Network Design

Gonçalo Martins

Class Notes Support

First Edition: October 2020

www.engredu.com

Contents

1 Python Program Concepts 1

1.1 Environment Setup 2

1.2 Getting & Installing Python 2

1.3 Running Python 3

1.4 Python Fundamental Concepts 4

2 Python for Network Engineers 15

2.1 Encoding & Decoding 15

2.2 Shifting Bytes 18

2.3 Struct Package 20

2.4 Other Functions 25

3 Python Socket Vocabulary 27

3.1 Creating Sockets 28

3.2 Sending Data 29

3.3 Receiving Data 30

3.4 Socket Objects 31

3.5 Socket Services 35

4 Programming with Sockets 37

4.1 Working with UDP sockets 39

4.2 Working with TCP sockets 43

5 Concurrency 47

5.1 Multithreading Server 49

5.2 Event Driven Server 57

6 Packet Analyzer 64

6.1 Basic Sniffer 66

6.2 Turtle Sniffer 72

A Basic Sniffer Code 94

B Turtle Sniffer Code 96

1

Python Program Concepts

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. This chapter provides enough understanding on Python programming language in order to use it to study and program network concepts.

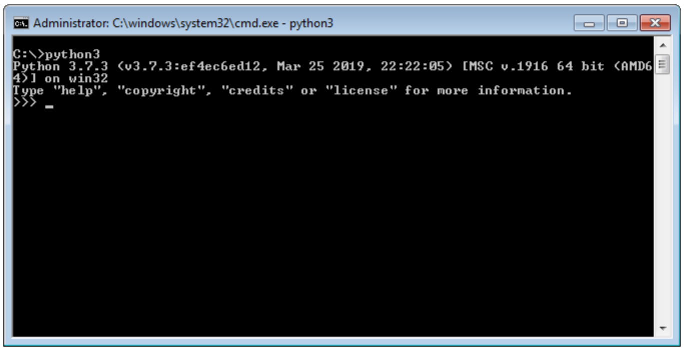
To take the best out of these notes, you should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages is a plus.

Reference Link

tutorialspoint.com - Learn Python 3

1.1 Environment Setup

We will be using python 3 throughout the chapters examples. Open a terminal window and type "python3" to find out if it is already installed and which version is installed.



```
C:\>python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Figure 1.1: *Windows Terminal*

1.2 Getting & Installing Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python `https://www.python.org/`

You can download Python documentation from `https://www.python.org/doc/`. The documentation is available in HTML, PDF, and Postscript formats.

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

1.3 Running Python

There are three different ways to start Python.

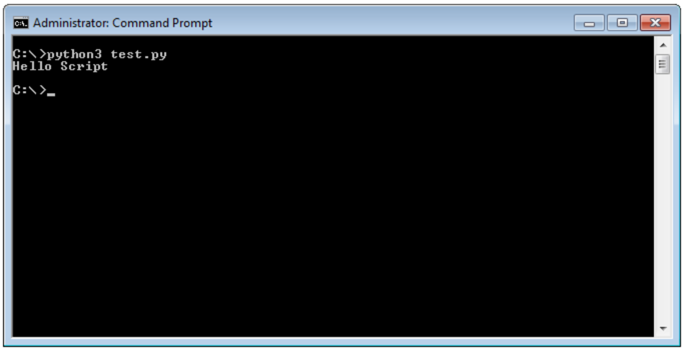
Command Line

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter python the command line (Figure 1.1) and start coding right away in the interactive interpreter.

Script from Command Line

A Python script can be executed at command line by invoking the interpreter on your application, as in the Figure 1.2



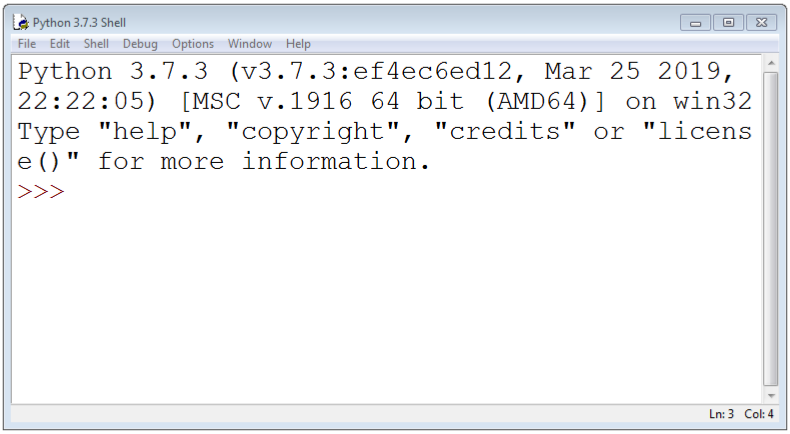
```
C:\>python3 test.py
Hello Script

C:\>_
```

Figure 1.2: *Run Script from Command Line*

Integrated Development Environment (IDE)

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

All the examples given in subsequent chapters are executed with Python 3.7.3 version available on Windows using IDLE IDE.



