSystemLoader, Environment
import os
import xlrd

def upload_and_create_directory(vm_name, hdd_size, source_file):

    commands = ["mkdir /vmfs/volumes/datastore1/{0}".format(vm_name),
                "vmkfstools -c {0}g -a lsilogic -d zeroedthick
/vmfs/volumes/datastore1/{1}/{1}.vmdk".format(hdd_size,
vm_name),]
    register_command = "vim-cmd solo/registervm
```

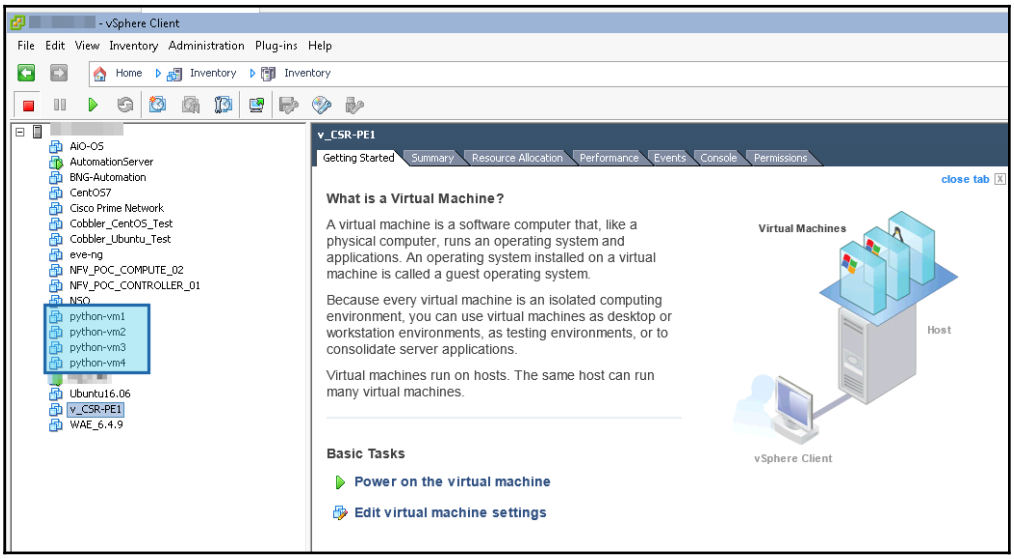
```
/vmfs/volumes/datastore1/{0}/{0}.vmx".format(vm_name)

    ipaddr = "10.10.10.115"
    username = "root"
    password = "access123"

    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    ssh.connect(ipaddr, username=username, password=password,
look_for_keys=False, allow_agent=False)

    for cmd in commands:
        try:
            stdin, stdout, stderr = ssh.exec_command(cmd)
            print " DEBUG: ... Executing the command on ESXi
server".format(str(stdout.readlines()))

        except Exception as e:
            print e
            pass
            print " DEBUG: **ERR....unable to execute command"
        time.sleep(2)
    with SCPClient(ssh.get_transport()) as scp:
        print(" DEBUG: ... Uploading file to the datastore")
        scp.put(source_file,
remote_path='/vmfs/volumes/datastore1/{0}'.format(vm_name))
        print(" DEBUG: ... Register the virtual machine
{}".format(vm_name))
        ssh.exec_command(register_command)

    ssh.close()

print("The script working directory is {}"
.format(os.path.dirname(__file__)))
script_dir = os.path.dirname(__file__)

vmx_env = Environment(
    loader=FileSystemLoader(script_dir),
    trim_blocks=True,
    lstrip_blocks= True
)

workbook = xlrd.open_workbook(os.path.join(script_dir,"vm_inventory.xlsx"))
sheet = workbook.sheet_by_index(0)
print("The number of rows inside the Excel sheet is {}"
.format(sheet.nrows))
print("The number of columns inside the Excel sheet is {}"
```

```
.format(sheet.ncols))


vmx_data = {}

for row in range(1,sheet.nrows):
    vm_name = sheet.row(row)[0].value
    vm_memory_size = int(sheet.row(row)[1].value)
    vm_cpu = int(sheet.row(row)[2].value)
    cpu_per_socket = int(sheet.row(row)[3].value)
    vm_hdd_size = int(sheet.row(row)[4].value)
    vm_guest_os = sheet.row(row)[5].value
    vm_network1 = sheet.row(row)[6].value

    vmx_data["vm_name"] = vm_name
    vmx_data["vm_memory_size"] = vm_memory_size
    vmx_data["vm_cpu"] = vm_cpu
    vmx_data["cpu_per_socket"] = cpu_per_socket
    vmx_data["vm_hdd_size"] = vm_hdd_size
    vmx_data["vm_guest_os"] = vm_guest_os
    if vm_guest_os == "ubuntu-64":
        vmx_data["vm_image"] = "ubuntu-16.04.4-server-amd64.iso"

    elif vm_guest_os == "centos-64":
        vmx_data["vm_image"] = "CentOS-7-x86_64-Minimal-1708.iso"

    elif vm_guest_os == "windows7-64":
        vmx_data["vm_image"] = "windows_7_ultimate_sp1_ x86-x64_bg-en_IE10_
April_2013.iso"

    vmx_data["vm_network1"] = vm_network1

    vmx_data = vmx_env.get_template("vmx_template.j2").render(vmx_data)
    with open(os.path.join(script_dir,"vmx_files/{}.vmx".format(vm_name)),
"w") as f:
        print("Writing Data of {} into directory".format(vm_name))
        f.write(vmx_data)
        print(" DEBUG:Communicating with ESXi server to upload and register
the VM")
    upload_and_create_directory(vm_name,
                                vm_hdd_size,
os.path.join(script_dir,"vmx_files","{}.vmx".format(vm_name)))
    vmx_data = {}
```

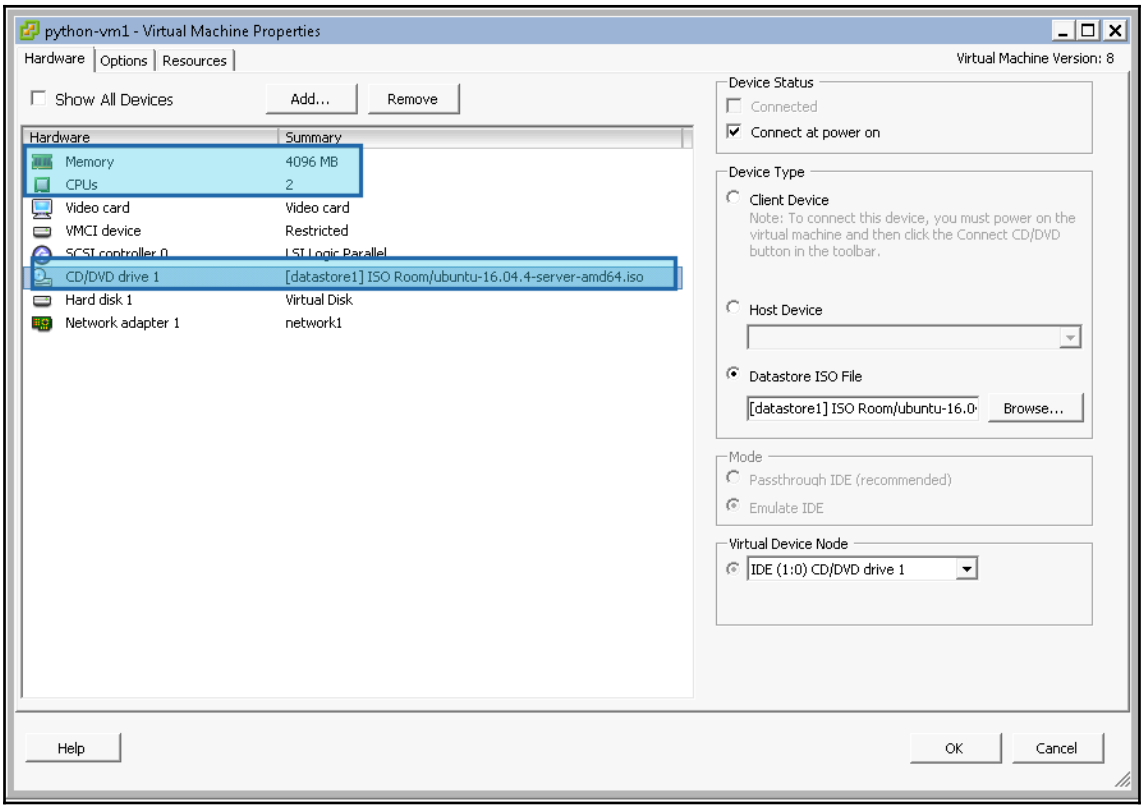
The script output is as follows:



If you check the vSphere client after you run the script, you will find four machines have been created with the name provided in the Excel sheet:



Also, you will find the virtual machine customized with settings such as **CPUs**, **Memory**, and connected ISO room:



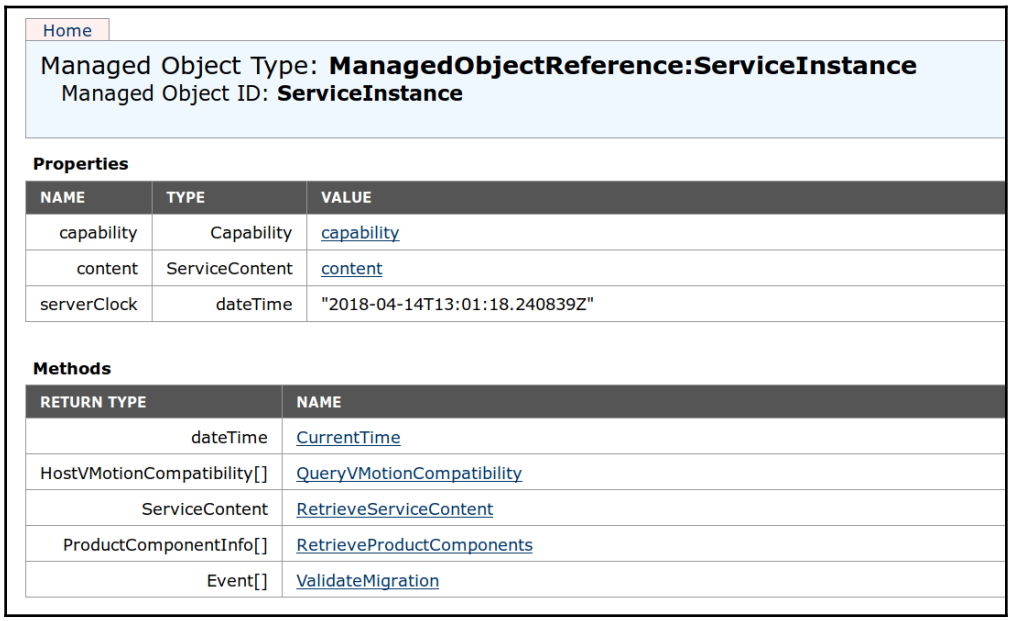
You can complete your automation workflow in VMware by connecting the created virtual machine to Cobbler. We covered it in `Chapter 8`, *Preparing the System Lab Environment*. Cobbler will automate the operating system installation and customization either Windows, CentOS, or Ubuntu. After that, you can use Ansible, which we covered in `Chapter 13`, *Ansible for System Administration*, to prepare the system in terms of security, configuration, and installed packages, then deploy your application after that. This is a full-stack automation that covers things such as virtual machine creation and getting your application up and running.

VMware Python clients

VMware products (ESXi and vCenter, which used to manage ESXi) support receiving external API requests through the web service. You can execute the same administration tasks you do on the vSphere client, such as creating a new virtual machine, creating a new vSwitch, or even controlling the `vm` status, but this time through the supported API that has bindings for many languages, such as Python, Ruby, and Go.

vSphere has a special model for the inventory and everything inside it is an object with specific values. You can access this model and see the actual values for your infrastructure through the **Managed Object Browser** (**MoB**) which gives you access to all object details. We will use the official Python bindings from VMware (`pyvmomi`) to interact with this model and alter the values (or create them) inside the inventory.

It's worth noting that the MoB can be accessed through the web browser by going to `http://<ESXi_server_ip_or_domain>/mob`,which will ask you to provide the root username and password:



Home

Managed Object Type: **ManagedObjectReference:ServiceInstance**
Managed Object ID: **ServiceInstance**

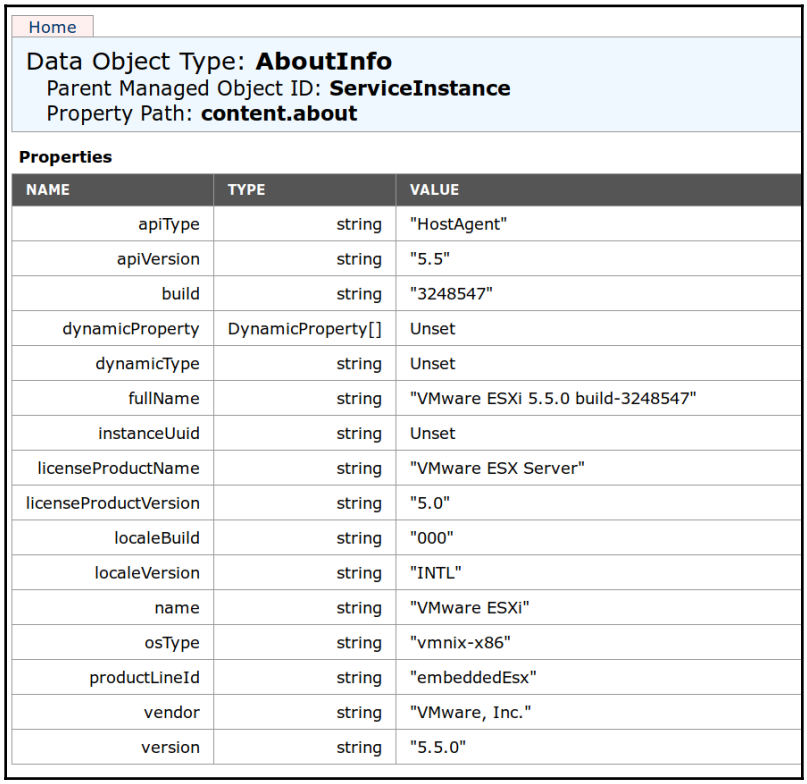
Properties

NAME	TYPE	VALUE
capability	Capability	capability
content	ServiceContent	content
serverClock	dateTime	"2018-04-14T13:01:18.240839Z"

Methods

RETURN TYPE	NAME
dateTime	CurrentTime
HostVMotionCompatibility[]	QueryVMotionCompatibility
ServiceContent	RetrieveServiceContent
ProductComponentInfo[]	RetrieveProductComponents
Event[]	ValidateMigration

You can click on any of the hyperlinks to see more details and access each *leaf* inside each tree or context. For example, click on **Content.about** to see full details about your server such as the exact version, build, and full name:



Home

Data Object Type: **AboutInfo**
Parent Managed Object ID: **ServiceInstance**
Property Path: **content.about**

Properties

NAME	TYPE	VALUE
apiType	string	"HostAgent"
apiVersion	string	"5.5"
build	string	"3248547"
dynamicProperty	DynamicProperty[]	Unset
dynamicType	string	Unset
fullName	string	"VMware ESXi 5.5.0 build-3248547"
instanceUuid	string	Unset
licenseProductName	string	"VMware ESX Server"
licenseProductVersion	string	"5.0"
localeBuild	string	"000"
localeVersion	string	"INTL"
name	string	"VMware ESXi"
osType	string	"vmnix-x86"
productLineId	string	"embeddedEsx"
vendor	string	"VMware, Inc."
version	string	"5.5.0"

Notice how the table is structured. The first column contains the property name, the second column is the data type of that property, and, finally, the third column is the actual running value.

Installing PyVmomi

PyVmomi is available to download either though Python `pip` or as a system package from different repos.

For Python installation, use the following command:

```
pip install -U pyvmomi
```

```
[root@AutomationServer ~]#  pip install pyvmomi
Collecting pyvmomi
  Downloading pyvmomi-6.5.0.2017.5-1.tar.gz (252kB)
    100% |████████████████████████████████| 256kB 1.3MB/s
Requirement already satisfied: requests>=2.3.0 in /usr/lib/python2.7/site-packages (from pyvm
omi)
Requirement already satisfied: six>=1.7.3 in /usr/lib/python2.7/site-packages (from pyvmomi)
Requirement already satisfied: certifi>=2017.4.17 in /usr/lib/python2.7/site-packages (from r
equests>=2.3.0->pyvmomi)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/lib/python2.7/site-packages (fro
m requests>=2.3.0->pyvmomi)
Requirement already satisfied: idna<2.7,>=2.5 in /usr/lib/python2.7/site-packages (from reque
sts>=2.3.0->pyvmomi)
Requirement already satisfied: urllib3<1.23,>=1.21.1 in /usr/lib/python2.7/site-packages (fro
m requests>=2.3.0->pyvmomi)
Building wheels for collected packages: pyvmomi
  Running setup.py bdist_wheel for pyvmomi ... done
  Stored in directory: /root/.cache/pip/wheels/5a/e2/d8/1a5692c5a3190b0dc406ea9613ad399943b2e
138462b21ae0c
Successfully built pyvmomi
Installing collected packages: pyvmomi
Successfully installed pyvmomi-6.5.0.2017.5-1
[root@AutomationServer ~]#
```

Notice the version downloaded from `pip` is `6.5.2017.5-1`, which correlates with the vSphere release VMware vSphere 6.5, but this doesn't mean it won't work with older releases of ESXi. For example, I have VMware vSphere 5.5, which works flawlessly with the latest `pyvmomi` version.

For system installation:

```
yum install pyvmomi -y
```

The Pyvmomi library uses dynamic types which means features such as Intelli-Sense and autocomplete features in IDE do not work with it. You have to rely on documentation and MoB to discover what classes or methods are needed to get the job done but, once you discover the way it works, it will be pretty easy to work with.

First steps with pyvmomi

The first thing is you need to do is connect to ESXi MoB by providing the username, password, and host IP, and start to navigate to the MoB to get the required data. This can be done by using the `SmartConnectNoSSL()` method:

```
from pyVim.connect import SmartConnect, Disconnect,SmartConnectNoSSL
ESXi_connection = SmartConnectNoSSL(host="10.10.10.115", user="root",
pwd='access123')
```

Note that there's another method called `SmartConnect()` and you must provide the SSL context to it when establishing a connection, otherwise the connection will fail. However, you can use the following code snippet to request that the SSL does not verify the certificate and to pass this context to `SmartConnect()` in the `sslCContext` argument:

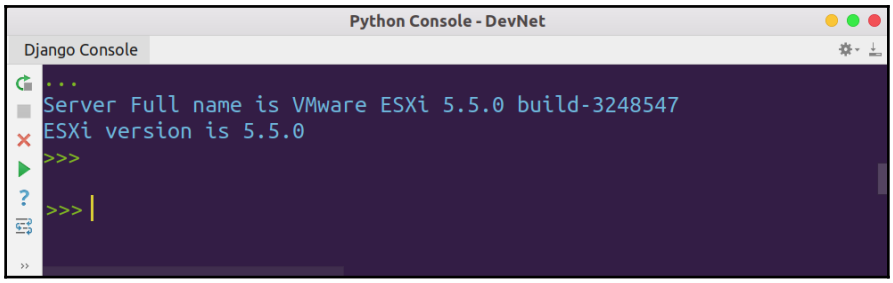
```
import ssl
import requests
certificate = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
certificate.verify_mode = ssl.CERT_NONE
requests.packages.urllib3.disable_warnings()
```

For the sake of beverity and to keep our code short, we will use the built-in `SmartConnectNoSSL()`.

Next, we will start exploring the MoB and get the full name and version of our server in the `about` object. Remember, it's located under the `content` object, so we need to access that too:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from pyVim.connect import SmartConnect, Disconnect,SmartConnectNoSSL
ESXi_connection = SmartConnectNoSSL(host="10.10.10.115", user="root",
pwd='access123')

full_name = ESXi_connection.content.about.fullName
version = ESXi_connection.content.about.version
print("Server Full name is {}".format(full_name))
print("ESXi version is {}".format(version))
Disconnect(ESXi_connection)
```

The output is as follows:



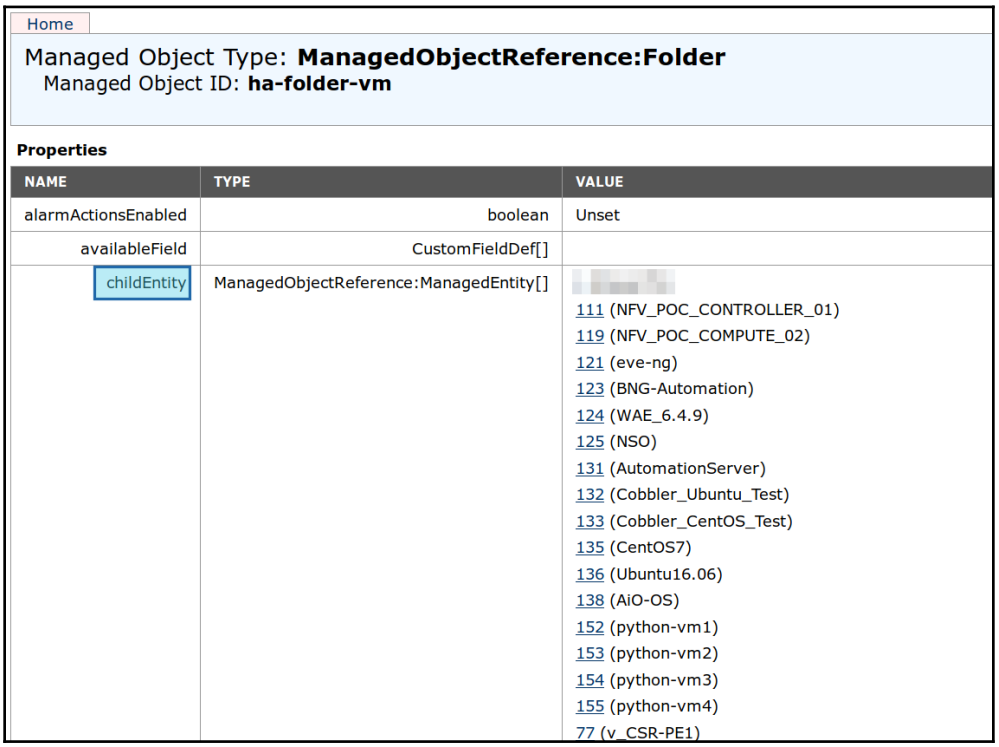
Great. Now we understand how the API works. Let's get into some serious scripts and retrieve some details about the deployed virtual machine in our ESXi.