```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019,
22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "licens
e()" for more information.
>>>
```

Figure 1.3: *IDLE*

1.4 Python Fundamental Concepts

Assuming that you have some exposure in the past to any type of programming language, let's review some programming concepts using python.

Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Data Types

Python has five standard data types

- Numbers: Python 3 supports three different numerical types.

```
>> number = 5      #int
>> number = 0xFF   #int(hex)
>> number = 5.5    #float
>> number = 1 + 5j #complex
```

- String: a set of characters represented in the quotation marks.

```
>> str = "String Variable"
```

- List: contains items separated by commas and enclosed within square brackets.

```
>> list = ['abcd', 786, 2.23, 'john', 70.2]
>> tinylist = [123, 'john']
```

- Tuple: consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

```
>> tuple = ('abcd', 786, 2.23, 'john', 70.2)
>> tinytuple = (123, 'john')
```

- Dictionary: are kind of hash table type.

```
>> dict = {'name':'john',
           'code':6734,
           'dept':'sales'}
```

List vs Tuple

The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated. Tuples can be thought of as read-only lists.

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

```
>> number = 5    #int
>> str(number)   #int to string
>> hex(number)   #int to hex string
>> float(number) #int to float
>> str = "10"    #string
>> int(str)      #string to int
```

Basic Operations

Operators are the constructs, which can manipulate the value of operands. Consider the expression $4 + 5 = 9$. Here, 4 and 5 are called the operands and + is called the operator.

Python language supports the following types of operators:

Arithmetic Operators

```
>> a = 10
>> b = 20
>> a + b  #addition
>> a - b  #subtraction
>> a * b  #multiplication
>> a / b  #division
>> a % b  #modulus
>> a**b   #exponent
>> a//b   #floor division
```

Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

```
>> a = 10
>> b = 20
>> (a == b)  #equal
>> (a != b)  #not equal
>> (a > b)   #greater than
>> (a < b)   #less than
>> (a >= b)  #greater than or equal
>> (a <= b)  #less than or equal
```

Assignment Operators

```
>> a = 10
>> b = 20
>> c = a + b  #assign
>> c += a     #c = c + a
>> c *= a     #c = c * a
>> c /= a     #c = c / a
>> c %= a     #c = c % a
>> c **= a    #c = c ** a
>> c //= a    #c = c // a
```

Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation.

```
>> a = 60   # 0011 1100
>> b = 13   # 0000 1101
>> (a & b)  # 0000 1100
>> (a | b)  # 0011 1101
>> (a ^ b)  # 0011 0001
>> (~a)     # 1100 0011
>> a << 2   # 1111 0000
>> a >> 2   # 0000 1111
```

Python's built-in function bin() can be used to obtain binary representation of an integer number.

```
>> a = 60   # 0011 1100
>> bin(a)
'0b111100'
```

Logical Operators

Assume variable a holds True and variable b holds False

```
>> (a and b)    #returns False
>> (a or b)     #returns True
>> Not(a and b) #returns True
```

Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators.

```
>> x in y
>> x not in y
```

Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators.

```
>> x is y
>> x is not y
```

Decision Makers

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.

Following is the general form of a typical decision making structure found in most of the programming languages.

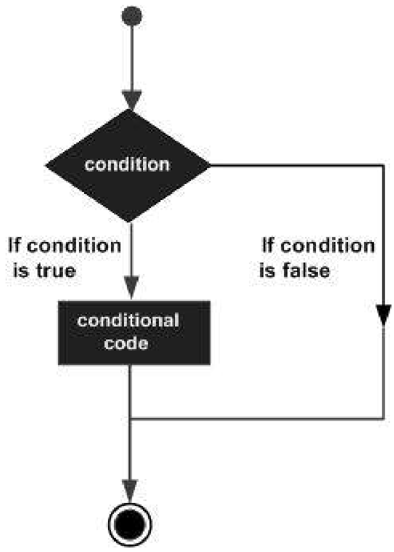


Figure 1.4: *Decision Making*

```
>> var = 100
>> if var == 100:
...   print("True")
...else:
...   print("False")
```

Loops

In general, statements are executed sequentially The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement

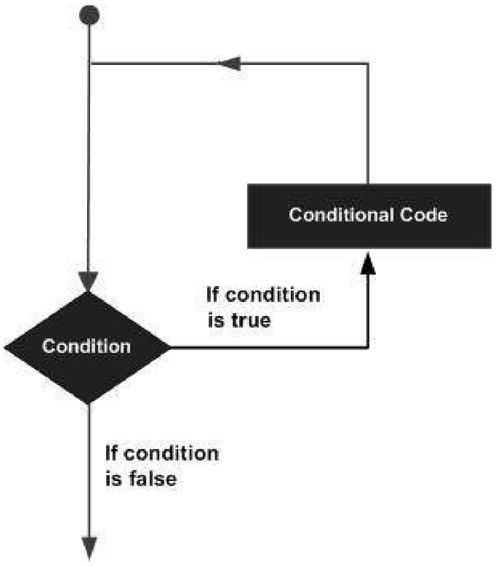


Figure 1.5: *Loops*

Python programming language provides the following types of loops to handle looping requirements:

While Loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

```
>> n = 0
>> while n < 10:
...   n += 1
...   print(n)
...print("Done.")
>>
```

For Loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

```
>> list = [1, 2, 3, 4]
#this builds an iterator object
>> it = iter(list)
>> for x in it:
...   print(x, end=" ")
>>
```

Loop Control Statements

The Loop control statements change the execution from its normal sequence. When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.

```
>> break     # terminates the loop
>> continue  # causes the loop to skip
               the remainder of its body
>> pass      # you don’t want any command
               or code to execute
```

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

User Defined Functions Example

```
#!/usr/bin/python3

# Print string
def printme(str):
   print("Print me: " + str)
   return

# Add a to b
def sum(a, b):
   # Add both the parameters and return them.
   total = a + b
   print ("Sum function: ", total)
   return total

result = sum(10, 20)
printme( str(result) )
```

2

Python for Network Engineers

Once we start writing code with sockets you will notice that information is sent in bytes. However, sometimes we need to print exchanged information to the user in a friendly format or we need to capture information given by the user and send the respective bytes over the network. We also might need to manipulate bits around in order to extract specific information from the socket data.

This section covers some useful python functions that are going to be used in subsequent chapters.

2.1 Encoding & Decoding

Today's programs need to be able to handle a wide variety of characters. Applications are often internationalized to display messages and output in a variety of user-selectable languages; the same program might need to output an error message in English, French, Japanese, Hebrew, or Russian. Web content can be written in any of these languages and can also include a variety of emoji symbols. Python's string type uses the Unicode Standard for representing characters, which lets Python programs work with all these different possible characters.

Unicode is a specification that aims to list every character used by human languages and give each character its own unique code. The Unicode specifications are continually revised and updated to add new languages and symbols.

To represent a unicode string as a string of bytes is known as encoding. To convert a string of bytes to a unicode string is known as decoding. You typically encode a unicode string whenever you need to use it for IO, for instance transfer it over the network, or save it to a disk file. You typically decode a string of bytes whenever you receive string data from the network or from a disk file.

UTF-8 is one of the most commonly used encodings, and Python often defaults to using it. UTF stands for “Unicode Transformation Format”, and the ‘8’ means that 8-bit values are used in the encoding. (There are also UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) [Reference]

Now that we reviewed the rudiments of Unicode, we can look at Python’s Unicode features.

String

Since Python 3.0, the language’s str type contains Unicode characters, meaning any string created using "unicode rocks!", ’unicode rocks!’, or the triple-quoted string syntax is stored as Unicode.

```
# String message
msg_1 = "Hello World!"
msg_2 = 'Hello World!'
```

Bytes to String

We can create a string using the decode() method of bytes. This method takes an encoding argument, such as UTF-8, and optionally an errors argument.

```
# Receive data from socket
data = conn.recv(1024)

# Convert bytes to string
msg = data.decode("utf-8")
```

String to Bytes

The opposite method of bytes.decode() is str.encode(), which returns a bytes representation of the Unicode string, encoded in the requested encoding.

```
# String message
msg = "Hello World!"