The script is as follows:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from pyVim.connect import SmartConnect, Disconnect,SmartConnectNoSSL

ESXi_connection = SmartConnectNoSSL(host="10.10.10.115", user="root",
pwd='access123')

datacenter = ESXi_connection.content.rootFolder.childEntity[0] #First
Datacenter in the ESXi\

virtual_machines = datacenter.vmFolder.childEntity #Access the child inside
the vmFolder

print virtual_machines

for machine in virtual_machines:
    print(machine.name)
    try:
        guest_vcpu = machine.summary.config.numCpu
        print("  The Guest vCPU is {}" .format(guest_vcpu))

        guest_os = machine.summary.config.guestFullName
        print("  The Guest Operating System is {}" .format(guest_os))

        guest_mem = machine.summary.config.memorySizeMB
        print("  The Guest Memory is {}" .format(guest_mem))

        ipadd = machine.summary.guest.ipAddress
        print("  The Guest IP Address is {}" .format(ipadd))
        print "================================="
    except:
        print("  Can't get the summary")
```

In the previous example, we did the following:

1. We established the API connection again to MoB by providing the ESXi/vCenter credentials to the `SmartConnectNoSSL` method.
2. Then, we accessed the data center object by accessing the `content` then `rootFolder` objects and finally `childEntity`. The returned object was an iterable so we accessed the first element (the first data center) since we had only one ESXi in the lab. You could iterate over all data centers to get a list of all virtual machines in all registered data centers.
3. The virtual machines can be accessed via the `vmFolder` and the `childEntity`. Again, remember the returned output is iteratable and represents the virtual machine list stored inside the `virtual_machines` variable:


Home

Managed Object Type: **ManagedObjectReference:Folder**
Managed Object ID: **ha-folder-vm**

Properties

NAME	TYPE	VALUE
alarmActionsEnabled	boolean	Unset
availableField	CustomFieldDef[]	
childEntity	ManagedObjectReference:ManagedEntity[]	[redacted] 111 (NFV_POC_CONTROLLER_01) 119 (NFV_POC_COMPUTE_02) 121 (eve-ng) 123 (BNG-Automation) 124 (WAE_6.4.9) 125 (NSO) 131 (AutomationServer) 132 (Cobbler_Ubuntu_Test) 133 (Cobbler_CentOS_Test) 135 (CentOS7) 136 (Ubuntu16.06) 138 (AiO-OS) 152 (python-vm1) 153 (python-vm2) 154 (python-vm3) 155 (python-vm4) 77 (v_CSR-PE1)

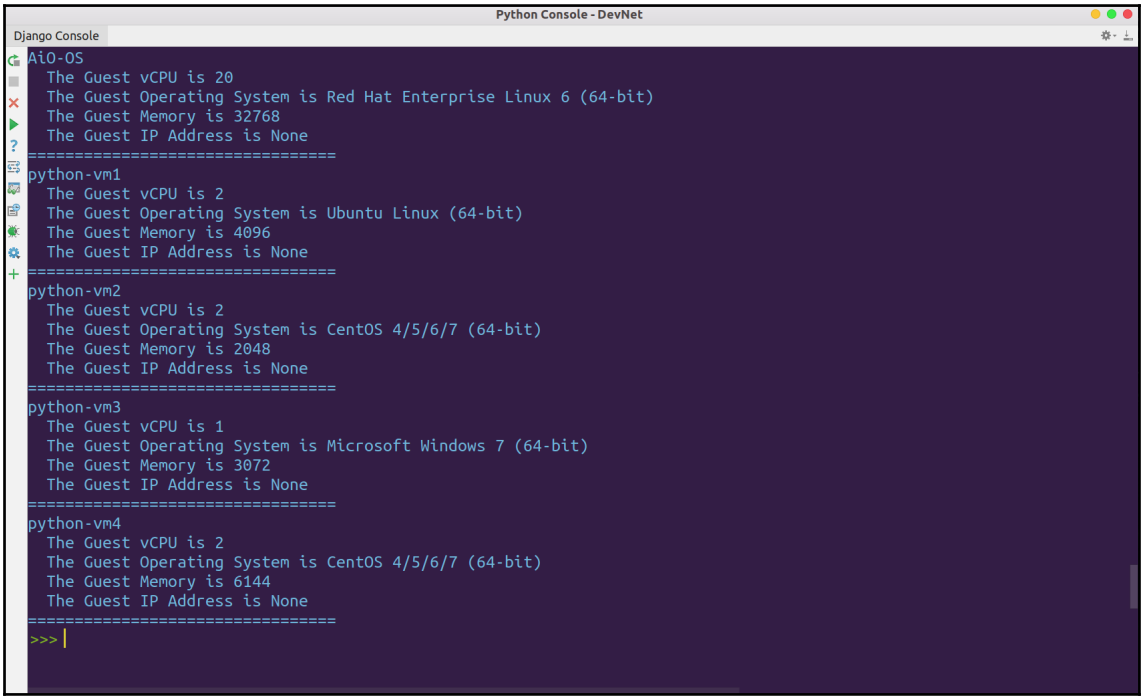
4. We iterated over the `virtual_machines` object and we query the CPU, Memory, Full name, and IP address of each element (for each virtual machine). These elements are located under each virtual machine tree in the `summary` and `config` leafs. Here is an example of our `AutomationServer` settings:

Data Object Type: **VirtualMachineConfigSummary**
Parent Managed Object ID: **131**
Property Path: **summary.config**

Properties

NAME	TYPE	VALUE
annotation	string	""
cpuReservation	int	0
dynamicProperty	DynamicProperty[]	Unset
dynamicType	string	Unset
ftInfo	FaultToleranceConfigInfo	Unset
guestFullName	string	"Ubuntu Linux (64-bit)"
guestId	string	"ubuntu64Guest"
installBootRequired	boolean	Unset
instanceUuid	string	"523b23be-7100-c891-959a-0d5b0b1f7cad"
managedBy	ManagedByInfo	Unset
memoryReservation	int	0
memorySizeMB	int	4096
name	string	"AutomationServer"
numCpu	int	1
numEthernetCards	int	2
numVirtualDisks	int	1
product	VAppProductInfo	Unset
template	boolean	false
uuid	string	"564de65b-1c66-be66-b11d-7472aa3428a6"
vmPathName	string	"[datastore1] AutomationServer/AutomationServer.vmx"

The script output is as follows:



Note that the `python-vm` machines that we created early at the beginning of the chapter are printed in the last screenshot. You can use PyVmomi as a validation tool that integrates with your automation workflow to validate whether machines are up and running and to make decisions based on the returned output.

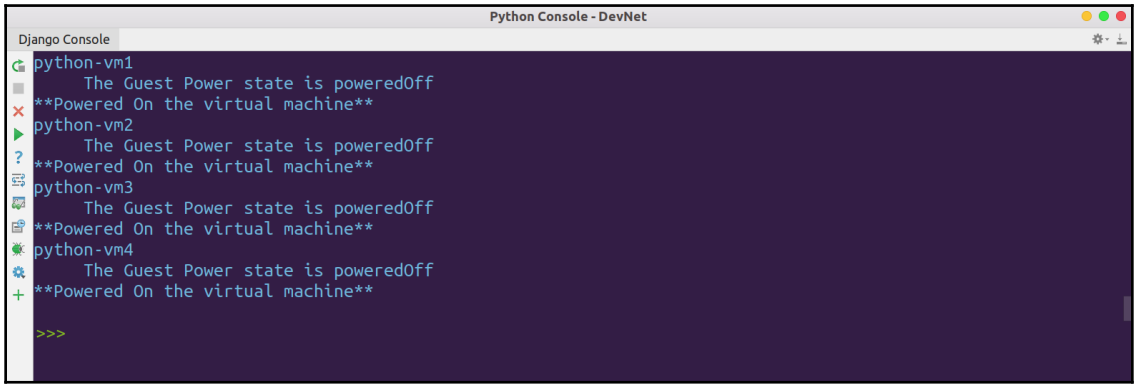
Changing the virtual machine state

This time we will use the `pyvmomi` bindings to change the virtual machine state. This will be done by checking the virtual machine name as we did before; then, we will navigate to another tree in MoB and get the runtime status. Finally, we will apply either the `PowerOn()` or `PowerOff()` function on the machine depending on its current status. This will switch the machine state from `On` to `Off` and vice versa.

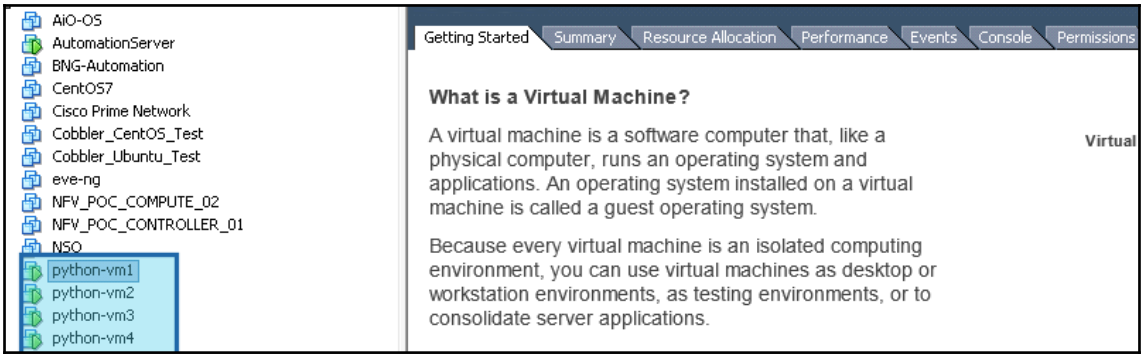
The script is as follows:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from pyVim.connect import SmartConnect, Disconnect,SmartConnectNoSSL

ESXi_connection = SmartConnectNoSSL(host="10.10.10.115", user="root",
pwd='access123')

datacenter = ESXi_connection.content.rootFolder.childEntity[0] #First
Datacenter in the ESXi\

virtual_machines = datacenter.vmFolder.childEntity #Access the child inside
the vmFolder

for machine in virtual_machines:
    try:
        powerstate = machine.summary.runtime.powerState
        if "python-vm" in machine.name and powerstate == "poweredOff":
            print(machine.name)
            print("     The Guest Power state is {}".format(powerstate))
            machine.PowerOn()
            print("**Powered On the virtual machine**")

        elif "python-vm" in machine.name and powerstate == "poweredOn":
            print(machine.name)
            print("     The Guest Power state is {}".format(powerstate))
            machine.PowerOff()
            print("**Powered Off the virtual machine**")
    except:
        print("  Can't execute the task")

Disconnect(ESXi_connection)
```

The script output is as follows:



Also, you can validate the virtual machine statue from the vSphere client and check the hosts that start with `python-vm*`, changing their power state from `poweredOff` to `poweredOn`:



There's more

You can find many useful scripts based on the `pyvmomi` bindings (in different languages) in the official VMware repository at GitHub (`https://github.com/vmware/pyvmomi-community-samples/tree/master/samples`). The scripts are provided by numerous contributors who use the tools and test them on a daily basis. Most of the scripts provide room to enter your configuration (such as ESXi IP address and credentials) without modifying the script source code by providing it as arguments.

Using Ansible playbook to manage instances

In the last part of VMware automation, we will utilize the Ansible tool to administrate the VMware infrastructure. Ansible ships with more than 20 VMware modules (`http://docs.ansible.com/ansible/latest/modules/list_of_cloud_modules.html#vmware`), which can execute many tasks such as managing data centers, clusters, and virtual machines. In older Ansible versions, Ansible used the `pysphere` module (which is not official; the author of the module has not maintained it since 2013) to automate the tasks. However, the newer version now supports the `pyvmomi` bindings.

Ansible also supports the VMware SDN product (NSX). Ansible Tower can be accessed from **VMware vRealize Automation** (**vRA**), which allows for complete workflow integration between different tools.

The following is the Ansible Playbook:

```
- name: Provision New VM
  hosts: localhost
  connection: local
  vars:
    - VM_NAME: DevOps
    - ESXi_HOST: 10.10.10.115
    - USERNAME: root
    - PASSWORD: access123
  tasks:
    - name: current time
      command: date +%D
      register: current_time
    - name: Check for vSphere access parameters
      fail: msg="Must set vsphere_login and vsphere_password in a Vault"
      when: (USERNAME is not defined) or (PASSWORD is not defined)
    - name: debug vCenter hostname
      debug: msg="vcenter_hostname = '{{ ESXi_HOST }}'"
    - name: debug the time
      debug: msg="Time is = '{{ current_time }}'"

    - name: "Provision the VM"
      vmware_guest:
        hostname: "{{ ESXi_HOST }}"
        username: "{{ USERNAME }}"
        password: "{{ PASSWORD }}"
        datacenter: ha-datacenter
```

```
          validate_certs: False
          name: "{{ VM_NAME }}"
          folder: /
          guest_id: centos64Guest
          state: poweredon
          force: yes
          disk:
            - size_gb: 100
              type: thin
              datastore: datastore1

          networks:
            - name: network1
              device_type: e1000
#              mac: ba:ba:ba:ba:01:02
#              wake_on_lan: True

            - name: network2
              device_type: e1000

          hardware:
            memory_mb: 4096
            num_cpus: 4
            num_cpu_cores_per_socket: 2
            hotadd_cpu: True
            hotremove_cpu: True
            hotadd_memory: True
            scsi: lsilogic
          cdrom:
            type: "iso"
            iso_path: "[datastore1] ISO Room/CentOS-7-x86_64-
Minimal-1708.iso"
        register: result
```

In the previous playbook, we can see the following:

- The first part of the playbook was to define the ESXi host IP and credentials in the `vars` section and to use them later in tasks.
- Then we wrote a simple validation to fail the playbook if the username or password was not provided.

- Then, we used the `vmware_guest` module provided by ansible (`https://docs.ansible.com/ansible/2.4/vmware_guest_module.html`) to provision the virtual machine. Inside this task, we provided the required information, such as disk size and hardware in term of CPU and memory. Notice that we defined the state of the virtual machine as `poweredon` so ansible will power on the virtual machine after creating it.
- Disks, networks, hardware, and CD-ROMs are all keys inside the `vmware_guest` module used to describe the virtualized hardware specs needed for spawning the new VM over the VMware ESXi.

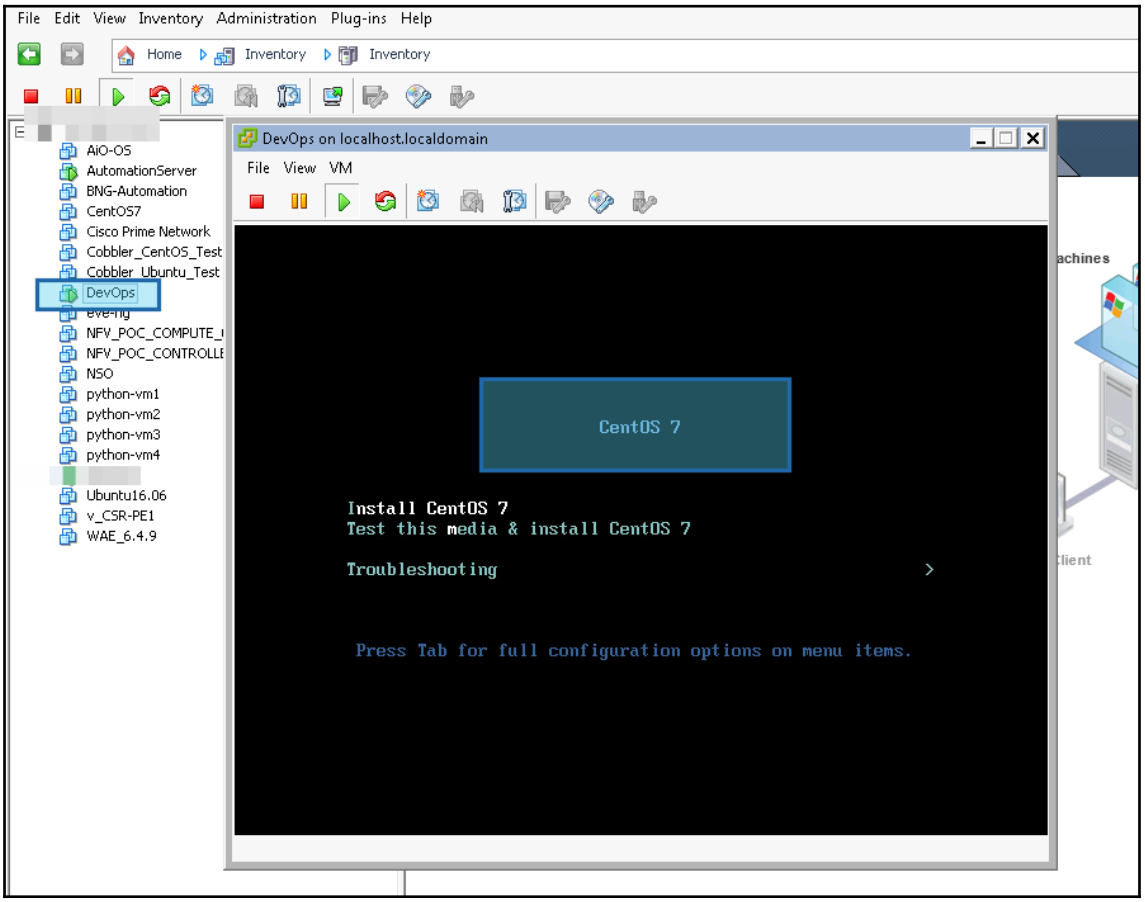
Run the playbook using the following command:

```
# ansible-playbook esxi_create_vm.yml -vv
```

The following is the screenshot of the Playbook output:

```
TASK [Provision the VM] ********************************************************************
task path: /root/esxi_create_vm.yml:26
changed: [localhost] => {"changed": true, "instance": {"annotation": "", "current_snapshot":
null, "customvalues": {}, "guest_consolidation_needed": false, "guest_question": null, "guest
_tools_status": "guestToolsNotRunning", "guest_tools_version": "0", "hw_cores_per_socket": 2,
 "hw_datastores": ["datastore1"], "hw_esxi_host": "localhost.localdomain", "hw_eth0": {"addre
sstype": "generated", "ipaddresses": null, "label": "Network adapter 1", "macaddress": "00:0c
:29:55:d5:3e", "macaddress_dash": "00-0c-29-55-d5-3e", "summary": "network1"}, "hw_eth1": {"a
ddresstype": "generated", "ipaddresses": null, "label": "Network adapter 2", "macaddress": "0
0:0c:29:55:d5:48", "macaddress_dash": "00-0c-29-55-d5-48", "summary": "network2"}, "hw_files"
: ["[datastore1] DevOps/DevOps.vmx", "[datastore1] DevOps/DevOps.vmxf", "[datastore1] DevOps/
DevOps.vmsd", "[datastore1] DevOps/DevOps.nvram", "[datastore1] DevOps/DevOps.vmdk"], "hw_fol
der": "/ha-datacenter/vm", "hw_guest_full_name": null, "hw_guest_ha_state": null, "hw_guest_i
d": null, "hw_interfaces": ["eth0", "eth1"], "hw_is_template": false, "hw_memtotal_mb": 4096,
 "hw_name": "DevOps", "hw_power_status": "poweredOn", "hw_processor_count": 4, "hw_product_uu
id": "564d655b-92bd-384a-7d2e-86907e55d53e", "ipv4": null, "ipv6": null, "module_hw": true, "
snapshots": []}}
META: ran handlers
META: ran handlers

PLAY RECAP *********************************************************************************
localhost                  : ok=5    changed=2    unreachable=0    failed=0
```

You can validate the virtual machine creation and binding with the CentOS ISO file in the vSphere client:



You can also change the state of the existing virtual machine and choose from `poweredon`, `poweredoff`, `restarted`, `absent`, `suspended`, `shutdownguest`, and `rebootguest` by changing the value in `state` inside the playbook.

Summary

VMware products are used widely inside IT infrastructure to provide virtualized environments for running applications and workloads. At the same time, VMware also provides API bindings in many languages that can be used to automate administration tasks. In the next chapter, we will explore another virtualization framework called OpenStack that relies on the KVM hypervisor from Red Hat.

15
Interacting with the OpenStack API

For a long time, IT infrastructure depended on commercial software (from vendors such as VMWare, Microsoft, and Citrix) to provide virtual environments for running workloads and managing resources (such as computing, storage, and networking). However, IT industry is moving to cloud era and engineers are migrating workloads and applications to the cloud (either public or private), and that requires a new framework that is able to manage all application resources, providing an open and robust API interface to interact with external calls from other applications.

OpenStack provides an open access and integration to manage all of your computing, storage, and networking resources, avoiding a vendor lock-in when you're building your cloud. It can control a large pool of compute nodes, storage arrays, and networking devices, regardless of the vendor for each resource and provide a seamless integration between all resources. The core idea of OpenStack is to abstract all configuration applied on the underlay infrastructure into a *project* which is responsible for managing the resource. so you will find a project that manage the compute resources (called Nova) , another project that provide networking to the instances (neutron) and a projects to interact with different storage type (Swift and Cinder).