# Convert string to bytes
data = msg.encode("utf-8")

# Send data to socket
conn.send(data)
```

2.2 Shifting Bytes

Python supports a range of types to store sequences. There are six sequence types: strings, byte sequences (bytes objects), byte arrays (bytearray objects), lists, tuples, and range objects.

Notice that you can't manipulate bytes on integers by itself. They need to be part of a list or tuple or range of objects.

Let's go over an example and see how can we shift bits with integer values.

We create a list (L) with one integer element (15) and we print the respective types.

```
>> L = [15]
>> print(type(L))
<class 'list'>
>> print(type(L[0]))
<class 'int'>
```

Using the function **bytes()** to the list, it returns a new "bytes" object, which is an immutable sequence of small integers in the range 0 <= x < 256, print as ASCII characters when displayed. The number 15 is not a visible character so it prints the respective in hexadecimal number - 0x0f.

```
>> print(bytes(L))
b'\x0f'
>> print(bytes(L[0]))
b'\x00\x00\x00\x00\x00\x00\x00\x00
   \x00\x00\x00\x00\x00\x00\x00'
```

However, when **bytes()** is applied to the integer itself, it creates a byte object with that size - in this case it creates 15 bytes.

To shift bits to the left or to the right, apply '«' or '»' respectively to the list element itself.

```
>> L[0] = L[0] >> 2
>> print(bytes(L))
b'\x03'
>> L[0] = L[0] << 1
>> print(bytes(L))
b'\x06'
```

You can apply any bitwise operator described in Chapter 1.

```
>> L[0] |= 1
>> print(bytes(L))
b'\x07'
```

2.3 Struct Package

This module performs conversions between Python values and C structs represented as Python bytes objects. This can be used in handling binary data stored in files or from network connections, among other sources. It uses Format Strings as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

struct.unpack()

```
struct.unpack(format, buffer)
```

Unpack from the buffer buffer (presumably packed by pack(format, ...)) according to the format string format. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by calcsize().

struct.pack()

```
struct.pack(format, v1, v2, ...)
```

Return a bytes object containing the values v1, v2, ... packed according to the format string format. The arguments must match the values required by the format exactly.

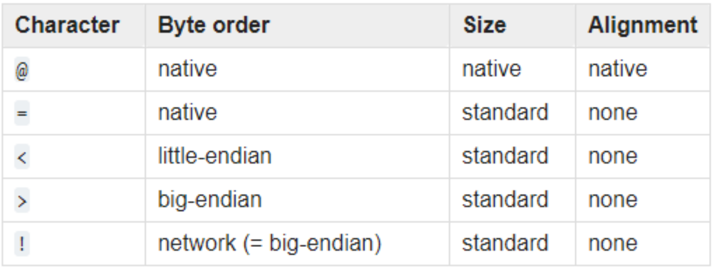
Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from Format Characters, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the Byte Order, Size, and Alignment.

Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to Figure 2.1.



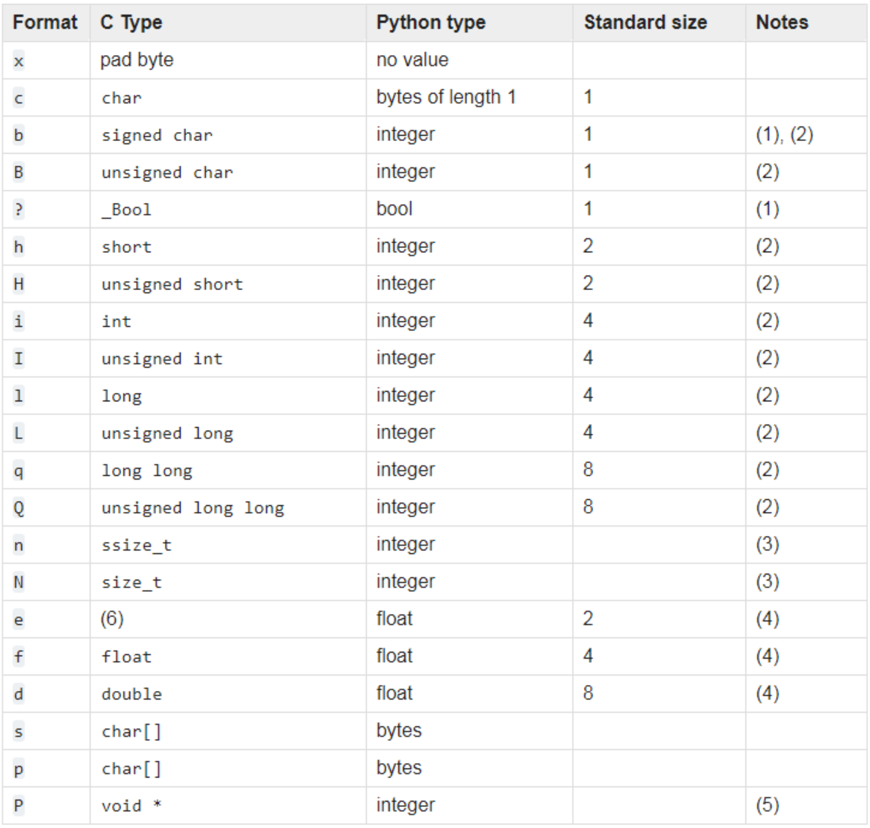
Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

Figure 2.1: *Struct - Byte Order, Size, and Alignment*

If the first character is not one of these, '@' is assumed.

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The "Standard size" column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of ’<’, ’>’, ’!’ or ’=’. When using native size, the size of the packed value is platform-dependent.



Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	bytes of length 1	1	
b	signed char	integer	1	(1), (2)
B	unsigned char	integer	1	(2)
?	_Bool	bool	1	(1)
h	short	integer	2	(2)
H	unsigned short	integer	2	(2)
i	int	integer	4	(2)
I	unsigned int	integer	4	(2)
l	long	integer	4	(2)
L	unsigned long	integer	4	(2)
q	long long	integer	8	(2)
Q	unsigned long long	integer	8	(2)
n	ssize_t	integer		(3)
N	size_t	integer		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	bytes		
p	char[]	bytes		
P	void *	integer		(5)

Figure 2.2: *Struct - Format Characters*

A format character may be preceded by an integral repeat count. For example, the format string '4h' means exactly the same as 'hhhh'.

Examples

Let's go over a simple example for the pack and unpack functions.

```
>> import struct
>> struct.pack('h h l', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
```

From Figure 2.2, h (integer) has 2 bytes and l (integer) has 4 bytes. Notice that in terms of C Types, h is a short and l is a long integer. With that, number 1 will pack with 2 bytes (0x00 0x01), number 2 will pack with 2 bytes (0x00 0x02), and number 3 will pack with 4 bytes (0x00 0x00 0x00 0x03).

```
>> import struct
>> struct.unpack('h h l', b'\x00\x01\x00\x02
                               \x00\x00\x00\x03')
(1, 2, 3)
```

When you unpack, you need to make sure that the format argument matches the data in the buffer argument.

Let's cover another example but this time let's use these functions with a raw network packet.

In Chapter 6 we cover with more detail how to parse data from raw packets. For now we will just give one example to see these functions in action.

An Ethernet frame (Figure 2.3 consists of a destination MAC address (6 bytes), source MAC address (6 bytes), ethernet type (2 bytes), payload data, and a CRC checksum (last 4 bytes).

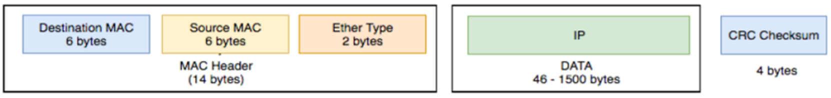


Figure 2.3: *Ethernet Frame Structure*

To extract the first 14 bytes from an ethernet frame received from a socket connection, we would do the following unpack command:

```
(...)
raw_data = conn.recv(65535)

dest_mac, src_mac, ether_type =
     struct.unpack('! 6s 6s H', raw_data[:14])
```

We use '!' to indicate that we are dealing with network byte order, then group the first 6 chars (6s) for destination MAC, the second 6 chars (6s) for source MAC and at last 2 bytes (H) for Ethernet Type.

2.4 Other Functions

map()

```
map(function, iterable, ...)
```

Return an iterator that applies function to every item of iterable, yielding the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see itertools.starmap().

Python program to demonstrate working of map:

```
# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

join()

```
string_name.join(iterable)
```

The **join()** method is a string method and returns a string in which the elements of sequence have been joined by str separator.

Python program to demonstrate the use of join function to join list elements with a character

```
>> list1 = ['1','2','3','4']
>> s = "-"
>> # joins elements of list1 by '-'
>> # and stores in sting s
>> s = s.join(list1)
>>
>> # join use to join a list of
>> # strings to a separator s
>> print(s)
1-2-3-4
```

This function is going to be used later to format IP and MAC addresses to a friendly format.