You can find a full list of the current OpenStack projects in this link

`https://www.OpenStack.org/software/project-navigator/`

Also OpenStack provide unified API access to the application developer and system administrators to orchestrate the resource creation.

In this chapter, we will explore the new and open world of OpenStack, and will learn how we can leverage Python and Ansible to interact with it.

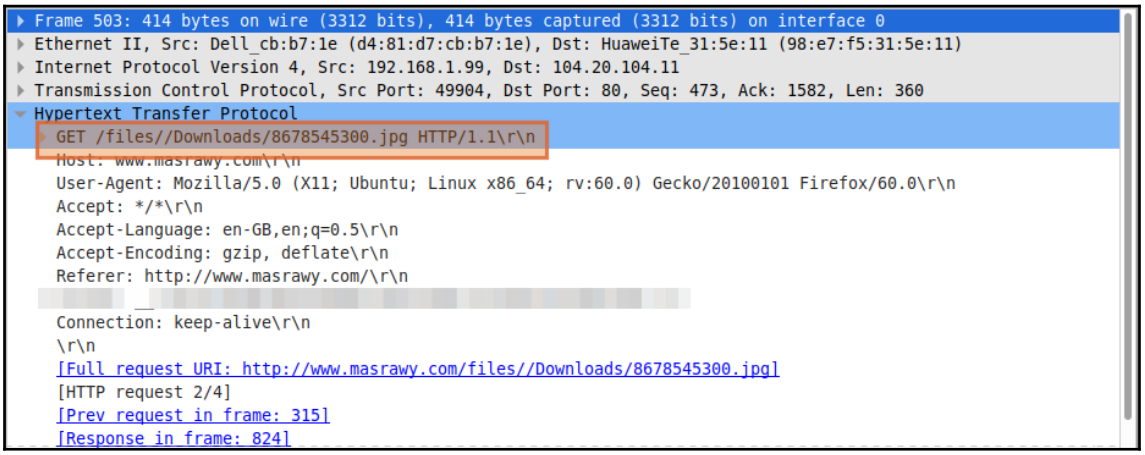
The following topics will be covered in this chapter:

- Understanding RESTful web services
- Setting up the environment
- Sending requests to OpenStack
- Creating workloads from Python
- Managing OpenStack instances using Ansible

Understanding RESTful web services

Representational State Transfer (**REST**) depends on HTTP protocol to transfer messages between the client and server. HTTP was originally designed to deliver HTML pages from web servers (servers) to browsers (clients), when requested. The pages represent a set of resources that the user wants to access, and are requested by **Universal Resource Identifiers** (**URIs**).

An HTTP request typically contains a method that indicates the type of operation that needs to be executed on the resource. For example, when visiting a website from your browser, you can see (in the following screenshot) that the method is `GET`:



The following are the most common HTTP methods, and their usage:

HTTP Method	Action
`GET`	The client will ask the server to retrieve the resource.
`POST`	The client will instruct the server to create a new resource.
`PUT`	The client will ask the server to modify/update the resource.
`DELETE`	The client will ask the server to delete the resource.

The application developer can expose certain resources of his application, to be consumed by the clients in the outside world. The transport protocol that carries the requests from the clients to servers and returns the responses back is HTTP. It is responsible for securing the communication and encoding the packet with the appropriate data encoding mechanism that is accepted by the server, and it is a stateless communication across both of them.

On the other hand, the packet payloads are usually encoded in either XML or JSON, to represent the structure of the request handled by the server and how the client prefers the response back.

There are many companies around the world that provide public access to their data for developers, in real time. For example, the Twitter API (`https://developer.twitter.com/`) provides a data fetch in real time, allowing other developers to consume the data in third-party applications like ads, searches, and marketing. The same goes for big names like Google (`https://developers.google.com/apis-explorer/#p/discovery/v1/`), LinkedIn (`https://developer.linkedin.com/`), and Facebook (`https://developers.facebook.com/`).

Public access to APIs is usually limited to a specific number of requests, either per hour of per day, for a single application, in order to not overwhelm the public resources.

Python provides a large set of tools and libraries to consume the APIs, encode the messages, and parse the responses. For example, Python has a `requests` package that can format and send HTTP requests to external resources. Also, it has tools to parse the responses in a JSON format and convert them to the standard dictionary in Python.

Python also has many frameworks that can expose your resources to the external world. `Django` and `Flask` are among the best, serving as full stack frameworks.

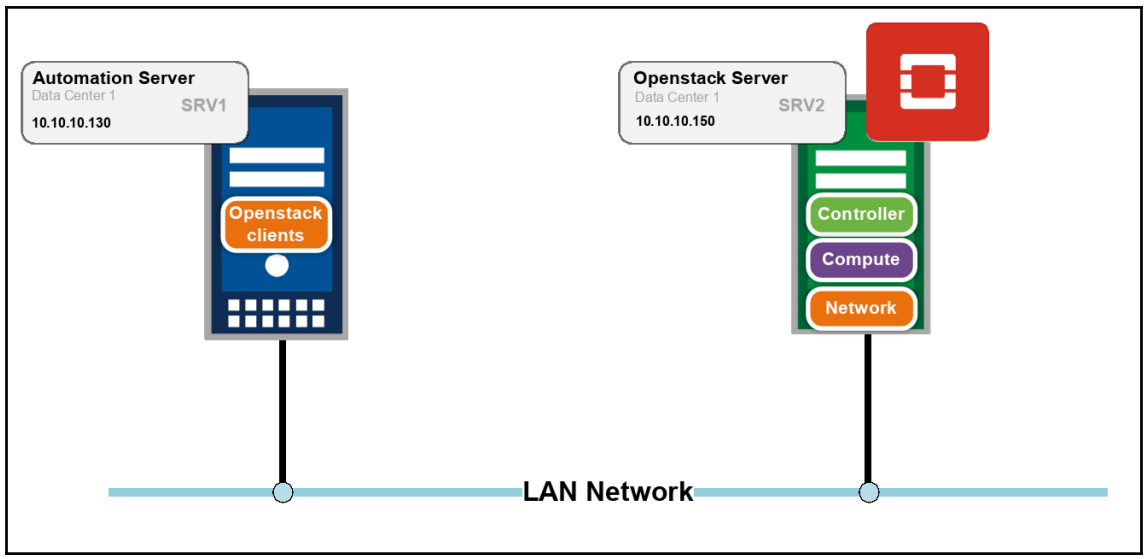
Setting up the environment

OpenStack is a free and open source project, used with **Infrastructure as a Service** (**IaaS**), that can control your hardware resources in terms of CPU, memory, and storage and provide an open framework for many vendors to build and integrate plugins.

To set up our lab, I will use the latest `OpenStack-rdo` release (at the time of writing), Queens, and install it onto CentOS 7.4.1708. The installation steps are pretty straightforward, and can be found at `https://www.rdoproject.org/install/packstack/`.

Our environment consists of a machine that has 100 GB storage, 12 vCPU, and 32 GB of RAM, This server will contains the OpenStack controller, the compute and neutron roles on the same server. The OpenStack server is connected to the same switch that has our automation server and in same subnet. Note that this is not always the case in a production environment, but you need to make sure that your server that runs Python code can reach the OpenStack.

The lab topology is as follows:



Installing rdo-OpenStack package

The steps for installing rdo-OpenStack on RHEL 7.4 and CentOS are as follows:

On RHEL 7.4

First, make sure that your system is up to date, and then install the `rdo-release.rpm` from the website to get the latest version. Finally, install the `OpenStack-packstack` package that will automate the OpenStack installation, as shown in the following snippet:

```
$ sudo yum install -y https://www.rdoproject.org/repos/rdo-release.rpm
$ sudo yum update -y
$ sudo yum install -y OpenStack-packstack
```

On CentOS 7.4

First, make sure that your system is up to date, and then install the rdoproject to get the latest version. Finally, install the `centos-release-OpenStack-queens` package that will automate the OpenStack installation, as shown in the following snippet:

```
$ sudo yum install -y centos-release-OpenStack-queens
$ sudo yum update -y
$ sudo yum install -y OpenStack-packstack
```

Generating answer file

Now, you will need to generate the answer file that contains the deployment parameters. Most of these parameters are fine on their defaults, but we will change a few things:

```
# packstack --gen-answer-file=/root/EnterpriseAutomation
```

Editing answer file

Edit the `EnterpriseAutomtion` file with your favorite editor, and change the following:

```
CONFIG_DEFAULT_PASSWORD=access123
CONFIG_CEILOMETER_INSTALL=n
CONFIG_AODH_INSTALL=n
CONFIG_KEYSTONE_ADMIN_PW=access123
CONFIG_PROVISION_DEMO=n
```

The `CELIOMETER` and `AODH` are an optional projects within OpenStack ecosystem and could be ignored in lab environment.

Also we setup a `KEYSTONE` password that used to generate temp token for accessing the resource using API and used also to access the OpenStack GUI

Run the packstack

Save the file and run the installation through the `packstack`:

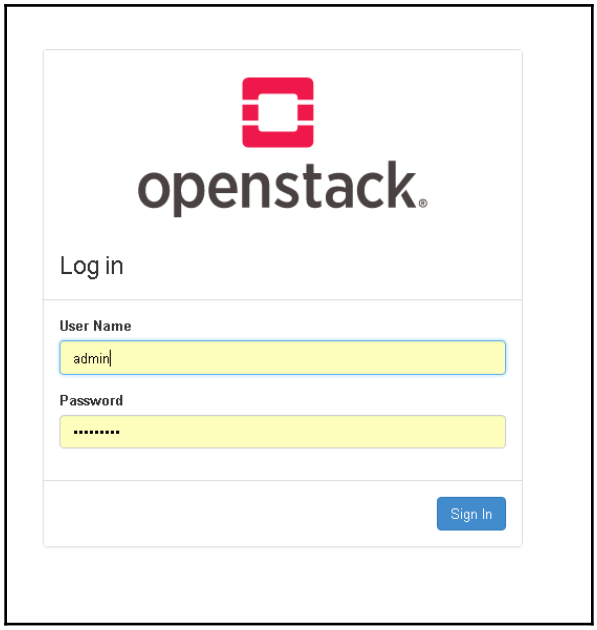
```
# packstack answer-file=EnterpriseAutomation
```

This command will download the packages from the Queens repository and install the OpenStack services, then start them. After the installation has completed successfully, the following message will be printed on the console:

```
 **** Installation completed successfully ******

Additional information:
 * Time synchronization installation was skipped. Please note that
unsynchronized time on server instances might be problem for some OpenStack
components.
 * File /root/keystonerc_admin has been created on OpenStack client host
10.10.10.150. To use the command line tools you need to source the file.
 * To access the OpenStack Dashboard browse to
http://10.10.10.150/dashboard .
Please, find your login credentials stored in the keystonerc_admin in your
home directory.
 * The installation log file is available at:
/var/tmp/packstack/20180410-155124-CMpsKR/OpenStack-setup.log
 * The generated manifests are available at:
/var/tmp/packstack/20180410-155124-CMpsKR/manifests
```

Access the OpenStack GUI

You can now access the OpenStack GUI using `http://<server_ip_address>/dashboard`. The credentials will be **admin** and **access123** (depending on what you wrote in `CONFIG_KEYSTONE_ADMIN_PW` in the previous steps):



Our cloud is now up and running, ready to receive requests.

Sending requests to the OpenStack keystone

OpenStack contains collections of services that work together to manage the virtual machine **create, read, update, and delete** (**CRUD**) operations. Each service can expose its resources to be consumed by external requests. For example, the `nova` service is responsible for spawning the virtual machine and acts as a hypervisor layer (though it's not a hypervisor itself, it can control other hypervisors, like KVM and vSphere). Another service is `glance`, responsible for hosting the instance images in either an ISO or qcow2 format. The `neutron` service is responsible for providing networking services to spawned instances and ensures that the instances located on different tenants (projects) are isolated from each other, while instances on the same tenants can reach each others through an overlays network (VxLAN or GRE).

In order to access the APIs of each of the preceding services, you will need to have an authenticated token that is used for a specific period of time. That's the role of the `keystone`, which provides an identity service and manages the roles and permissions of each user.

First, we need to install the Python bindings on our automation server. These bindings contain python code used to access each service and authenticate the request with the token generated from KEYSTONE. Also bindings contains supported operation for each project (like create/delete/update/list):

```
yum install -y gcc openssl-devel python-pip python-wheel
pip install python-novaclient
pip install python-neutronclient
pip install python-keystoneclient
pip install python-glanceclient
pip install python-cinderclient
pip install python-heatclient
pip install python-OpenStackclient
```

Note that the Python client name is `python-<service_name>client`

You can download into your site's global packages or the Python `virtualenv` environment. Then, you will need OpenStack admin privileges, which can be found in the following path, inside the OpenStack server:

```
cat /root/keystonerc_admin
unset OS_SERVICE_TOKEN
export OS_USERNAME=admin
export OS_PASSWORD='access123'
export OS_AUTH_URL=http://10.10.10.150:5000/v3
export PS1='[\u@\h \W(keystone_admin)]\$ '
export OS_PROJECT_NAME=admin
export OS_USER_DOMAIN_NAME=Default
export OS_PROJECT_DOMAIN_NAME=Default
export OS_IDENTITY_API_VERSION=3
```

Notice that we will use the keystone version 3 in both the `OS_AUTH_URL` and `OS_IDENTITY_API_VERSION` parameters when we communicate with the OpenStack keystone service. Most of the Python clients are compatible with older versions, but require you to change your script a little bit. Other parameters are also required during token generation, so make sure that you have access to the `keystonerc_admin` file. Also the access credentials can be found in `OS_USERNAME` and `OS_PASSWORD` in the same file

our Python script will be as follows:

```
from keystoneauth1.identity import v3
from keystoneauth1 import session

auth = v3.Password(auth_url="http://10.10.10.150:5000/v3",
                   username="admin",
                   password="access123",
                   project_name="admin",
                   user_domain_name="Default",
                   project_domain_name="Default")
sess = session.Session(auth=auth, verify=False)
print(sess)
```

In the preceding example, the following applies:

- `python-keystoneclient` made a request to the keystone API using the `v3` class (which reflects the keystone API version). This class is available inside of `keystoneayth1.identity`.
- Then, we supplied the full credentials taken from the `keystonerc_admin` file to the `auth` variable.
- Finally, we established the session, using the session manager inside of the keystone client. Notice that we set `verify` to `False`, since we don't use the certificate to generate the token. Otherwise, you can supply the certificate path.
- The token generated can be used with any service, and it will last for one hour, then expire. Also, if you change the user role, the token will expire immediately, without waiting for an hour.

OpenStack administrators can configure the `admin_token` field inside the `/etc/keystone/keystone.conf` file, which never expires. However, this is not recommended in a production environment, for security reasons.

If you don't want to store the credentials inside the Python script, you can store them in the `ini` file and load them using the `configparser` module. First, create a `creds.ini` file in the automation server, and give it appropriate Linux permissions, so it can only be opened with your own account:

```
#vim /root/creds.ini

[os_creds]
auth_url="http://10.10.10.150:5000/v3"
username="admin"
password="access123"
project_name="admin"
user_domain_name="Default"
project_domain_name="Default"
```

The modified script is as follows:

```
from keystoneauth1.identity import v3
from keystoneauth1 import session
import ConfigParser
config = ConfigParser.ConfigParser()
config.read("/root/creds.ini")
auth = v3.Password(auth_url=config.get("os_creds","auth_url"),
                   username=config.get("os_creds","username"),
                   password=config.get("os_creds","password"),
                   project_name=config.get("os_creds","project_name"),
user_domain_name=config.get("os_creds","user_domain_name"),
project_domain_name=config.get("os_creds","project_domain_name"))
sess = session.Session(auth=auth, verify=False)
print(sess)
```

The `configparser` module will parse the `creds.ini` file and look at the `os_creds` section inside the file. Then, it will get the value in front of each parameter by using the `get()` method.

The `config.get()` method will accept two arguments. The first argument is the section name inside the `.ini` file, and the second is the parameter name. The method will return the value associated with the parameter.

This method should provide additional security to your cloud credentials. Another valid method to secure your file is to load the `keystonerc_admin` file into the environmental variables using the Linux `source` command, and read the credentials using the `environ()` method inside of the `os` module.

Creating instances from Python

To get instance up and running, OpenStack instances require three components. The boot image, which is provided by `glance`, the network ports, which provided by `neutron`, and finally, the compute flavor that defines the number of CPUs, amount of RAM that will be allocated to the instance and disk size. The flavor is provided by `nova` project.

Creating the image

We will start by downloading a `cirros` image to the automation server. `cirros` is a lightweight, Linux-based image, used by many OpenStack developers and testers around the world to validate the functionality of OpenStack services:

```
#cd /root/ ; wget
http://download.cirros-cloud.net/0.4.0/cirros-0.4.0-x86_64-disk.img
```

Then, we will upload the image to the OpenStack image repository using `glanceclient`. Notice that we need to have the keystone token and the session parameter first, in order to communicate with `glance`, otherwise, `glance` won't accept any API requests from us.

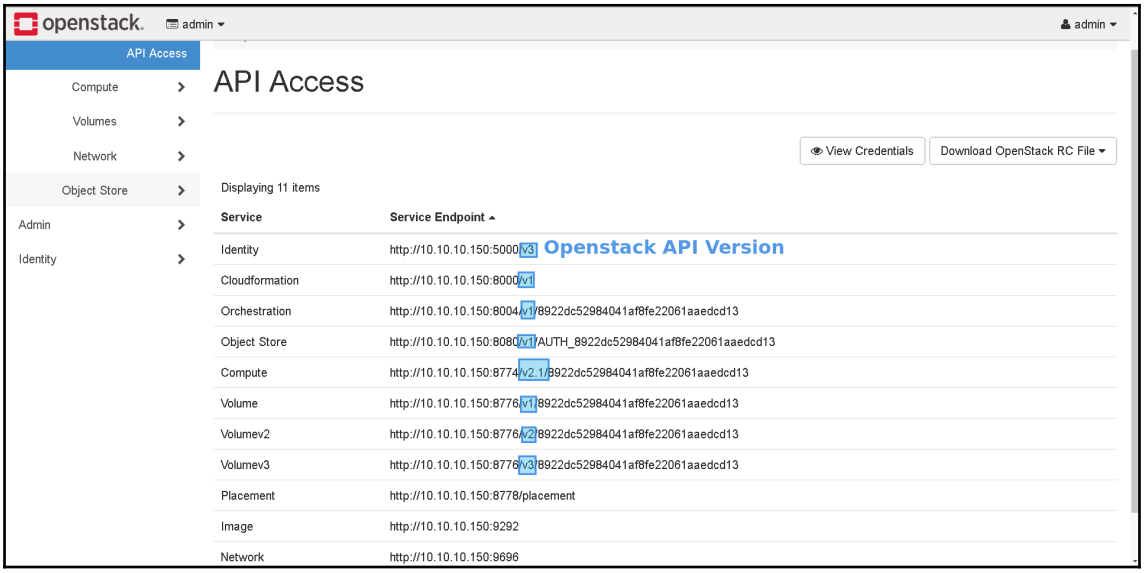
The script will be as follows:

```
from keystoneauth1.identity import v3
from keystoneauth1 import session
from glanceclient import client as gclient
from pprint import pprint


auth = v3.Password(auth_url="http://10.10.10.150:5000/v3",
                   username="admin",
                   password="access123",
                   project_name="admin",
                   user_domain_name="Default",
                   project_domain_name="Default")


sess = session.Session(auth=auth, verify=False)

#Upload the image to the Glance
glance = gclient.Client('2', session=sess)

image = glance.images.create(name="CirrosImage",
                             container_format='bare',
                             disk_format='qcow2',
```

```
                                    )

glance.images.upload(image.id, open('/root/cirros-0.4.0-x86_64-disk.img',
'rb'))
```

In the preceding example, the following applies:

- Since we are communicating with `glance` (the image hosting project), we will import the `client` from the installed `glanceclient` module.
- The same keystone scripts used to generate the `sess` that holds the keystone token.
- We created the glance parameter that initializes the client manager with `glance` and provide the version (`version 2`) and the generated token.
- You can see all supported API versions by accessing the **OpenStack GUI | API Access** tab as in below screenshot. notice also the supported version for each project.



- The glance client manager is designed to operate on the glance OpenStack service. the manager is instructed to create an image with a name `CirrosImage` and disk type is in `qcow2` format.

- Finally, we will open the downloaded image as a binary, using the `'rb'` flag, and will upload it to the created image. Now, `glance` will import the image to the newly created file in the image repository.