3
Python Socket Vocabulary

This section provides a brief description to the BSD socket interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

This section covers all the functions used in the examples presented in this document. For additional details or information go to the python socket API library.

3.1 Creating Sockets

The following function creates a socket object using the given address family, socket type and protocol number. Returns socket object handle.

socket.socket()

```
s = socket.socket(family, type, proto)
```

family

- socket.AF_INET (the default)
- socket.AF_INET6
- socket.AF_UNIX
- socket.AF_CAN
- socket.AF_PACKET
- socket.AF_RDS

type

- socket.SOCK_STREAM (the default)
- socket.SOCK_DGRAM
- socket.SOCK_RAW

proto

- socket.IPPROTO_IP = 0

- socket.IPPROTO_ICMP = 1
- ntohs(3) (Gateway-to-Gateway Protocol)
- socket.IPPROTO_TCP = 6
- socket.IPPROTO_UDP = 17
- socket.IPPROTO_RAW = 255

3.2 Sending Data

After creating a socket with **socket.socket()** and using the socket object handle (s), the following functions can be used to read data from the socket.

socket.send()

```
nbytes_sent = s.send(bytes)
```

Send data to the socket. The socket must be connected to a remote socket (e.g. TCP connection). Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

socket.sendall()

```
answer = s.sendall(bytes)
```

Send data to the socket. The socket must be connected to a remote socket. Unlike send(), this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

socket.sendto()

```
nbytes_sent = s.sendto(bytes, address)
```

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address (e.g. UDP connection). Return the number of bytes sent.

3.3 Receiving Data

After creating a socket with **socket.socket()** and using the socket object handle (s), the following functions can be used to write data from the socket.

socket.recv()

```
bytes_data = socket.recv(bufsize)
```

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize.

For best match with hardware and network realities, the value of bufsize should be a relatively small power of 2, for example, 4096.

socket.recvfrom()

```
bytes_data, address = socket.recvfrom(bufsize)
```

Receive data from the socket. The return value is a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data.

3.4 Socket Objects

After creating a socket with **socket.socket()** and using the socket object handle (s), the following functions can be used.

socket.accept()

```
conn, address = s.accept()
```

Accept a connection. The socket must be bound to an address and listening for connections (e.g. TCP connection). The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

socket.bind()

```
s.bind(address)
```

Bind the socket to address. The socket must not already be bound.

socket.listen()

```
s.listen([backlog])
```

Enable a server to accept connections. If backlog is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

socket.close()

```
s.close()
```

Mark the socket closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly, or to use a with statement around them.

socket.connect()

```
s.connect(address)
```

Connect to a remote socket at address. If the connection is interrupted by a signal, the method waits until the connection completes, or raise a socket.timeout on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an InterruptedError exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

socket.setsockopt()

```
s.setsockopt(level, optname, value: int)
```

Set the value of the given socket option.

level

- SOL_IP = 0
- SOL_TCP = 6
- SOL_UDP = 17

- socket.SOL_SOCKET = 65535

optname

- SOMAXCONN = 2147483647
- SO_ACCEPTCONN = 2
- SO_BROADCAST = 32
- SO_DEBUG = 1
- SO_DONTROUTE = 16
- SO_ERROR = 4103
- SO_EXCLUSIVEADDRUSE = -5
- SO_KEEPALIVE = 8
- SO_LINGER = 128
- SO_OOBINLINE = 256
- SO_RCVBUF = 4098
- SO_RCVLOWAT = 4100
- SO_RCVTIMEO = 4102
- SO_REUSEADDR = 4
- SO_SNDBUF = 4097
- SO_SNDLOWAT = 4099
- SO_SNDTIMEO = 4101
- SO_TYPE = 4104
- SO_USELOOPBACK = 64

3.5 Socket Services

After creating a socket with **socket.socket()** and using the socket object handle (s), the socket module also offers various network-related services.

socket.gethostbyname()

```
hostname = socket.gethostbyname(hostname)
```

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'. If the host name is an IPv4 address itself it is returned unchanged.

socket.ntohs()

```
socket.ntohs(x)
```

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.ntohl()

```
socket.ntohl(x)
```

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.htons()

```
socket.htons(x)
```

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.htonl()

```
socket.htonl(x)
```

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

4
Programming with Sockets

It is always fun to create your own custom clients and servers for any protocol of your choice. Python provides a good coverage on the low-level networking interface. It all starts with BSD socket interface. Python has a socket module that gives you the necessary functionality to work with the socket Interface.