You can validate that the operation was successful in two ways:

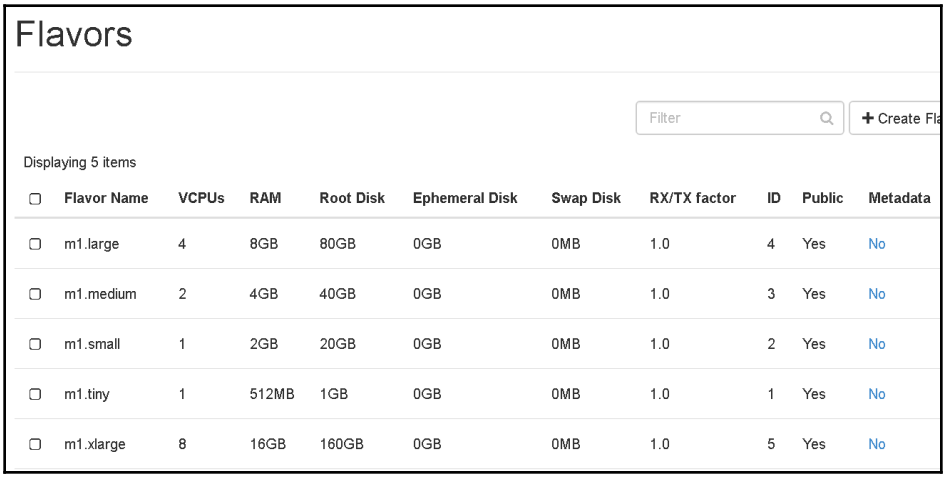
1. If no error is printed back after executing `glance.images.upload()`, it means that the request is correctly formatted and has been accepted by the OpenStack `glance` API.
2. Run the `glance.images.list()` . The returned output will be a generate which you can iterate over it to see more details about the uploaded image:

```
print("==========================Image
Details==========================")
for image in glance.images.list(name="CirrosImage"):
    pprint(image)

{u'checksum': u'443b7623e27ecf03dc9e01ee93f67afe',
 u'container_format': u'bare',
 u'created_at': u'2018-04-11T03:11:58Z',
 u'disk_format': u'qcow2',
 u'file': u'/v2/images/3c2614b0-e53c-4be1-b99d-bbd9ce14b287/file',
 u'id': u'3c2614b0-e53c-4be1-b99d-bbd9ce14b287',
 u'min_disk': 0,
 u'min_ram': 0,
 u'name': u'CirrosImage',
 u'owner': u'8922dc52984041af8fe22061aaedcd13',
 u'protected': False,
 u'schema': u'/v2/schemas/image',
 u'size': 12716032,
 u'status': u'active',
 u'tags': [],
 u'updated_at': u'2018-04-11T03:11:58Z',
 u'virtual_size': None,
 u'visibility': u'shared'}
```

Assigning a flavor

Flavors are used to determine the CPU, memory, and storage size of the instance. OpenStack comes with a predefined set of flavors, with different sizes that range from tiny to extra large. For the `cirros` image, we will use the small flavor, which has 2 GB RAM, 1 vCPU, and 20 GB storage. Access to flavors doesn't have a standalone API client; rather, it's a part of the `nova` client.

You can see all available built-in flavors at **OpenStack GUI | Admin | Flavors:**



Flavors

Filter + Create Fl

Displaying 5 items

Flavor Name	VCPUs	RAM	Root Disk	Ephemeral Disk	Swap Disk	RX/TX factor	ID	Public	Metadata
m1.large	4	8GB	80GB	0GB	0MB	1.0	4	Yes	No
m1.medium	2	4GB	40GB	0GB	0MB	1.0	3	Yes	No
m1.small	1	2GB	20GB	0GB	0MB	1.0	2	Yes	No
m1.tiny	1	512MB	1GB	0GB	0MB	1.0	1	Yes	No
m1.xlarge	8	16GB	160GB	0GB	0MB	1.0	5	Yes	No

The script will be as follows:

```
from keystoneauth1.identity import v3
from keystoneauth1 import session
from novaclient import client as nclient
from pprint import pprint

auth = v3.Password(auth_url="http://10.10.10.150:5000/v3",
                   username="admin",
                   password="access123",
                   project_name="admin",
                   user_domain_name="Default",
                   project_domain_name="Default")

sess = session.Session(auth=auth, verify=False)

nova = nclient.Client(2.1, session=sess)
instance_flavor = nova.flavors.find(name="m1.small")
print("==========================Flavor Details==========================")
pprint(instance_flavor)
```

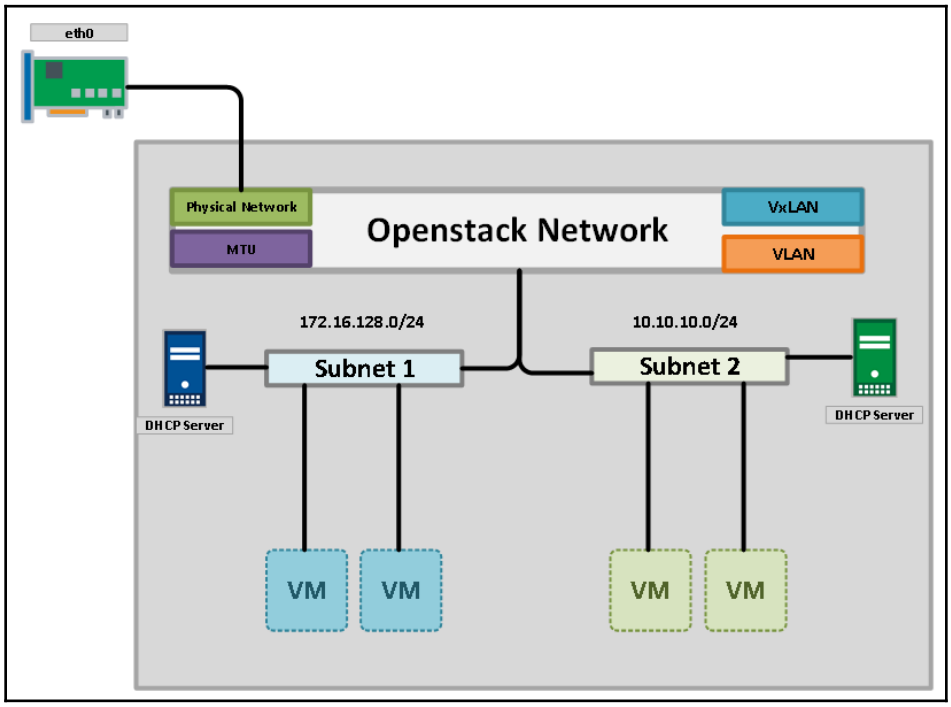
In the preceding script, the following applies:

- Since we will communicate with `nova` (the compute service) to retrieve the flavor, we will import the `novaclient` module as `nclient`.
- The same keystone script is used to generate the `sess` that holds the keystone token.

- We created the `nova` parameter that initialized the client manager with the `nova` and provide the version to the client (version 2.1) and the generated token.
- Finally, we used the `nova.flavors.find()` method to locate the desired flavor, which is `m1.small`. The name has to match the name in OpenStack exactly, otherwise it will throw an error.

Creating the network and subnet

Creating the network for the instance requires two things: the network itself, and associating subnet with it. First, we need to supply the network properties, such as the ML2 driver (Flat, VLAN, VxLAN, and so on), the segmentation ID that differentiates between the networks running on the same interface, the MTU, and the physical interface, if the instance traffic needs to traverse external networks. Second, we need to provide the subnet properties, such as the network CIDR, the gateway IP, The IPAM parameters (DHCP/DNS server if defined), and which network ID is associated with the subnet as in below screenshot:



Now we will develop a Python script to interact with the neutron project and create a network with a subnet

```
from keystoneauth1.identity import v3
from keystoneauth1 import session
import neutronclient.neutron.client as neuclient

auth = v3.Password(auth_url="http://10.10.10.150:5000/v3",
                   username="admin",
                   password="access123",
                   project_name="admin",
                   user_domain_name="Default",
                   project_domain_name="Default")

sess = session.Session(auth=auth, verify=False)

neutron = neuclient.Client(2, session=sess)

# Create Network

body_network = {'name': 'python_network',
                'admin_state_up': True,
                #'port_security_enabled': False,
                'shared': True,
                # 'provider:network_type': 'vlan|vxlan',
                # 'provider:segmentation_id': 29
                # 'provider:physical_network': None,
                # 'mtu': 1450,
                }
neutron.create_network({'network':body_network})
network_id =
neutron.list_networks(name="python_network")["networks"][0]["id"]


# Create Subnet

body_subnet = {
            "subnets":[
                {
                    "name":"python_network_subnet",
                    "network_id":network_id,
                    "enable_dhcp":True,
                    "cidr": "172.16.128.0/24",
                    "gateway_ip": "172.16.128.1",
                    "allocation_pools":[
                        {
                            "start": "172.16.128.10",
```

```
                                "end": "172.16.128.100"
                            }
                        ],
                        "ip_version": 4,
                    }
                ]
            }
    neutron.create_subnet(body=body_subnet)
```

In the preceding script, the following applies:

- Since we will communicate with `neutron` (the network service) to create both the network and associated subnet, we will import the `neutronclient` module as the `neuclient`.
- The same keystone script is used to generate the `sess` that holds the keystone token used later to access neutron resource.
- We will create the `neutron` parameter that initializes the client manager with neutron and provide the version to it (version 2) and the generated token.
- Then, we created two Python dictionaries, `body_network` and `body_subnet` which hold the message bodies for the network and subnet respectively. Note that the dictionary keys are static and can't be changed, while the values could be changed and usually provided from external portal system or Excel sheet, depending on your deployment. Also, I commented on the parts that are not necessary during network creation, such as `provider:physical_network` and `provider:network_type`, since our `cirros` image won't communicate with the provider network (networks defined outside OpenStack domains) but provided here for reference.
- Finally the subnet and the network associated together by getting first the `network_id` through the `list_networks()` method and access the id and providing it as a value to `network_id` key inside the `body_subnet` variable.

Launching the instance

The final part is to glue everything together. We have the boot image, the instance flavor, and the network that connects the machine with the other instances. We're ready to launch the instance using the `nova` client (remember that `nova` is responsible for the virtual machine life cycle and the CRUD operations on the VM):

```
print("=================Launch The Instance=================")

image_name = glance.images.get(image.id)
```

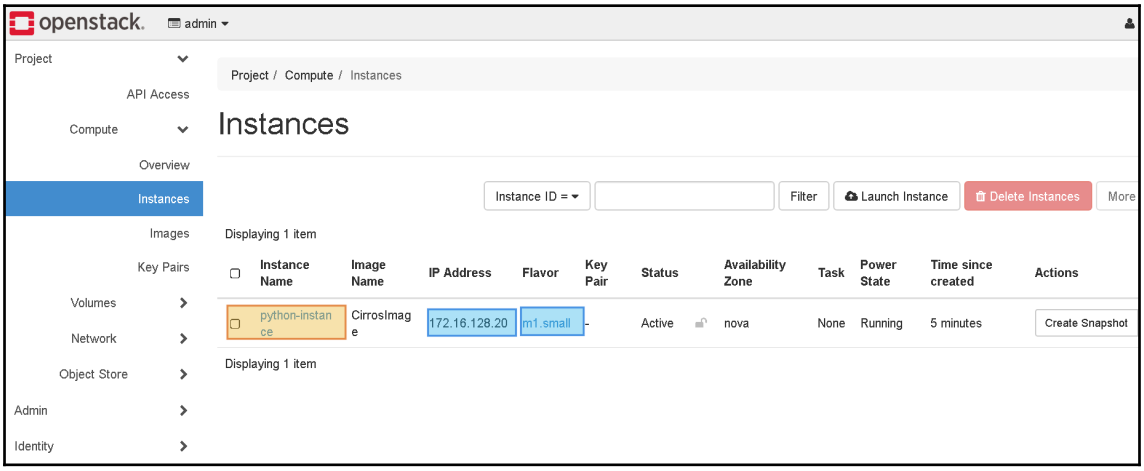
```
network1 = neutron.list_networks(name="python_network")
instance_nics = [{'net-id': network1["networks"][0]["id"]}]

server = nova.servers.create(name = "python-instance",
                             image = image_name.id,
                             flavor = instance_flavor.id,
                             nics = instance_nics,)
status = server.status
while status == 'BUILD':
    print("Sleeping 5 seconds till the server status is changed")
    time.sleep(5)
    instance = nova.servers.get(server.id)
    status = instance.status
    print(status)
print("Current Status is: {0}".format(status))
```

In the preceding script, we used the `nova.servers.create()` method and passed all of the information required to spawn the instance(instance name, operating system, flavor and networks). Additionally, we implemented a polling mechanism that polls the nova service for the server current status. If the server is still in `BUILD` phase, then the script will sleeps for five seconds then poll again. The loop will exit when the server status is changes to either `ACTIVE` or `FAILURE` and will prints the server status at the end.

The script's output is as follows:

```
Sleeping 5 seconds till the server status is changed
Sleeping 5 seconds till the server status is changed
Sleeping 5 seconds till the server status is changed
Current Status is: ACTIVE
```

Also, you can check the instance from the **OpenStack GUI** | **Compute** | **Instances**:



Managing OpenStack instances from Ansible

Ansible provides modules that can manage the OpenStack instance life cycle, just like we did using APIs. You can find the full list of supported modules at `http://docs.ansible.com/ansible/latest/modules/list_of_cloud_modules.html#OpenStack`.

All OpenStack modules rely on the Python library called `shade` (`https://pypi.python.org/pypi/shade`), which provides a wrapper around OpenStack clients.

Once you have installed `shade` on the automation server, you will have access to the `os-*` modules that can manipulate the OpenStack configuration, such as `os_image` (to handle OpenStack images), `os_network` (to create the network), `os_subnet` (to create and associate the subnet with the created network), `os_nova_flavor` (to create flavors, given the RAM, CPU, and disk), and finally, the `os_server` module (to bring up the OpenStack instance).

Shade and Ansible installation

In the automation server, use the Python `pip` to download and install `shade`, with all dependencies:

```
pip install shade
```

After installation, you will have `shade` under the normal `site-packages` in Python, but we will use Ansible instead.

Also, you will need to install Ansible in the automation server, if you haven't done it in previous chapters:

```
# yum install ansible -y
```

Verify that Ansible has installed successfully by querying the Ansible version from the command line:

```
[root@AutomationServer ~]# ansible --version
ansible 2.5.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Aug  4 2017, 00:39:18) [GCC 4.8.5
20150623 (Red Hat 4.8.5-16)]
```

Building the Ansible playbook

As we saw in `Chapter 13`, *Ansible for Administration*, depends on a YAML file to contain everything you will need to execute against hosts in the inventory. In this case, we will instruct the playbook to establish a local connection to the `shade` library on the automation server, and provide the playbook with the `keystonerc_admin` credentials that help `shade` to send requests to our OpenStack server.

The playbook script is as follows:

```
---
- hosts: localhost
  vars:
      os_server: '10.10.10.150'
  gather_facts: yes
  connection: local
  environment:
    OS_USERNAME: admin
    OS_PASSWORD: access123
    OS_AUTH_URL: http://{{ os_server }}:5000/v3
    OS_TENANT_NAME: admin
    OS_REGION_NAME: RegionOne
    OS_USER_DOMAIN_NAME: Default
    OS_PROJECT_DOMAIN_NAME: Default

  tasks:
    - name: "Upload the Cirros Image"
      os_image:
        name: Cirros_Image
        container_format: bare
        disk_format: qcow2
        state: present
        filename: /root/cirros-0.4.0-x86_64-disk.img
      ignore_errors: yes

    - name: "CREATE CIRROS_FLAVOR"
      os_nova_flavor:
        state: present
        name: CIRROS_FLAVOR
        ram: 2048
        vcpus: 4
        disk: 35
      ignore_errors: yes

    - name: "Create the Cirros Network"
      os_network:
        state: present
        name: Cirros_network
        external: True
        shared: True
      register: Cirros_network
      ignore_errors: yes


    - name: "Create Subnet for The network Cirros_network"
      os_subnet:
```

```
      state: present
      network_name: "{{ Cirros_network.id }}"
      name: Cirros_network_subnet
      ip_version: 4
      cidr: 10.10.128.0/18
      gateway_ip: 10.10.128.1
      enable_dhcp: yes
      dns_nameservers:
        - 8.8.8.8
    register: Cirros_network_subnet
    ignore_errors: yes


  - name: "Create Cirros Machine on Compute"
    os_server:
      state: present
      name: ansible_instance
      image: Cirros_Image
      flavor: CIRROS_FLAVOR
      security_groups: default
      nics:
        - net-name: Cirros_network
    ignore_errors: yes
```

In the playbook, we make use of the `os_*` modules to upload the image to the OpenStack `glance` server, create a new flavor (and not using this built-in), and create the network with the subnet associated; then, we glue everything together in `os_server`, which communicates with the `nova` server to spawn the machine.

Please note that the hosts will be the localhost (or the machine name that hosts the `shade` library), while we added the OpenStack keystone credentials in the environmental variables.

Running the playbook

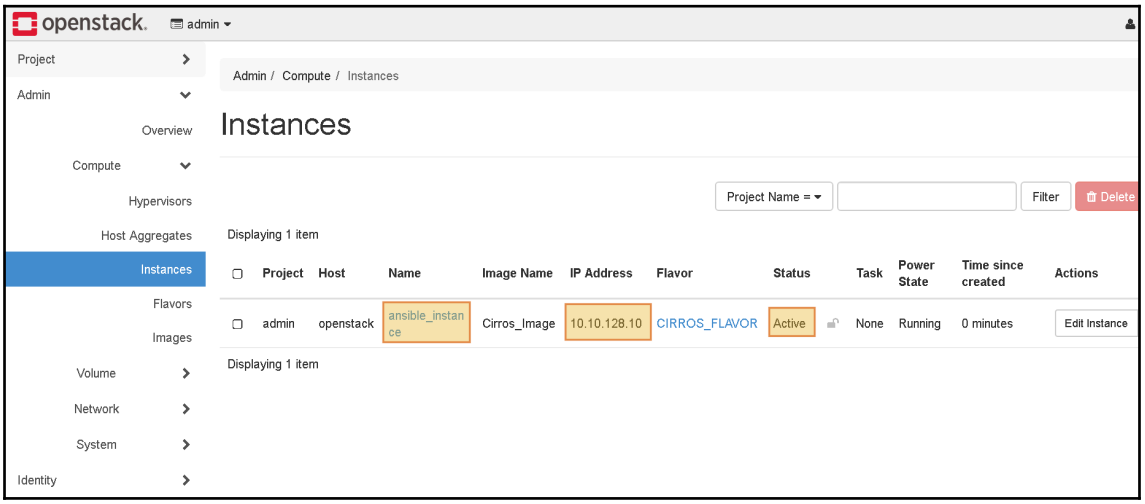
Upload the playbook to the automation server and execute the following command to run it:

```
ansible-playbook os_playbook.yml
```

The playbook's output will be as follows:

```
 [WARNING]: No inventory was parsed, only implicit localhost is available

 [WARNING]: provided hosts list is empty, only localhost is available. Note
that the implicit localhost does not match 'all'


PLAY [localhost]
*********************************************************************************
*

TASK [Gathering Facts]
*********************************************************************
ok: [localhost]

TASK [Upload the Cirros Image]
************************************************************
changed: [localhost]

TASK [CREATE CIRROS_FLAVOR]
*************************************************************
ok: [localhost]

TASK [Create the Cirros Network]
*********************************************************
changed: [localhost]

TASK [Create Subnet for The network Cirros_network]
*************************************
changed: [localhost]

TASK [Create Cirros Machine on Compute]
************************************************
changed: [localhost]

PLAY RECAP
*********************************************************************************
*******
localhost                  : ok=6    changed=4    unreachable=0    failed=0
```

You can access the OpenStack GUI to validate that the instance was created from the Ansible playbook:



Summary

Nowadays, the IT industry is trying to avoid vendor lock-in by moving to the open source world whenever possible. OpenStack provides a window into this world; many large organizations and telecom operators are considering moving their workloads to OpenStack, to build their private clouds in its data center. They can then build their own tools to interact with the open source APIs provided by OpenStack.

In the next chapter, we will explore another (paid) public Amazon cloud, and will learn how we can leverage Python to automate instance creation.

16
Automating AWS with Boto3

In previous chapters, we explored how to automate the OpenStack and VMware private clouds using Python. We will continue on our cloud automation journey by automating one of the most popular public clouds: **Amazon Web Services** (**AWS**). In this chapter, we will explore how to create Amazon **Elastic Compute Cloud** (**EC2**) and Amazon **Simple Storage Systems** (**S3**) using Python script.

We will cover the following topics in this chapter:

- AWS Python modules
- Managing AWS instances
- Automating AWS S3 services

AWS Python modules

Amazon EC2 is a scalable computing system that is used to provide virtualization layers for hosting different virtual machines (such as the nova-compute project in the OpenStack ecosystem). It can communicate with other services, such as S3, Route 53, and AMI, in order to instantiate instances. Basically, you can think of EC2 as an abstraction layer above other hypervisors that are set over the virtual infrastructure manager (such as KVM and VMware). EC2 will receive the incoming API calls then will translate them into suitable calls for each hypervisor.

The **Amazon Machine Image** (**AMI**) is a packaged image system that contains the operating system and packages needed to start a virtual machine (like Glance in OpenStack). You can create your own AMI from existing virtual machines and use it when you need to replicate those machines on other infrastructures, or you can simply choose from publicly available AMIs on the internet or on the Amazon Marketplace. We will need to get the AMI ID from the Amazon web console and add it to our Python script.

AWS designed an SDK called **Boto3** (`https://github.com/boto/boto3`) that allows Python developers to have scripts and software that interact and consume the APIs of different services, like Amazon EC2 and Amazon S3. The library was written to provide native support for Python 2.6.5, 2.7+, and 3.3.

The major Boto3 features are described in the official documentation at `https://boto3.readthedocs.io/en/latest/guide/new.html`, and below are some important features:

- **Resources**: A high-level, object-oriented interface.
- **Collections**: A tool to iterate and manipulate groups of resources.
- **Clients**: A low-level service connection.
- **Paginators**: Automatic paging of responses.
- **Waiters**: A way to suspend execution until a certain state has been reached or a failure occurs. Each AWS resource has a waiter name that could be accessed using `<resource_name>.waiter_names`.

Boto3 installation

A few things are needed before connecting to AWS:

1. First, you will need an Amazon admin account that has privileges to create, modify, and delete from the infrastructure.
2. Secondly, install the `boto3` Python modules that are used to interact with AWS. You can create a user dedicated to sending API requests by going to the AWS **Identity and Access Management** (**IAM**) console and adding a new user. You should see the **Programmatic access** option, available under the **Access Type** section.

3. Now, you will need to assign a policy that allows full access across the Amazon services, such as EC2 and S3. Do that by clicking on **Attach existing policy to user** and attaching **AmazonEC2FullAccess** and **AmazonS3FullAccess** policies to the username.
4. At the end, click on **Create user** to add the user with the configured options and policies.