Network programming in any programming language can begin with sockets. But what is a socket? Simply put, a network socket is a virtual end point where entities can perform inter-process communication. For example, one process sitting in a computer, exchanges data with another process sitting on the same or another computer. We typically label the first process which initiates the communication as the client and the latter one as the server.

Python has quite an easy way to start with the socket interface. In order to understand this better, let's use an echo client/server application as an example. Figure 4.1 shows a flow of client/server interaction diagram. In the interaction between this client and server, the server is listening for clients to transmit information. Once a client sends a message to the

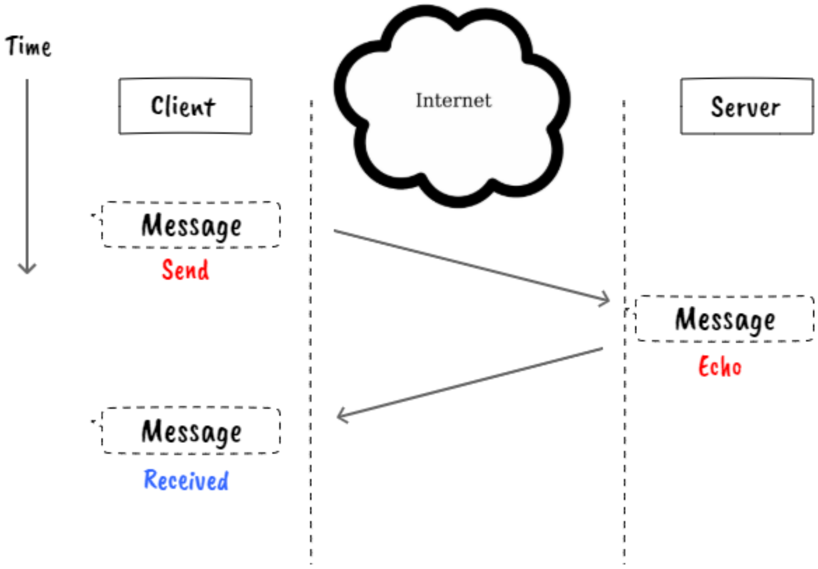


Figure 4.1: *Echo Diagram*

sever, the server echos that same message back to the client.

There are two possible ways to program this interaction between client and server; you can use UDP or TCP sockets.

Material Review

TCP vs UDP Comparison [Video]

4.1 Working with UDP sockets

Let's start by implementing the echo application using a UDP socket. Figure 4.2 represents the interaction between the client and the server using socket vocabulary.

Block Diagram

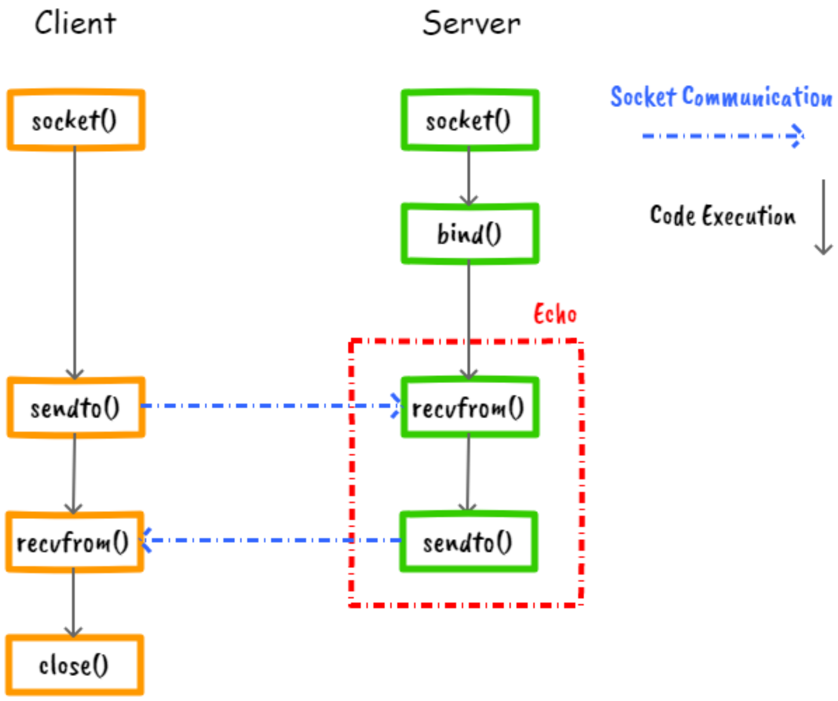


Figure 4.2: *UDP Echo Diagram*

Notice that Figure 4.2 presents more information than Figure 4.1. This is something common that happens when you are developing any type of code. You start with a simple diagram of your problem or application and as you develop your idea, you keep improving the diagram. Sometimes you might have different diagrams for that same problem or application representing different interactions between the client and the server.