You can sign up for a free tier account on AWS, which will give you access to many services offered by Amazon for up to 12 months. Free access can be acquired at `https://aws.amazon.com/free/`.

When using Python script to manage AWS, the access key ID is used to send API requests and get the responses back from the API server. We won't use the username or the password for sending requests, as they're easily captured by others. This information is obtained by downloading the text file that appears after creating the username. It's important to keep this file in a safe place and provide a proper Linux permission for it, for opening and reading file content.

Another method is to create a `.aws` directory under your home user directory and place two files under it: `credentials` and `config`. The first file will have both the access key ID and the secret access ID.

`~/.aws/credentials` appears as follows:

```
[default]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

The second file will hold user-specific configurations, such as the preferred data center (zone) that will host the created virtual machines. (This is like the availability zone option in OpenStack.) In the following example, we are specifying that we want to host our machines in the `us-west-2` data center.

The config file, `~/.aws/config`, looks like the following:

```
[default]
region=us-west-2
```

Now, installing `boto3` requires using the usual `pip` command to get the latest `boto3` version:

```
pip install boto3
```

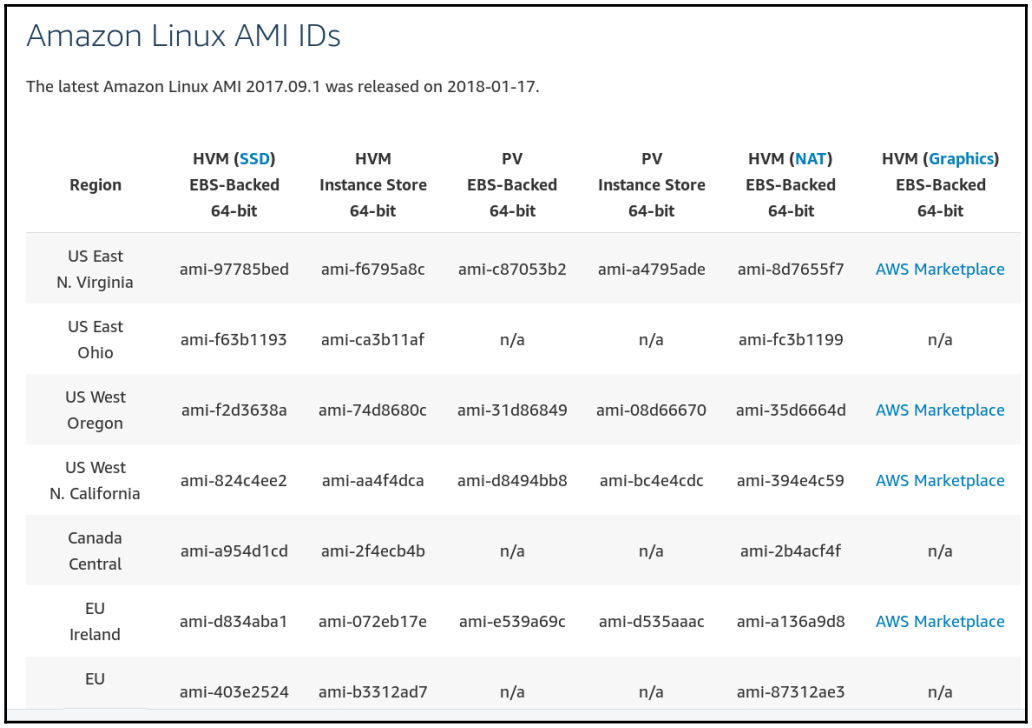
```
bassim:~$ pip install boto3
Collecting boto3
  Downloading https://files.pythonhosted.org/packages/b8/29/f35b0a055014296bf4188043e2cc1fd4
ca041a085991765598842232c2f5/boto3-1.7.26-py2.py3-none-any.whl (128kB)
    100% |████████████████████████████████| 133kB 351kB/s
Collecting jmespath<1.0.0,>=0.7.1 (from boto3)
  Downloading https://files.pythonhosted.org/packages/b7/31/05c8d001f7f87f0f07289a5fc0fc3832
e9a57f2dbd4d3b0fee70e0d51365/jmespath-0.9.3-py2.py3-none-any.whl
Collecting botocore<1.11.0,>=1.10.26 (from boto3)
  Downloading https://files.pythonhosted.org/packages/87/c5/7ed94b700d30534f346bb55408ca8501
325840bcdc371628cff10d7ba68d/botocore-1.10.26-py2.py3-none-any.whl (4.2MB)
    100% |████████████████████████████████| 4.2MB 324kB/s
Collecting s3transfer<0.2.0,>=0.1.10 (from boto3)
  Downloading https://files.pythonhosted.org/packages/d7/14/2a0004d487464d120c9fb85313a75cd3
d71a7506955be458eebfe19a6b1d/s3transfer-0.1.13-py2.py3-none-any.whl (59kB)
    100% |████████████████████████████████| 61kB 363kB/s
Collecting docutils>=0.10 (from botocore<1.11.0,>=1.10.26->boto3)
  Downloading https://files.pythonhosted.org/packages/50/09/c53398e0005b11f7ffb27b7aa720c617
aba53be4fb4f4f3f06b9b5c60f28/docutils-0.14-py2-none-any.whl (543kB)
    100% |████████████████████████████████| 552kB 391kB/s
Requirement already satisfied: python-dateutil<3.0.0,>=2.1; python_version >= "2.7" in ./.lo
cal/lib/python2.7/site-packages (from botocore<1.11.0,>=1.10.26->boto3) (2.6.1)
Collecting futures<4.0.0,>=2.2.0; python_version == "2.6" or python_version == "2.7" (from s
3transfer<0.2.0,>=0.1.10->boto3)
  Downloading https://files.pythonhosted.org/packages/2d/99/b2c4e9d5a30f6471e410a146232b4118
e697fa3ffc06d6a65efde84debd0/futures-3.2.0-py2-none-any.whl
```

To verify that the module has successfully installed, import `boto3` in the Python console, and you shouldn't see any import errors reported:

```
bassim:~$ python
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>>
```

Managing AWS instances

Now, we're ready to create our first virtual machine using `boto3`. As we have discussed, we need the AMI that we will instantiate an instance from. Think of an AMI as a Python class; creating an instance will create an object from it. We will use the Amazon Linux AMI, which is a special Linux operating system maintained by Amazon and used for deploying Linux machines without any extra charges. You can find a full AMI ID, per region, at `https://aws.amazon.com/amazon-linux-ami/`:



Amazon Linux AMI IDs

The latest Amazon Linux AMI 2017.09.1 was released on 2018-01-17.

Region	HVM (SSD) EBS-Backed 64-bit	HVM Instance Store 64-bit	PV EBS-Backed 64-bit	PV Instance Store 64-bit	HVM (NAT) EBS-Backed 64-bit	HVM (Graphics) EBS-Backed 64-bit
US East N. Virginia	ami-97785bed	ami-f6795a8c	ami-c87053b2	ami-a4795ade	ami-8d7655f7	AWS Marketplace
US East Ohio	ami-f63b1193	ami-ca3b11af	n/a	n/a	ami-fc3b1199	n/a
US West Oregon	ami-f2d3638a	ami-74d8680c	ami-31d86849	ami-08d66670	ami-35d6664d	AWS Marketplace
US West N. California	ami-824c4ee2	ami-aa4f4dca	ami-d8494bb8	ami-bc4e4cdc	ami-394e4c59	AWS Marketplace
Canada Central	ami-a954d1cd	ami-2f4ecb4b	n/a	n/a	ami-2b4acf4f	n/a
EU Ireland	ami-d834aba1	ami-072eb17e	ami-e539a69c	ami-d535aaac	ami-a136a9d8	AWS Marketplace
EU	ami-403e2524	ami-b3312ad7	n/a	n/a	ami-87312ae3	n/a

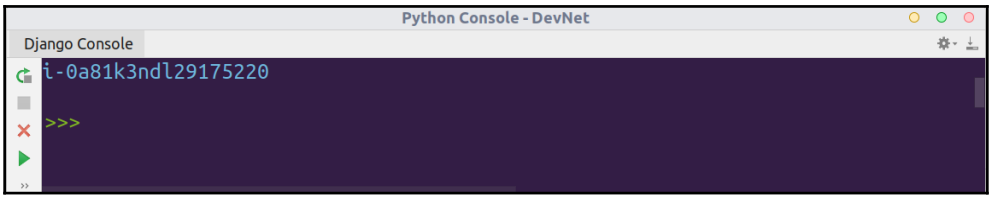
```
import boto3
ec2 = boto3.resource('ec2')
instance = ec2.create_instances(ImageId='ami-824c4ee2', MinCount=1,
MaxCount=1, InstanceType='m5.xlarge',
                                Placement={'AvailabilityZone': 'us-
west-2'},
                                )
print(instance[0])
```

In the preceding example, the following applies:

1. We imported the `boto3` module that we installed previously.
2. Then, we specified a resource type that we wanted to interact with, which is EC2, and assigned that to the `ec2` object.

3. Now, we are eligible to use the `create_instance()` method and provide it with instance parameters, such as `ImageID` and `InstanceType` (like flavor in OpenStack, which determines the instance specs in terms of computing and memory), and where we should create this instance in the `AvailabilityZone`.
4. `MinCount` and `MaxCount` determine how far EC2 can go when scaling our instances. For example, when a high CPU has occurred on one of the instances, EC2 will deploy another instance automatically, to share the loads and keep the service in a healthy state.
5. Finally, we printed the instance ID to be used in the next script.

The output is as follows:



You can check all valid Amazon EC2 instance types at the following link; please read them carefully, in order to not be overcharged from choosing the wrong type: `https://aws.amazon.com/ec2/instance-types/`

Instance termination

The printed ID is used in CRUD operations to manage or terminate the instance later. For example, we can terminate the instance by using the `terminate()` method also provided to the `ec2` resource created earlier:

```
import boto3
ec2 = boto3.resource('ec2')
instance_id = "i-0a81k3ndl29175220"
instance = ec2.Instance(instance_id)
instance.terminate()
```

Notice that we hardcoded `instance_id` in the preceding code (which is not always the case when you need to create a dynamic Python script that can be used in different environments). We can use other input methods that are available in Python, such as `raw_input()`, to take the input from the user or query the available instances in our accounts and make Python prompt us on which instances need to be terminated. Another use case is to create a Python script that checks the last login time or the resource consumption in our instance; if they exceed a specific value, we will terminate the instance. This is useful in a lab environment, where you don't want to be charged for consuming additional resources with a malicious or a poorly designed software.

Automating AWS S3 services

The AWS **Simple Storage Systems** (**S3**) provides a safe and highly scalable object storage service. You can use this service to store any amount of data and restore it from anywhere. The system provides you with a versioning option, so you can roll back to any previous version of the files. Additionally, it provides the REST web services API, so you can access it from external applications.

When data comes to S3, S3 will create an `object` for it, and these objects will be stored inside `Buckets` (think of them like folders). You can provide a sophisticated user permission for each created bucket, and can also control its visibility (public, shared, or private). The bucket access can be either a policy or an **Access Control List** (**ACL**).

The bucket is also stored with metadata that describes the object in key-value pairs, which you can create and set by HTTP `POST` methods. Metadata can include the object's name, size, and date, or any other customized key-values that you want. The user account has a limit of 100 buckets, but there's no limit on the size of the object hosted inside each bucket.

Creating buckets

The first logical thing to do, when interacting with an AWS S3 service, is create a bucket that can be used to store files. In that case, we will provide the `S3` to the `boto3.resource()` . That will tell the `boto3` to start the initialization process and will load required commands to interact with the S3 API system:

```
import boto3
s3_resource = boto3.resource("s3")

bucket = s3_resource.create_bucket(Bucket="my_first_bucket",
CreateBucketConfiguration={
```

```
    'LocationConstraint': 'us-west-2'})
print(bucket)
```

In the preceding example, the following applies:

1. We imported the `boto3` module that we installed previously.
2. Then, we specified a resource type that we wanted to interact with, which is `s3`, and assigned that to the `s3_resource` object.
3. Now, we can use the `create_bucket()` method inside the resource and provide it with the required parameter to create buckets, such as `Bucket`, where we can specify its name. Remember, the bucket name must be unique and cannot have been used previously. The second parameter is the `CreateBucketConfiguration` dictionary, where we set the data center location for the created bucket.

Uploading a file to a bucket

Now, we need to make use of the created bucket and upload a file to it. Remember, the file representation inside the bucket is an object. So, `boto3` provides some methods that contain the object as a part of it. We will start by using `put_object()`. This method will upload a file to the created bucket and store it as an object:

```
import boto3
s3_resource = boto3.resource("s3")
bucket = s3_resource.Bucket("my_first_bucket")

with open('~/test_file.txt', 'rb') as uploaded_data:
    bucket.put_object(Body=uploaded_data)
```

In the preceding example, the following applies:

1. We imported the `boto3` module that we installed previously.
2. Then, we specified a resource type that we wanted to interact with, which is `s3`, and assigned that to the `s3_resource` object.
3. We accessed `my_first_bucket` through the `Bucket()` method and assigned the returned value to the bucket variable.
4. Then, we opened a file using the `with` clause and named it `uploaded_data`. Notice that we opened the file as a binary data, using the `rb` flag.
5. Finally, we uploaded the binary data to our bucket using the `put_object()` method provided within the bucket space.

Deleting a bucket

To complete the CRUD operation for the bucket, the last thing we need to do is remove the bucket. This happens through calling the `delete()` method on our bucket variable, given that it already exists and we are referencing it by name, in the same manner that we created it and uploaded data to it. However, `delete()` may fail when the bucket is not empty. So, we will use the `bucket_objects.all().delete()` method to get all of the objects inside the bucket, then apply the `delete()` operation on them, and finally, delete the bucket:

```
import boto3
s3_resource = boto3.resource("s3")
bucket = s3_resource.Bucket("my_first_bucket")
bucket.objects.all().delete()
bucket.delete()
```

Summary

In this chapter, we learned how to install the Amazon **Elastic Compute Cloud** (**EC2**), and we learned about Boto3 and its installation. We also learned how to automate AWS S3 services.

In the next chapter, we will learn about the SCAPY framework, which is a powerful Python tool used to build and craft packets and send them on the wire.

17
Using the Scapy Framework

Scapy is powerful Python tool used to build and craft the packets then send them on the wire. You can build any type of network stream and send it on the wire. It can help you to test your network using different packet streams and manipulate the response returned from the source.

We will cover the following topics in this chapter:

- Understanding the Scapy framework
- Installing Scapy
- Generating packets and network streams using Scapy
- Capturing and replaying packets

Understanding Scapy

Scapy (`https://scapy.net`) is one of the powerful Python tools that is used to capture, sniff, analyze, and manipulate network packets. It can also build a packet structure of layered protocols and inject a wiuthib stream into the network. You can use it to build a wide number of protocols on top of each other and set the details of each field inside the protocol, or, better, let Scapy do its magic and choose the appropriate values so that each one can have a valid frame. Scapy will try to use the default values for packets if not overridden by users. The following values will be set automatically for each stream:

- The IP source is chosen according to the destination and routing table
- The checksum is automatically computed
- The source Mac is chosen according to the output interface
- The Ethernet type and IP protocol are determined by the upper layer

Scapy can be programmed to inject a frame into a stream and to resend it. You can, for example, inject a 802.1q VLAN ID into a stream and resend it to execute attacks or analysis on the network. Also, you can visualize the conversation between two endpoints and graph it using `Graphviz` and `ImageMagick` modules.

Scapy has its own **Domain-Specific Language** (**DSL**) that enables the user to describe the packet that he wants to build or manipulate and to receive the answer in the same structure. This works and integrates very well with Python built-in data types, such as lists and dictionaries. We will see in examples that the received packets from the network are actually a Python list, and we can iterate the normal list functions over them.

Installing Scapy

Scapy supports both Python 2.7.x and 3.4+, starting from Scapy version 2.x. However, for versions lower than 2.3.3, Scapy needs Python 2.5 and 2.7, or 3.4+ for versions after that. Since we already installed that latest Python version, it should be fine to run the latest version of Scapy without a problem.

Also, Scapy has an older version (1.x), which is deprecated and doesn't provide support for Python 3 and works only on Python 2.4.

Unix-based systems

To get the latest and greatest version, you need to use python pip:

```
pip install scapy
```

The output should look something like the following screenshot:

```
[root@AutomationServer ~]# pip install scapy
Collecting scapy
  Downloading https://files.pythonhosted.org/packages/68/01/b9943984447e7ea6f8948e90c1729b78
161c2bb3eef908430638ec3f7296/scapy-2.4.0.tar.gz (3.1MB)
    100% |████████████████████████████████| 3.1MB 256kB/s
Building wheels for collected packages: scapy
  Running setup.py bdist_wheel for scapy ... done
  Stored in directory: /root/.cache/pip/wheels/cf/03/88/296bf69fee1f9ec7a87e122da52253b65f30
67f6ea8719b473
Successfully built scapy
Installing collected packages: scapy
Successfully installed scapy-2.4.0
You are using pip version 9.0.3, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[root@AutomationServer ~]#
```

To verify that Scapy is installed successfully, access the Python console and try to import the `scapy` module into it. If no import error is reported back to the console then the installation completed successfully:

```
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import scapy
>>>
```

Some additional packages are required to visualize the conversation and to capture the packets. Use the following commands depending on your platform to install the additional packages:

Installing in Debian and Ubuntu

Run the following command to install additional packages:

```
sudo apt-get install tcpdump graphviz imagemagick python-gnuplot
python-cryptography python-pyx
```

Installing in Red Hat/CentOS

Run the following command to install additional packages:

```
yum install tcpdump graphviz imagemagick python-gnuplot python-
crypto python-pyx -y
```

You may need to install `epel` repository on a CentOS-based system and update the system if you don't find any of the preceding packages available in the main repository.

Windows and macOS X Support

Scapy is built and design to run on linux-based system. However it also can run on other operating systems. You can install and port it on both windows ported on both Windows and macOS, with some limitations on each platform. For a Windows-based system, you basically need to remove the WinPcap driver and use the Npcap driver instead (don't install both versions at the same time to avoid any conflict issues). You can read more about Windows installation at `http://scapy.readthedocs.io/en/latest/installation.html#windows`.

For macOS X, you will need to install some python bindings and use the libdnet and libpcap libraries. Full installation steps are available at `http://scapy.readthedocs.io/en/latest/installation.html#mac-os-x`.

Generating packets and network streams using Scapy

As we mentioned before, Scapy has its own DSL language, which is integrated with python. Also, you can access the Scapy console directly and start to send and receive packets directly from the Linux shell:

```
sudo scapy
```

The output of the preceding command is as follows:

```
[root@AutomationServer ~]# sudo scapy
WARNING: Cannot read wireshark manuf database
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: No route found for IPv6 destination :: (no default route?)
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

                    aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.4.0
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | We are in France, we say Skappee.
       scccccp///pSP///p          p//Y   | OK? Merci.
      sY/////////y  caa           S//P   |             -- Sebastien Chabal
       cayCyayP//Ya              pY/Ya   |
        sY/PsY////YCc           aC//Yp
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs
```

Notice there are a couple of warning messages about some missing *optional* packages, such as `matplotlib` and `PyX`, but this should be fine and won't affect the Scapy core functions.

We can start first by checking the supported protocols inside scapy. Run the `ls()` function to list all supported protocols:

```
>>> ls()
```

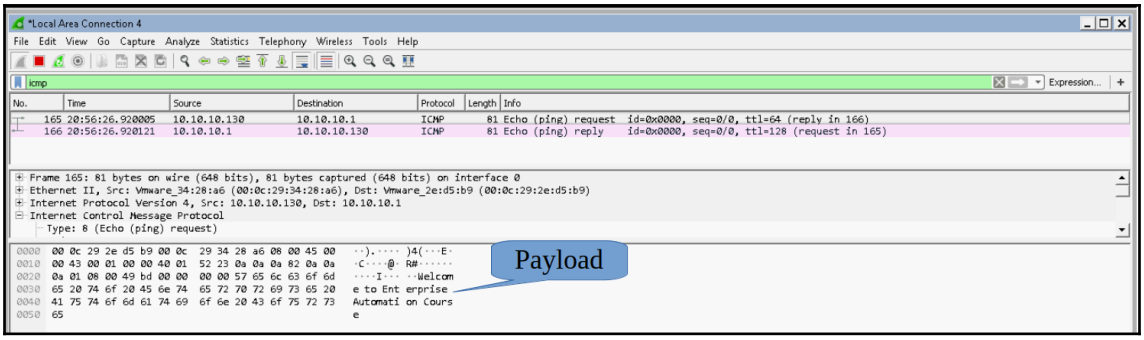
The output is quite lengthy and will span multiple pages if posted here, so you can take a quick look on the Terminal instead to check it.

Now let's develop hello world application and run it using SCAPY. The program will send a simple ICMP packet to server's gateway. I installed a Wireshark and configured it to listen to a network interface that will receive a stream from the automation server (which hosts Scapy).

Now, on the Scapy terminal, execute the following code:

```
>>> send(IP(dst="10.10.10.1")/ICMP()/"Welcome to Enterprise Automation
Course")
```

Return to Wireshark, and you should see the communication:



Let's analyze the command that Scapy executes:

- **Send**: This is a built-in function in Scapy **Domain Specific Language** (**DSL**) that instructs Scapy to send a single packet (and doesn't listen for any response back; it just sends one packet and exits).
- **IP:** Now, inside this class, we will start building packet layers. Starting with the IP layer, we need to specify the destination host that will receive the packet (in that case, we use the `dst` argument to specify the destination). Note also that we can specify the source IP in the `src` argument; however, Scapy will consult the host routing table and find the suitable source IP and put it in the packet. You can provide additional parameters, such as **time to live** (**TTL**), and Scapy will override the default one.

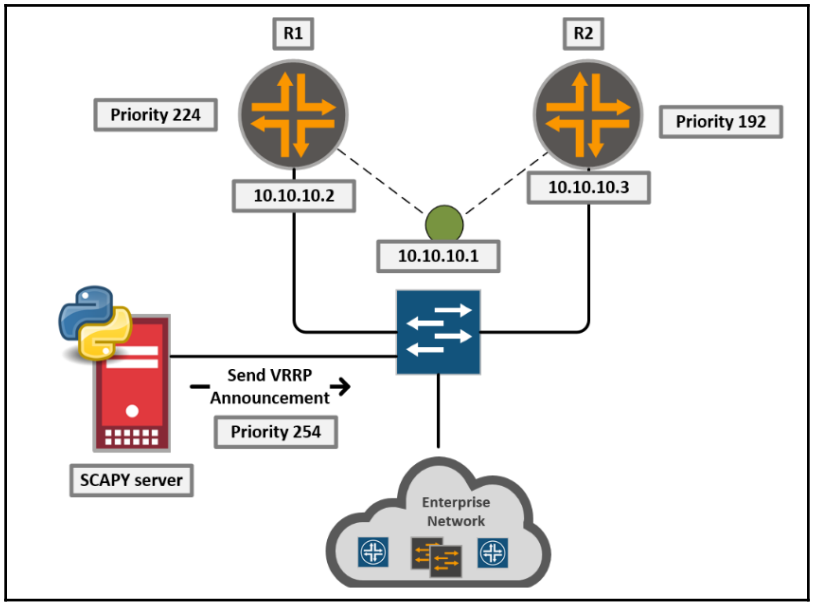
- / : Although it looks like the normal division operator used in Python, it's used in Scapy DSL to differentiate between packet layers and stack them over each other.
- **ICMP():** A built-in class used to create an ICMP packet with a default value. One of the values that could be provided to the function is the ICMP type, which determines the message type: `echo`, `echo reply`, `unreachable`, and so on.
- **Welcome to Enterprise Automation Course:** If a string is injected into the ICMP payload. Scapy will automatically convert it to a suitable format.

Note that we didn't specify the Ethernet layer in the stack and didn't provide any mac addresses (either source or destination). This is again filled by default in scapy to create a valid frame. It will automatically check the host ARP table and find the mac address for the source interface (and destination also, if it exists), them format then into an Ethernet frame.

A final thing to note before moving on to the next example is that you can use the same `ls()` function we used before to list all supported protocols to get the default values for each protocol, then set it to any other value when we call the protocol:

```
>>> ls(IP)
version    : BitField (4 bits)                   = (4)
ihl        : BitField (4 bits)                   = (None)
tos        : XByteField                          = (0)
len        : ShortField                          = (None)
id         : ShortField                          = (1)
flags      : FlagsField (3 bits)                 = (<Flag 0 ()>)
frag       : BitField (13 bits)                  = (0)
ttl        : ByteField                           = (64)
proto      : ByteEnumField                       = (0)
chksum     : XShortField                         = (None)
src        : SourceIPField                       = (None)
dst        : DestIPField                         = (None)
options    : PacketListField                     = ([])
>>>
```

Let's now do something more complex (and evil!). Assume we have two routers that form VRRP relationships between each other, and we need to to break this relationship to become the new master, or at least create a flapping issue in the network, as in the following topology:



Recall that routers configured to run VRRP join to multicast address (`255.0.0.18`) in order to receive the advertisements from other routers. The destination MAC address for the VRRP packet should contain the VRRP group number in last two numbers. Also it contains the router priority used in election process between routers. We will build a Scapy script that sends a VRRP announcement with a higher priority than is configured in the network. This will cause our Scapy server to be elected as the new master:

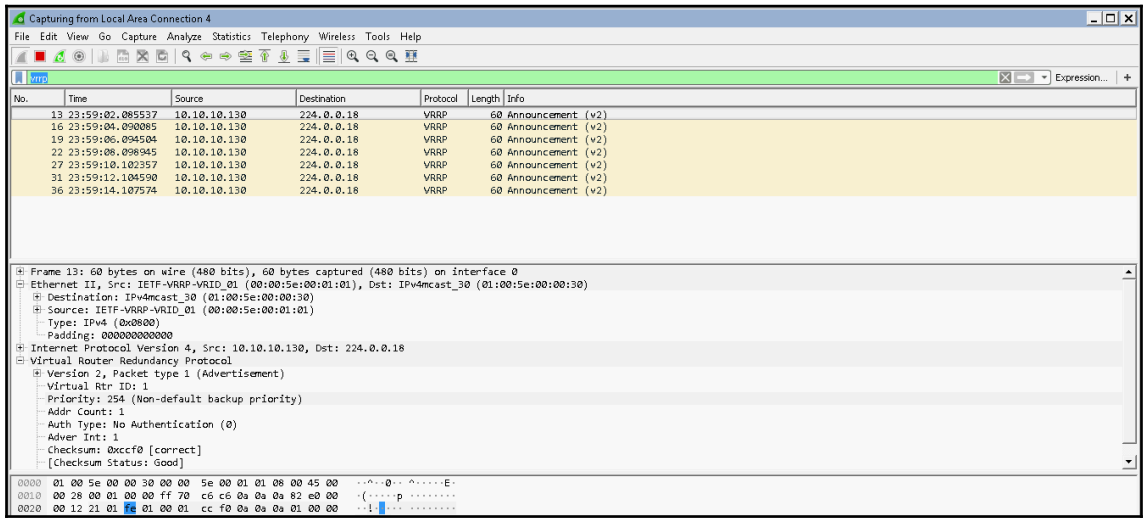
```
from scapy.layers.inet import *
from scapy.layers.vrrp import VRRP

vrrp_packet =
Ether(src="00:00:5e:00:01:01",dst="01:00:5e:00:00:30")/IP(src="10.10.10.130
", dst="224.0.0.18")/VRRP(priority=254, addrlist=["10.10.10.1"])
sendp(vrrp_packet, inter=2, loop=1)
```

In this example:

- First we imported some needed layers that we stacked over each other from the `scapy.layers` module. For example, the `inet` module contains the layers `IP()`, `Ether()`, `ARP()`, `ICMP()`, and so on.
- Also, we will need the VRRP layers, which could be imported from `scapy.layers.vrrp`.

- Second, we will build a VRRP packet and store it in the `vrrp_packet` variable. This packet contains the VRRP group number in the mac address inside ethernet frame . The multicast address will be inside the IP layer. Also we will configure a higher priority number inside the VRRP layer. That way we will have a valid VRRP announcement and router will accept it. We provided each layer with information such as the destination mac address (VRRP MAC + Group number) and the multicast IP (`225.0.0.18`).
- Finally, we used the `sendp()` function and provided it with a crafted `vrrp_packet`. The `sendp()` function will send a packet at layer 2, unlike the `send()` function, which we used in the previous example to send packets, but at layer 3. The `sendp()` function won't try to resolve the hostname like the `send()` function and will only operate at layer 2. Also, since we need to send this announcement continuously, we configured both `loop` and `inter` arguments to send announcements every 2 seconds.

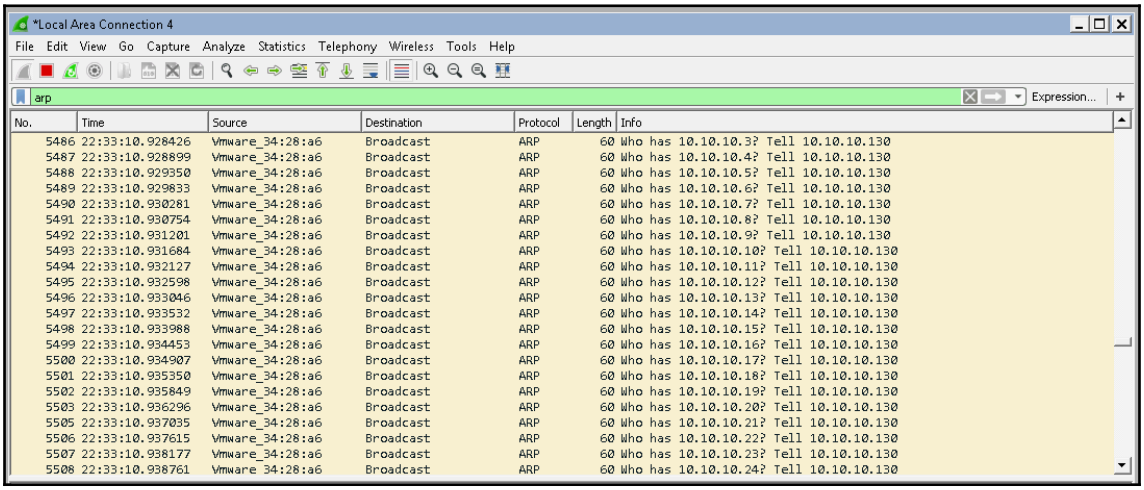
The script output is:



You can combine this attack with ARP poisoning and VLAN hopping attacks so you can change the mac address in the layer 2, switch to the Scapy server MAC address, and perform a **man in the middle** (**MITM**) attack.

Scapy also contains some classes that perform scan. For example, you can execute an ARP scan on the network range by using `arping()` and specifying the IP address in regex format inside it. Scapy will send an ARP request to all hosts on these subnets and inspect the reply:

```
from scapy.layers.inet import *
arping("10.10.10.*")
```



The script output is:

```
[root@AutomationServer ~]# python ping_arp.py
Begin emission:
Finished sending 256 packets.
*
Received 1 packets, got 1 answers, remaining 255 packets
  00:0c:29:2e:d5:b9 10.10.10.1
[root@AutomationServer ~]#
```

According to received packets, only one host is responding back to SCAPY meaning it's only host on the scanned subnet. The host mac and IP addresses are listed in the reply also

Capturing and replaying packets

Scapy has the ability to listen to the network interface and capture all incoming packets on it. It can write it on a `pcap` file in the same way that `tcpdump` works, but Scapy provides additional functions that can read and replay a `pcap` file, in the network again.

Starting with a simple packet replay, we will instruct Scapy to read a normal `pcap` file captured from the network (either using `tcpdump` or Scapy itself) and send it again to the network. This is very useful if we need to test the behavior of the network if a specific traffic pattern travels through it. For example, we may have a network firewall configured to block FTP communication. We can test the functionality of the firewall by hitting it with FTP data replayed from Scapy.

In this example, we have the FTP captured `pcap` file and we need to replay it to the network:

```
from scapy.layers.inet import *
from pprint import pprint
pkts = PcapReader("/root/ftp_data.pcap") #should be in wireshark-tcpdump
format

for pkt in pkts:
    pprint(pkt.show())
```

The `PcapReader()` will take the `pcap` file as an input and analyze it to get each packet alone and add it as an item inside the `pkts` list. Now we can iterate over the list and show each packet content.

The script output is:

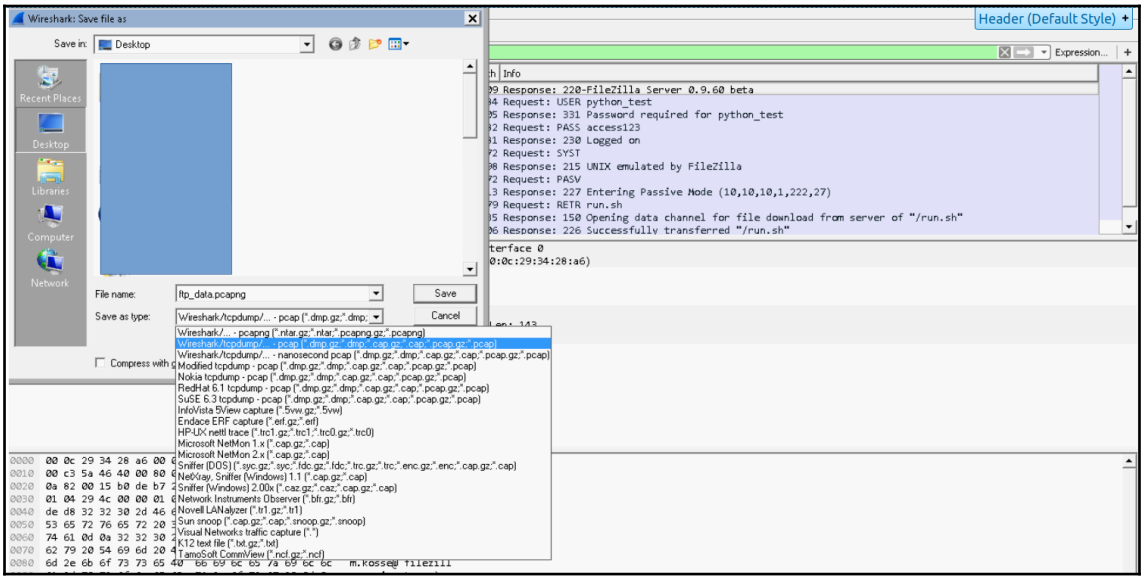
```
[root@AutomationServer ~]# python reading_pkt.py
###[ Ethernet ]###
  dst       = 00:0c:29:34:28:a6
  src       = 00:0c:29:2e:d5:b9
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 195
     id        = 27000
     flags     = DF
     frag      = 0
     ttl       = 128
     proto     = tcp
     chksum    = 0x0
     src       = 10.10.10.1
     dst       = 10.10.10.130
     \options   \
###[ TCP ]###
        sport     = ftp
        dport     = 45380
```

Also, you can get specific layer information via the `get_layer()` function that accesses packet layers. For example, if we were interested in getting the raw data without the header so we can build the transmitted file, we could use the following script to get the required data in hex then convert it to ASCII later:

```
from scapy.layers.inet import *
from pprint import pprint
pkts = PcapReader("/root/ftp_data.pcap") #should be in wireshark-tcpdump
format

ftp_data = b""
for pkt in pkts:
    try:
        ftp_data += pkt.get_layer(Raw).load
    except:
        pass
```

Notice that we have to surround the `get_layer()` method with a try-except clause as some layers don't contain the raw data (such as FTP control messages). Scapy will throw the error and the script will exit. Also, we can rewrite the script as an `if` clause that will add content to `ftp_data` only if the packet has the raw layer in it.

To avoid any errors while reading the `pcap` file, make sure you save (or export) your `pcap` file as Wireshark/tcpdump format, as shown here, and not the default format:



Injecting data inside packets

We can manipulate the packet and change its contents before replaying it back to the network. Since our packets are actually stored as items inside the list, we can iterate over those items and replace specific information. For example, we can change mac addresses, IP addresses, or add additional layers to each packet or for specific packets matching a condition. However, we should note that manipulating packets in specific layers such as the IP and TCP and changing the content will result in an invalid checksum for the whole layer and the receiver may drop the packet for that reason.

Scapy has an amazing feature (yes I know, I keep saying amazing many times but Scapy really is an awesome tool). It will automatically calculate the checksum for us based on the new content if we delete the original one in the `pcap` file.

So, we will modify the previous script and change a few packet parameters, then rebuild the checksum before sending the packets to the network:

```
from scapy.layers.inet import *
from pprint import pprint
pkts = PcapReader("/root/ftp_data.pcap") #should be in wireshark-tcpdump
format


p_out = []

for pkt in pkts:
    new_pkt = pkt.payload

    try:
        new_pkt[IP].src = "10.10.88.100"
        new_pkt[IP].dst = "10.10.88.1"
        del (new_pkt[IP].chksum)
        del (new_pkt[TCP].chksum)
    except:
        pass

    pprint(new_pkt.show())
    p_out.append(new_pkt)
send(PacketList(p_out), iface="eth0")
```

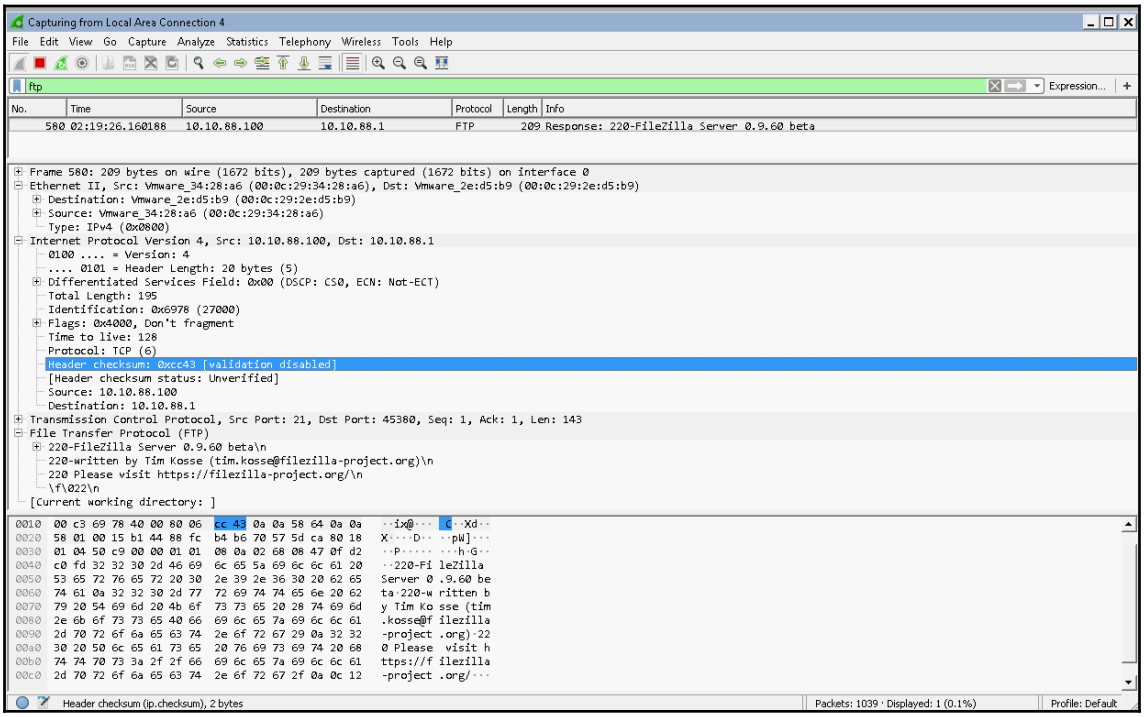
In the previous script:

- We used the `PcapReader()` class to read the content of the FTP `pcap` file and store the packets in a `pkts` variable.
- Then we iterated over the packet and assigned the payload to`new_pkt` so we could manipulate the content.
- Remember, the packet itself is considered as an object from the class. We can access the `src` and `dst` members and set them to any desired values. Here, we set the destination to the gateway and the source to a different value than the original packet.
- Setting a new IP value will invalidate the checksum, so we deleted both the IP and TCP checksum using the `del` keyword. Scapy will recalculate them again based on the new packet contents.
- Finally, we appended the `new_pkt` to the empty `p_out` list and sent it using the `send()` function. Notice that we can specify the exit interface in the send function or just leave it and Scapy will consult the host routing table; it will get the correct exit interface per packet.

The script output is:

```
[root@AutomationServer ~]# python manipulate_packets.py
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 195
  id        = 27000
  flags     = DF
  frag      = 0
  ttl       = 128
  proto     = tcp
  chksum    = None
  src       = 10.10.88.100
  dst       = 10.10.88.1
  \options   \
###[ TCP ]###
     sport     = ftp
     dport     = 45380
     seq       = 2298262710
     ack       = 1884773834
     dataofs   = 8
     reserved  = 0
     flags     = PA
     window    = 260
     chksum    = None
     urgptr    = 0
     options   = [('NOP', None), ('NOP', None), ('Timestamp', (40372295, 265470205))]
```

Also, if we still run the Wireshark in the gateway, we will notice that Wireshark captures the `ftp` packet stream with the checksum value set after recalculation:



Packet sniffing

Scapy has a built-in packet capture function called `sniff()`. By default, it will monitor all interfaces and capture all packets if you don't specify any filters or a certain interface:

```
from scapy.all import *
from pprint import pprint

print("Begin capturing all packets from all interfaces. send ctrl+c to
terminate and print summary")
pkts = sniff()

pprint(pkts.summary())
```

The script output is:

```
[root@AutomationServer ~]# python sniff_all.py
^CEther / IPv6 / UDP fe80::c1ec:5f5d:9e9b:c874:dhcpv6_client > ff02::1:2:dhcpv6_server / DHC
P6_Solicit / DHCP6OptElapsedTime / DHCP6OptClientId / DHCP6OptIA_NA / DHCP6OptClientFQDN / D
HCP6OptVendorClass / DHCP6OptOptReq
Ether / ARP who has 10.10.10.130 says 10.10.10.1 / Padding
Ether / ARP is at 00:0c:29:34:28:a6 says 10.10.10.130
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IPv6 / UDP fe80::c1ec:5f5d:9e9b:c874:dhcpv6_client > ff02::1:2:dhcpv6_server / DHCP6
_Solicit / DHCP6OptElapsedTime / DHCP6OptClientId / DHCP6OptIA_NA / DHCP6OptClientFQDN / DHC
P6OptVendorClass / DHCP6OptOptReq
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IP / TCP 10.10.10.1:49250 > 10.10.10.130:ssh PA / Raw
Ether / IP / TCP 10.10.10.130:ssh > 10.10.10.1:49250 A
```

You can of course provide filters and specific interfaces to monitor whether the condition is matched. For example, in the preceding output we can see a mix of ICMP, TCP, SSH, and DHCP traffic hitting all interfaces. If we're interested only in getting ICMP traffic on eth0, then we can provide the filter and `iface` arguments to sniff the function, and it will only filter all traffic and record only the ICMP:

```
from scapy.all import *
from pprint import pprint

print("Begin capturing all packets from all interfaces. send ctrl+c to
terminate and print summary")
pkts = sniff(iface="eth0", filter="icmp")

pprint(pkts.summary())
```

The script output is:

```
[root@AutomationServer ~]# python sniff_icmp_eth0.py
Begin capturing all packets from all interfaces. send ctrl+c to terminate and print summary
^CEther / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.10.10.1 > 10.10.10.130 echo-request 0 / Raw
Ether / IP / ICMP 10.10.10.130 > 10.10.10.1 echo-reply 0 / Raw
None
```

Notice how we capture only the ICMP communications on eth0 interfaces, and all other packets are discarded due to the filter applied on them. The *iface* value accepts a single interface that we used in the script or a list of interfaces to monitor them.

One of the advanced features of sniff is `stop_filter`, which is a Python function applied to each packet to determine if we have to stop the capture after that packet. For example, if we set `stop_filter = lambda x: x.haslayer(TCP)` then we will stop the capture once we hit a packet with a TCP layer. Also, the `store` option allows us to store the packets in the memory (which is by default enabled) or discard them after applying a specific function on each packet. This is a great feature if you're getting real-time traffic from the wire to SCAPY and don't want to write them to memory, if you set the store argument to false inside the sniff function, then SCAPY will apply any custom function you developed before (to get some information from packet for example or re-send them to different destination..etc) then won't store the original packet in the memory and will discard it. This will save some memory resources during sniffing.

Writing the packets to pcap

Finally, we can write our sniffed packets to a standard `pcap` file and open it with Wireshark as usual. This happens via a simple `wrpcap()` function that writes the list of packets to a `pcap` file. The `wrpcap()` function accepts two arguments—the first one is the full path to a file location, and the second is the packet list captured before using the `sniff()` function:

```
from scapy.all import *

print("Begin capturing all packets from all interfaces. send ctrl+c to
terminate and print summary")
pkts = sniff(iface="eth0", filter="icmp")

wrpcap("/root/icmp_packets_eth0.pcap",pkts)
```

Summary

In this chapter, we learned how to leverage the Scapy framework to build any type of packet containing any network layer and populated it with our values. Also, we saw how to capture packets on the interface and replay them.

18
Building a Network Scanner Using Python

In this chapter, we will build a network scanner that can identify the live hosts on the network and we will also expand it to include guessing the running operating system on each host and opened/closed ports. Usually, gathering this information requires multiple tools and some Linux ninja skills to get the required information but, using Python, we can build our own network scanner code that includes any tools and we can get a customized output.

The following topics will be covered in this chapter:

- Understanding the network scanner
- Building a network scanner with Python
- Sharing your code on GitHub

Understanding the network scanner

A network scanner is used to scan a provided range of network IDs in both layer 2 and layer 3. It can send requests and analyze responses for hundreds of thousands of computers. Also, you can expand its functionality to show some shared resources, via Samba and NetBIOS protocols, and the content of unprotected data on servers running sharing protocols. Another usage for the network scanner in penetration testing is when a white hat hacker tries to simulate an attack on network resources to find vulnerabilities and to evaluate company security. The final goal of the penetration test is to generate a report with all of the weaknesses in the target system so the origin point can reinforce and enhance security policies against the potential real attack.

Building a network scanner with Python

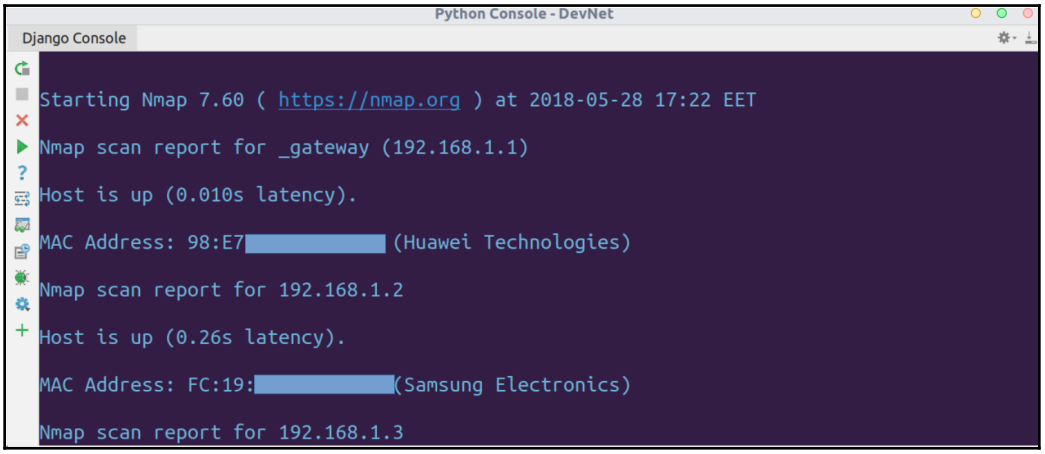
Python tools provide many native modules and support for working with sockets and TCP/IP in general. Additionally, Python can use the existing third-party commands available on the system to initiate the required scan and return the result. This can be done using the `subprocess` module that we discussed before, in `Chapter 9`, *Using the Subprocess Module*. A simple example is using Nmap to scan a subnet, as in the following code:

```
import subprocess
from netaddr import IPNetwork
network = "192.168.1.0/24"
p = subprocess.Popen(["sudo", "nmap", "-sP", network],
stdout=subprocess.PIPE)

for line in p.stdout:
    print(line)
```

In this example, we can see the following:

- At the beginning, we imported the `subprocess` module to be used in our script.
- Then, we defined the network that we want to scan with the `network` parameter. Notice that we used the CIDR notation, but we could use the subnet mask instead and convert that to CIDR notation using the Python `netaddr` module.
- The `Popen()` class inside `subprocess` is used to create an object that will send a regular Nmap command and scan the network. Notice that we added some flags, `-sP`, to tweak the Nmap operation and redirected the output to a special pipe created by `subprocess.PIPE`.
- Finally, we iterated over the created pipe and printed each line.

The script output is as follows:



Access to network ports on Linux requires root access, or your account must belong to a sudoers group in order to avoid any problems in the script. Also, the `nmap` package should be installed on the system prior to running the Python code.

This is a simple Python script and we can use the Nmap tool directly instead of using it inside Python. However, wrapping the Nmap (or any other system command) with Python code gives us the flexibility of tailoring the output and customizing it in any way. In the next section, we will enhance our script and add more functionality to it.

Enhancing the code

Although the output of Nmap gives us an overview of the live hosts on the scanned network, we can enhance it and have a better output view. For example, I need to know the total number of hosts at the beginning of the output, then the IP address, MAC address, and MAC vendor for each one, but in tabular form, so I can easily locate any host and all of the information associated with it.

For that reason, I will design a function and name it `nmap_report()`. This function will take the standard output generated from the `subprocess` pipe and will extract the required information and format it in table format:

```
def nmap_report(data):
    mac_flag = ""
    ip_flag = ""
    Host_Table = PrettyTable(["IP", "MAC", "Vendor"])
    number_of_hosts = data.count("Host is up ")

    for line in data.split("\n"):
        if "MAC Address:" in line:
            mac = line.split("(")[0].replace("MAC Address: ", "")
            vendor = line.split("(")[1].replace(")", "")
            mac_flag = "ready"
        elif "Nmap scan report for" in line:
            ip = re.search(r"Nmap scan report for (.*)", line).groups()[0]
            ip_flag = "ready"


        if mac_flag == "ready" and ip_flag == "ready":
            Host_Table.add_row([ip, mac, vendor])
            mac_flag = ""
            ip_flag = ""

    print("Number of Live Hosts is {}".format(number_of_hosts))
    print Host_Table
```

Starting with the easiest part, we can get the number of live hosts by counting the `Host is up` occurrences in the passed output and assigning this to the `number_of_hosts` parameter.

Secondly, Python has a nice module called `PrettyTable` which can create a text table and handle the cell sizing according to data inside it. The module accepts the table headers as a list and uses the `add_row()` function to add rows to the created table. So, the first thing is to import this module (after installing it, if it's not already installed). In our example, we will pass a list of three items (`IP`, `MAC`, `Vendor`) to the `PrettyTable` class (imported from the `PrettyTable` module) to create the table headers.

Now, to fill up this table, we will split the output on `\n` (carriage return). The split result will be a list, that we can iterate over to grab specific information such as MAC address and IP address. We used a few splitting and replace hacks to extract the MAC address alone. Also, we used the regular expression `search` function to get the IP address portion (or the hostname if DNS is enabled) from the output.

Finally, we added this information to the created `Host_Table` and continued to iterate over the next line.

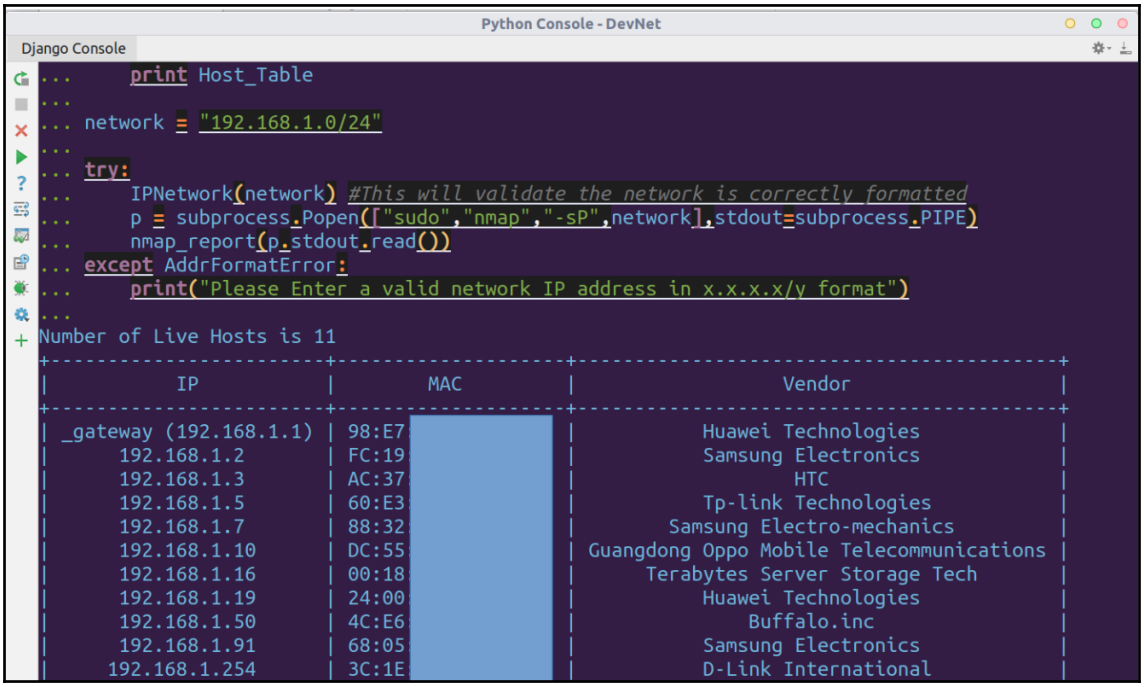
Following is the full script:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import subprocess
from netaddr import IPNetwork, AddrFormatError
from prettytable import PrettyTable
import re


def nmap_report(data):
    mac_flag = ""
    ip_flag = ""
    Host_Table = PrettyTable(["IP", "MAC", "Vendor"])
    number_of_hosts = data.count("Host is up ")

    for line in data.split("\n"):
        if "MAC Address:" in line:
            mac = line.split("(")[0].replace("MAC Address: ", "")
            vendor = line.split("(")[1].replace(")", "")
            mac_flag = "ready"
        elif "Nmap scan report for" in line:
            ip = re.search(r"Nmap scan report for (.*)", line).groups()[0]
            ip_flag = "ready"


        if mac_flag == "ready" and ip_flag == "ready":
            Host_Table.add_row([ip, mac, vendor])
            mac_flag = ""
            ip_flag = ""

    print("Number of Live Hosts is {}".format(number_of_hosts))
    print Host_Table

network = "192.168.1.0/24"
```

```
try:
    IPNetwork(network)
    p = subprocess.Popen(["sudo", "nmap", "-sP", network],
stdout=subprocess.PIPE)
    nmap_report(p.stdout.read())
except AddrFormatError:
    print("Please Enter a valid network IP address in x.x.x.x/y format")
```

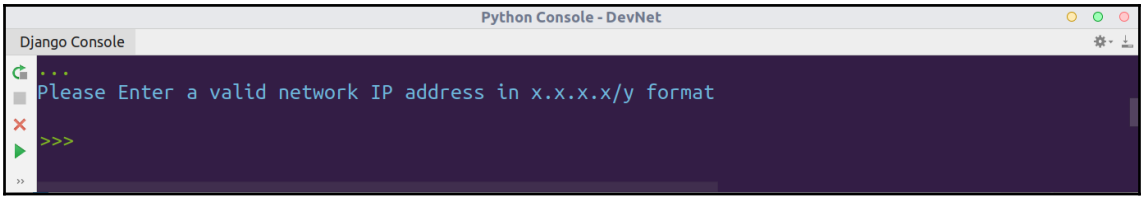
Notice we also added a pre-check to the `subprocess` command using the `netaddr.IPNetwork()` class. This class will validate whether the network is correctly formatted *before* executing the `subprocess` command, otherwise the class will raise an exception which should be handled by the `AddrFormatError` exception class and will print a customized error message to user.

The script output is:



Now, if we change the network to an incorrect value (either the subnet mask is wrong or the network ID is not valid), the `IPNetwork()` class will throw an exception and this error message will be printed:

```
network = "192.168.300.0/24"
```

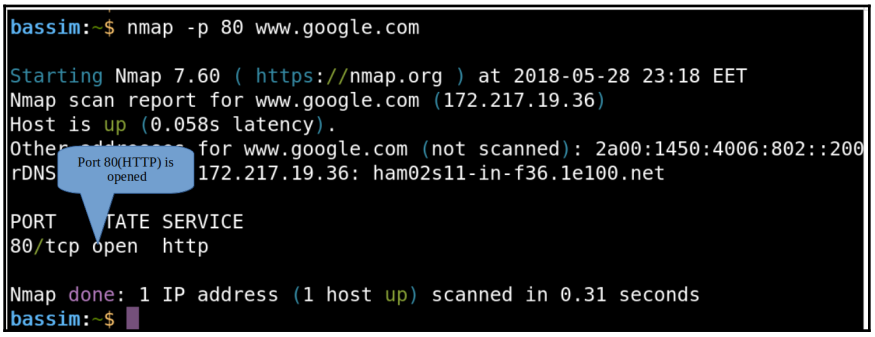


Scanning the services

Running services on a host machine typically open a port in the operating system and start listening to it in order to accept incoming TCP communication and start the three-way handshake. In Nmap, you can send an SYN packet on a specific port and, if the host responds with SYN-ACK, then the service is running and listening to the port.

Let's test the HTTP port, for example in `google.com`, using `nmap`:

```
nmap -p 80 www.google.com
```



We can use the same concept to discover the running services on the router. For example, the router that runs the BGP daemon will listen to port `179` for open/update/keep alive/notification messages. If you want to monitor the router, then the SNMP service should be enabled and should listen to incoming SNMP get/set messages. The MPLS LDP will usually listen to `646` for establishing a relationship with other neighbors. Here is a list of common services running on the router and their listening ports:

Service	**Listening port**
FTP	`21`
SSH	`22`
TELNET	`23`
SMTP	`25`
HTTP	`80`
HTTPS	`443`
SNMP	`161`
BGP	`179`
LDP	`646`
RPCBIND	`111`
NETCONF	`830`
XNM-CLEAR-TEXT	`3221`

We can create a dictionary with all of these ports and scan them using `subprocess` and Nmap. Then we use the returned output to create our table, which lists the open and closed ports for each scan. Also, with some additional logic, we can try to correlate information to guess the operating system type of the device function. For example, if the device is listening to port `179` (BGP port), then the device is most likely a network gateway and, if it listens to `389` or `636`, then the device is running an LDAP application and could be the company active directory. This will help us to create the proper attack against the device during the pen testing.

Without further ado, let's quickly put our idea and notes in the following script:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from prettytable import PrettyTable
import subprocess
import re
```

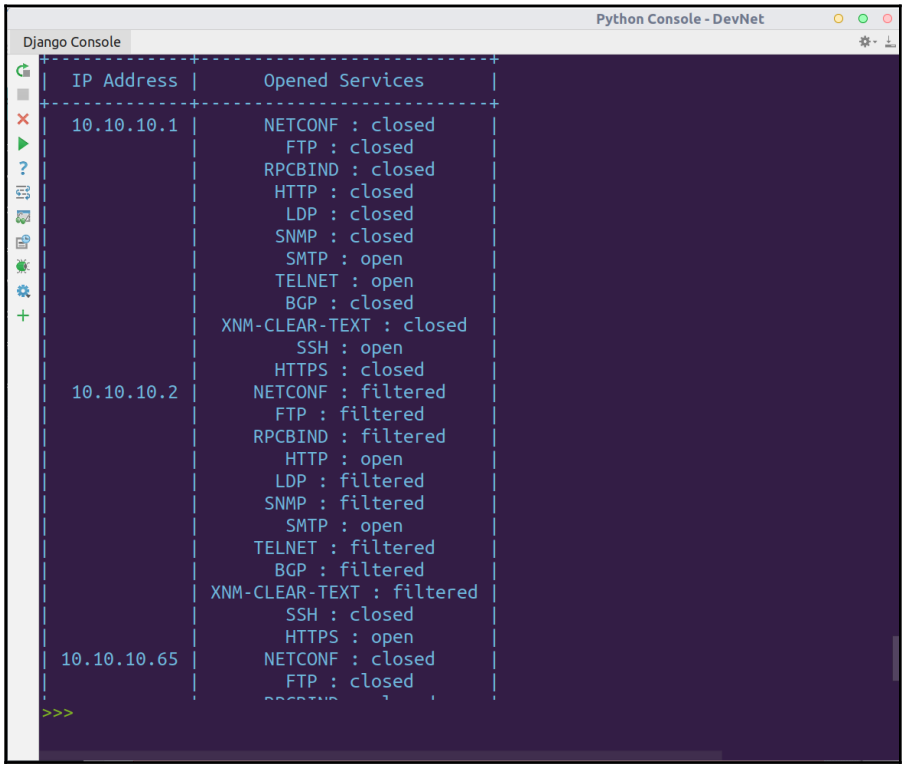
```
def get_port_status(port, data):
    port_status = re.findall(r"{0}/tcp (\S+) .*".format(port), data)[0]
    return port_status

Router_Table = PrettyTable(["IP Address", "Opened Services"])
router_ports = {"FTP": 21,
                "SSH": 22,
                "TELNET": 23,
                "SMTP": 25,
                "HTTP": 80,
                "HTTPS": 443,
                "SNMP": 161,
                "BGP": 179,
                "LDP": 646,
                "RPCBIND": 111,
                "NETCONF": 830,
                "XNM-CLEAR-TEXT": 3221}


live_hosts = ["10.10.10.1", "10.10.10.2", "10.10.10.65"]


services_status = {}
for ip in live_hosts:
    for service, port in router_ports.iteritems():
        p = subprocess.Popen(["sudo", "nmap", "-p", str(port), ip],
stdout=subprocess.PIPE)
        port_status = get_port_status(port, p.stdout.read())
        services_status[service] = port_status

    services_status_joined = "\n".join("{} : {}".format(key, value) for
key, value in services_status.iteritems())

    Router_Table.add_row([ip, services_status_joined])


print Router_Table
```

In this example, we can see the following:

- We developed a function named `get_port_status()` to take the Nmap port scanning result and to search for the port status (open, closed, filtered, and so on) using the regular expression inside the `findall()` function. It returns the port status result.

- Then, we added services ports mapped to the service name inside the `router_ports` dictionary, so we could access any port value using the corresponding service name (dictionary key). Also, we defined the router hosts' IP addresses inside the `live_hosts` list. Note that we can use the `nmap` with the `-sP` flag to get the live hosts, as we did before in a previous script.
- Now, we can iterate over each IP address in the `live_hosts` list and execute the Nmap to scan each port in the `router_ports` dictionary. This requires a nested `for` loop, so for each device we iterate over a list of ports and so on. The result will be added to the `services_status` dictionary—the service name is a dictionary key while the port status is the dictionary value.
- Finally, we will add the result to `Router_Table` created using the `prettytable` module to get a nice-looking table.

The script output is as follows:



Sharing your code on GitHub

GitHub is a place where you can share your code and collaborate with others on a common project using Git. Git is a source version control platform invented and created by Linus Trovalds, who started Linux but had a problem maintaining Linux development with a large number of developers contributing to it. He created a de-centralized version control where anyone could get the entire code (called cloning or forking), make changes, then push them back to the central repository to be merged with other developers' code. Git became the preferred method for many developers to work together on projects. You can learn how to code in Git interactively with this 15-minute course offered by GitHub: `https://try.github.io`.

GitHub is the website that hosts those projects, which is versioned using Git. It's like a developer social media platform, where you can track the code development, write a wiki, or raise an issue/bug report and get developer feedback on it. People on the same project can discuss the project progress and share code together to build a better and faster software. Also, some companies consider your code and repositories—shared in your account at GitHub—as an online resume that measures your skills and how you code in languages of interest.

Creating an account on GitHub

The first thing to do before sharing your code or downloading other codes is to create your account.

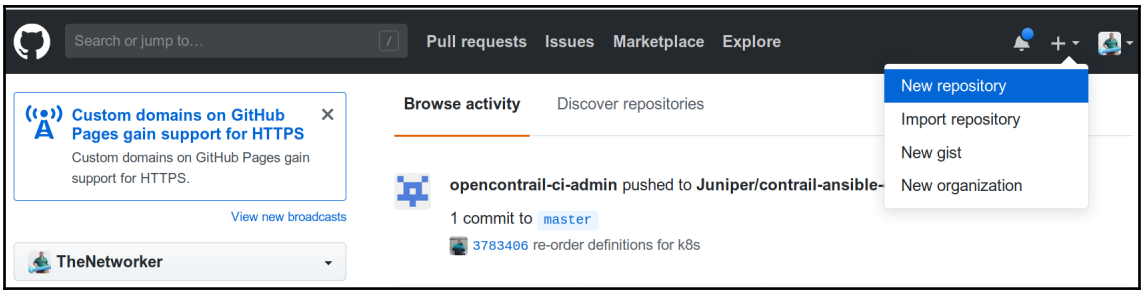
Head to `https://github.com/join?source=header-home` and choose a username, password, and email address, then click on the green **Create an account** button.

The second thing to do is to choose your plan. By default, the free plan is fine as it gives you unlimited public repositories and you can push any code developed in any languages you like. However, the free plan doesn't make your repository private and allows others to search for and download it. It's not a deal breaker if you're not working on secret or commercial projects in your company, however you need to make sure that you don't share any sensitive information, such as passwords, tokens, or public IP addresses in the code.

Creating and pushing your code

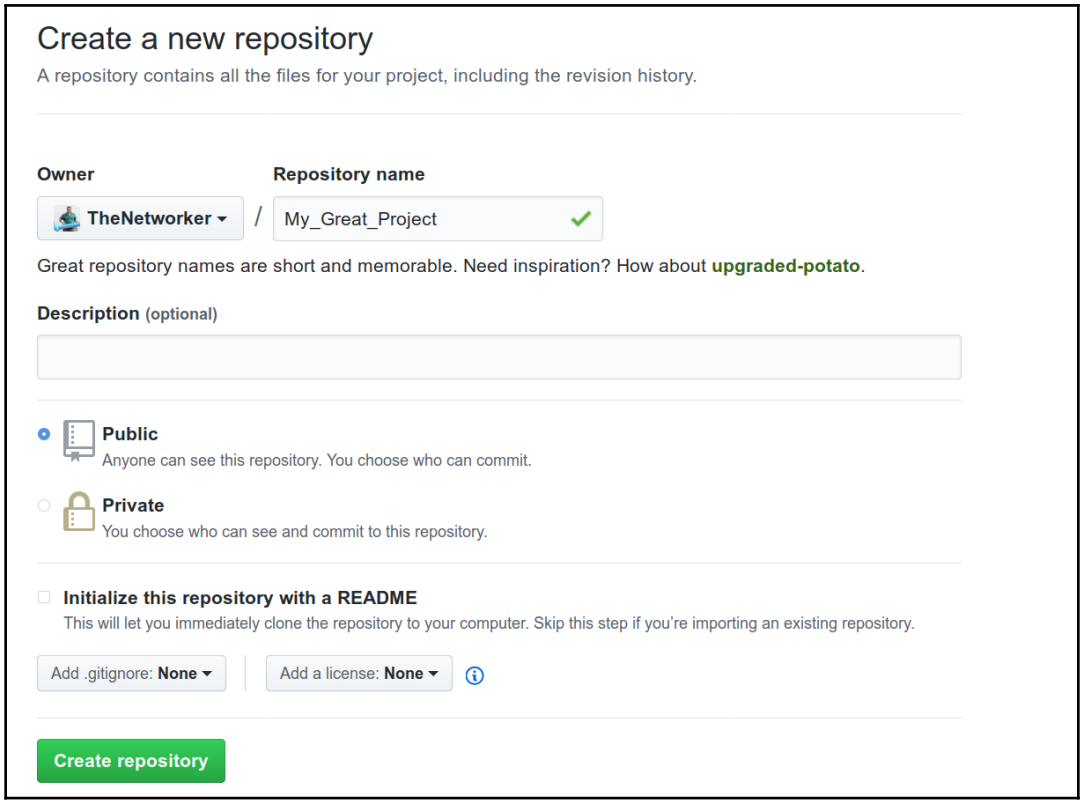
Now we're ready to share the code with others. The first thing after creating your GitHub account is to create a repository to host your files. Usually, you create one repository per project (not per file) and it contains project assets and files related to each other.

Click on the **+** icon in the top-right, just beside your profile picture, to create a new repository:

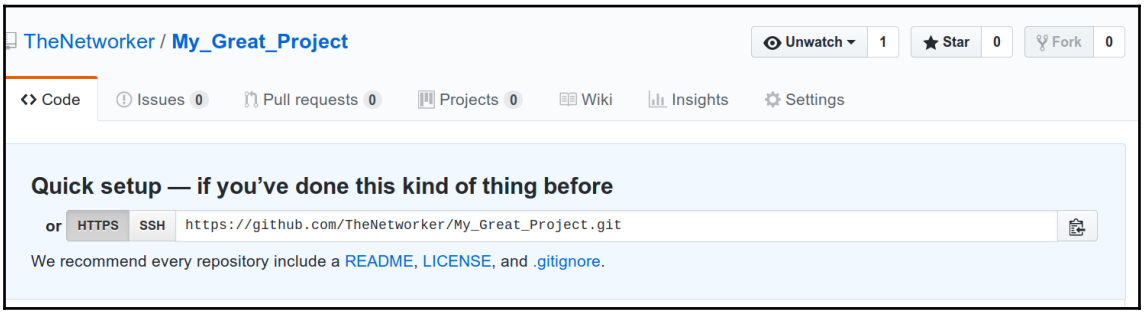


You will be redirected to a new page where you can enter your repository name. Notice that you can choose any you like, but it shouldn't conflict with other repository in your profile. Also, you will be give a unique URL for this repo so anyone can access it. You can set the repo settings, such as whether it is public or private (only for paid plans), and if you want to initialize it with a README file. This file is written using **markdown** text formatting that includes information about your project, and steps for other developers to follow if they use your project.

Finally, you will have an option to add a `.gitignore` file where you tell Git to ignore tracking a certain type of file in your directory, such as logs, `pyc`, compiled files, video, and so on:

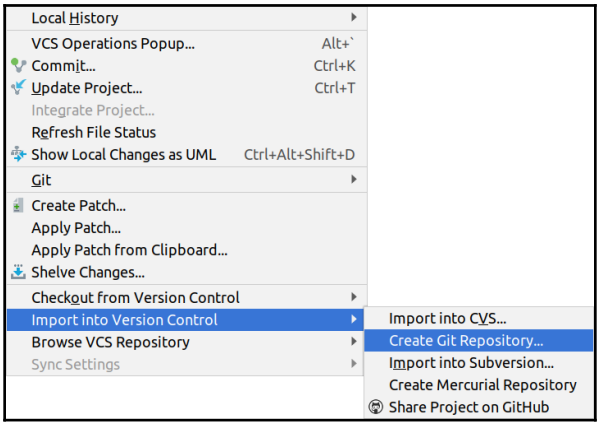


In the end, your repo is created and you will be given a unique URL for it. Note this URL down as we will use it later when pushing files to it:

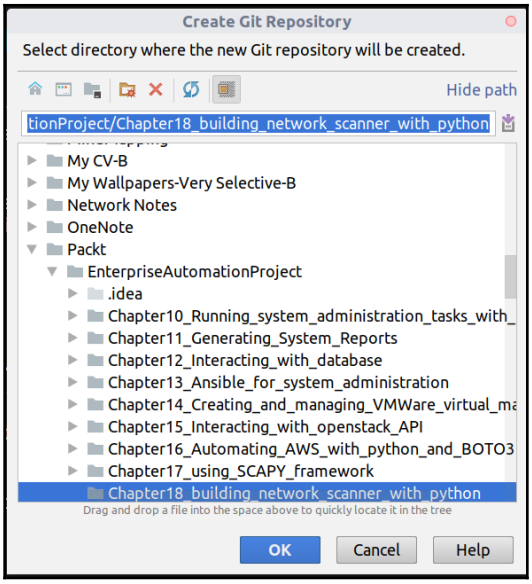


Now it's time to share your code. I will use the integrated Git functionality inside PyCharm to do the job although you can do the same steps in CLI. Also, there are many other GUI tools available (including one from GitHub itself) that can manage your GIT repo. I highly recommend that you do the Git training provided by GitHub (`https://try.github.io`) before following these steps:

1. Go to **VCS | Import into Version Control | Create Git Repository**:

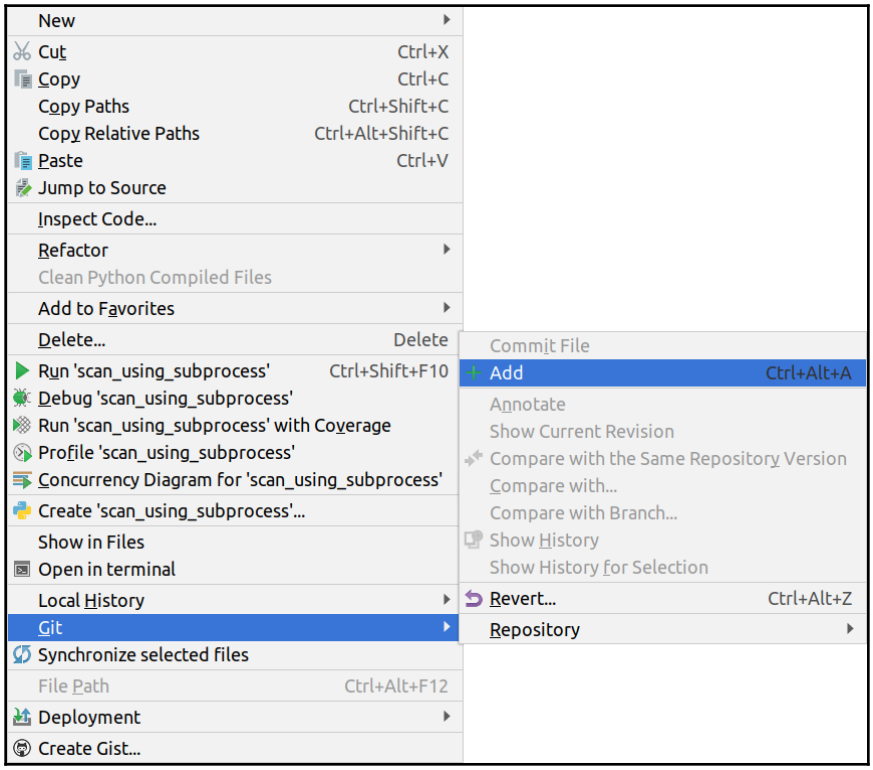


2. Choose the folder where your project files are stored locally:



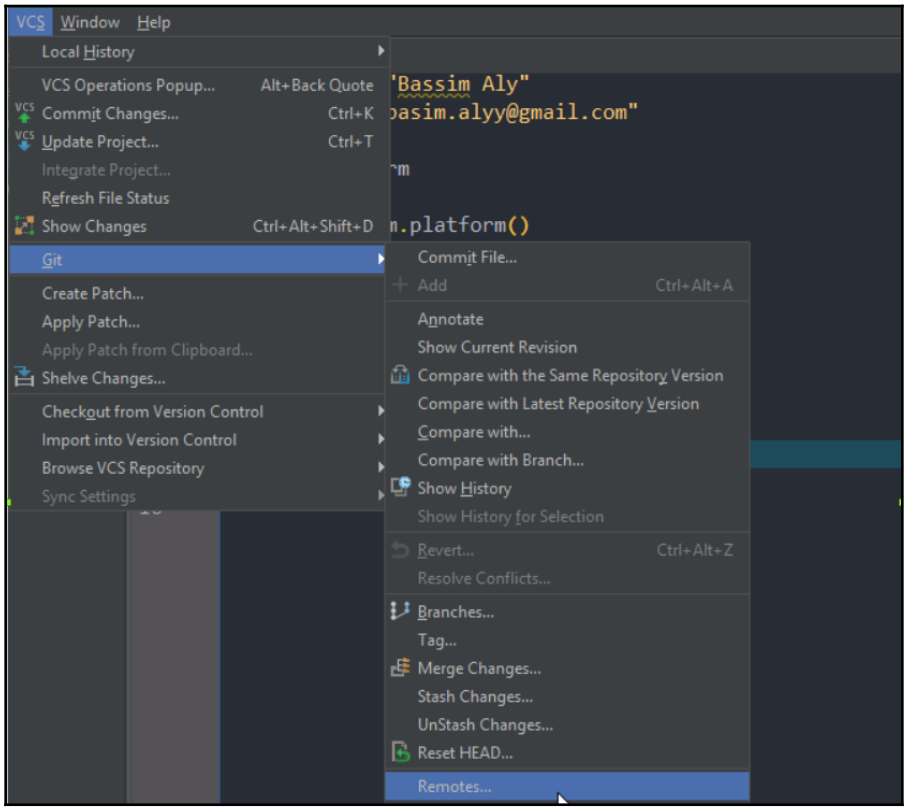
This will create a local Git repo in the folder.

3. Highlight all files that need to be tracked in the sidebar and right-click on them, then choose **Git** | **Add**:

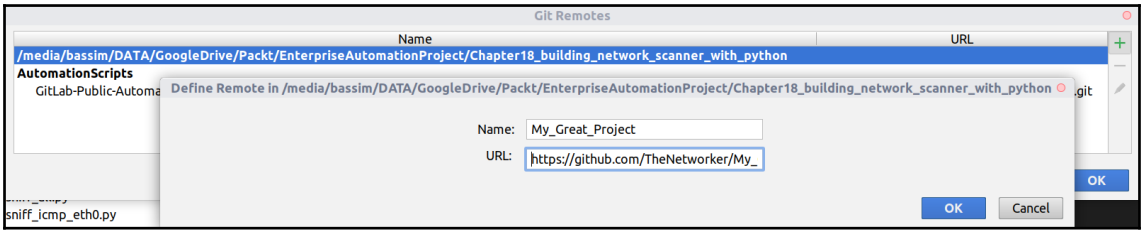


PyCharm uses file color code to indicate the type of file tracked in Git. When the files are not tracked, it will color them red and when the files are added to Git, it will color them green. This allows you to easily know file status without running commands.

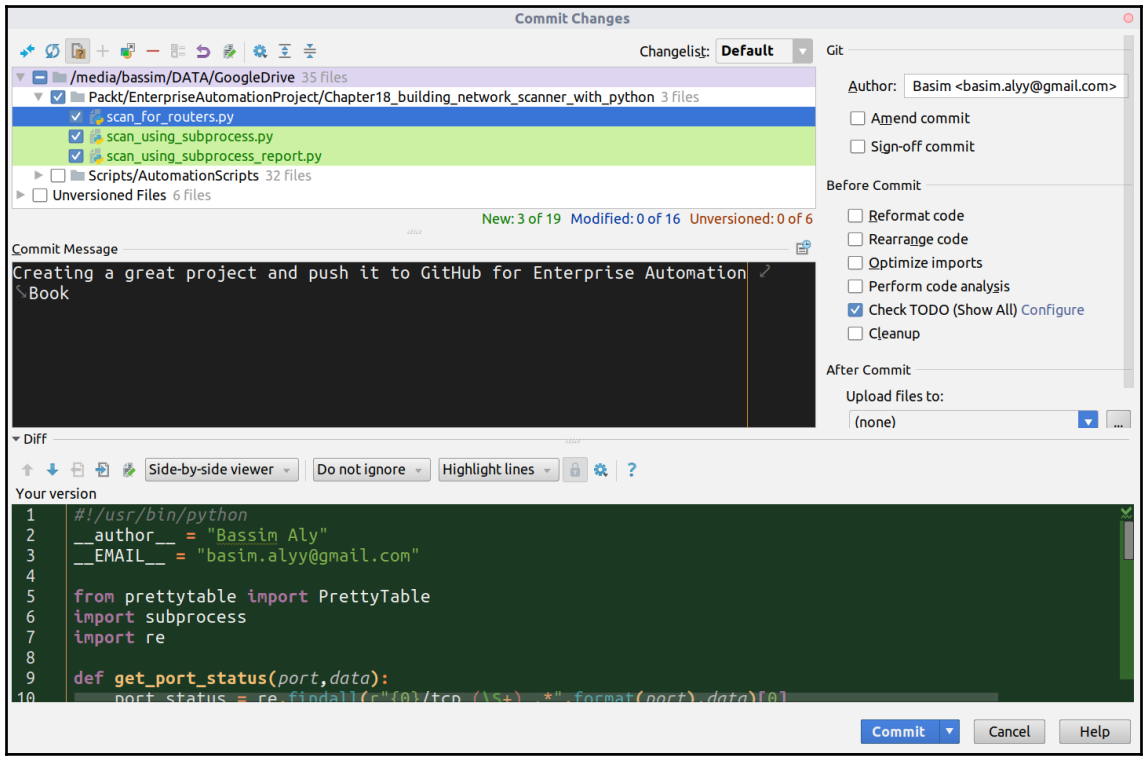
4. Define the remote repository in GitHub that will be mapped to the local repository by going to **VCS** | **Git** | **Remotes**:



5. Enter the repo name and the URL you noted down when we created the repo; click **OK** twice to exit the window:

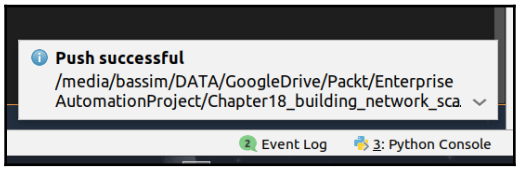


6. The final step is to commit your code. Go to **VCS** | **Git** | **Commit** and from the opened popup window, select your tracked files, enter a descriptive message in the **Commit Message** section, and instead of hitting **Commit**, click on the small arrow beside it and choose **Commit and Push**. A dialog box might be opened telling you that your **Git user Name Is Not Defined**. Just enter your name and email and make sure the **Set properties globally** box is ticked and hit **Set and Commit**:

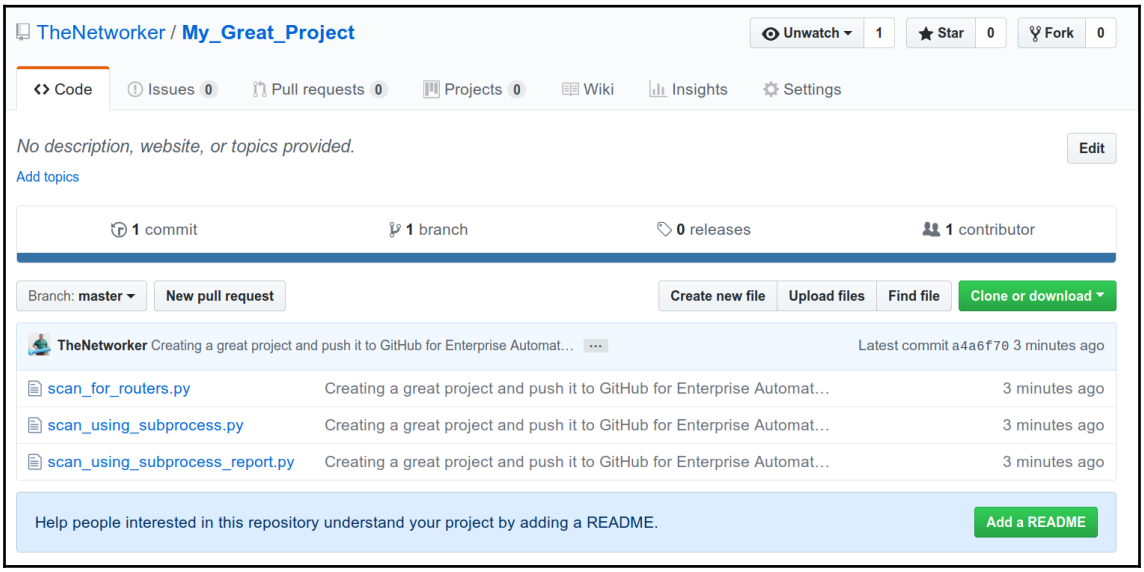


The PyCharm gives you an option to push to Gerrit for code review. If you have one, you can also share your files in it. Otherwise, click on **Push**.

A notification message will appear telling you the push completed successfully:



You can refresh your GitHub repo URL from the browser and you will see all your files stored in it:



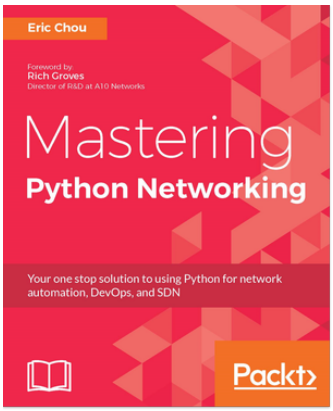
Now, whenever you make any change in the code inside the tracked files and commit, the changes will be tracked and added to the versioning system and will be available in GitHub for other users to download and comment on.

Summary

In this chapter, we built our network scanner, which can be used during authorized penetration testing, and learned how to scan different services and applications running on the device to detect their type. Also, we shared our code to GitHub so that we could keep different versions of our code and also allow other developers to use our shared code and enhance it, then share it again with others.

Other Books You May Enjoy

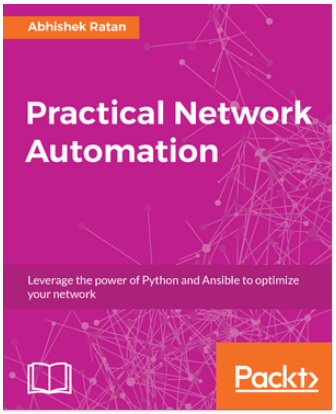
If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Python Networking
Eric Chou

ISBN: 978-1-784397-00-5

- Review all the fundamentals of Python and the TCP/IP suite
- Use Python to execute commands when the device does not support the API or programmatic interaction with the device
- Implement automation techniques by integrating Python with Cisco, Juniper, and Arista eAPI
- Integrate Ansible using Python to control Cisco, Juniper, and Arista networks
- Achieve network security with Python
- Build Flask-based web-service APIs with Python
- Construct a Python-based migration plan from a legacy to scalable SDN-based network.



Practical Network Automation
Abhishek Ratan

ISBN: 978-1-78829-946-6

- Get the detailed analysis of Network automation
- Trigger automations through available data factors
- Improve data center robustness and security through specific access and data digging
- Get an Access to APIs from Excel for dynamic reporting
- Set up a communication with SSH-based devices using netmiko
- Make full use of practical use cases and best practices to get accustomed with the various aspects of network automation

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y