The functions that are going to be used for the echo application using UDP sockets are summarized below. You can always go to chapter 3 to get more details about these functions.

Socket Functions

socket.socket() $\rightarrow$ Create socket

setsockopt() $\rightarrow$ Enable reuse port address

bind() $\rightarrow$ Indicates that this is the server

sendto() $\rightarrow$ Send message

recvfrom() $\rightarrow$ Receive message

close() $\rightarrow$ Close socket

UDP Client

```python
import socket

host = "localhost"
port = 10001

# Create Socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Send Message to Server
msg = input()
s.sendto(msg.encode('utf-8'), (host, port))

# Receive Echo from Server
data, addr = s.recvfrom(1024)
print("IP: " + addr[0] + " | " + "Port: " + str(addr[1]))
print(str(data.decode('utf-8')))

# Close Socket
s.close()
```

Listing 4.1: *UDP Client Echo*

UDP Server

```
import socket

host = ''
port = 10001

# Create Socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind Socket
s.bind((host, port))

# Receive Message from Client
data, addr = s.recvfrom(1024)

# Send Echo to Client
s.sendto(data, addr)

# Close Socket
s.close()
```

Listing 4.2: *UDP Server Echo*

4.2 Working with TCP sockets

Now we cover the echo application done with TCP sockets. Figure 4.3 represents the interaction between the client and the server echo application using socket vocabulary.

Block Diagram

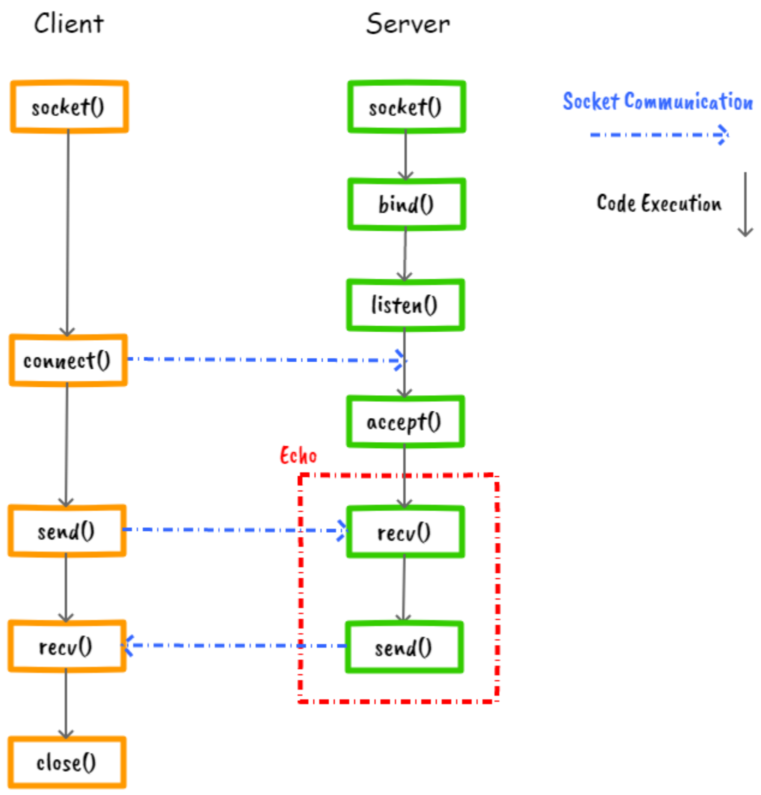


Figure 4.3: *TCP Echo Diagram*

For more details about the functions used for the TCP client/server echo application, review chapter 3

Socket Functions

socket.socket()	Create socket
setsockopt()	Enable reuse port address
bind()	Indicates that this is the server
listen()	Listen for incoming connections
accept()	Accept connections
send()	Send message
recv()	Receive message
close()	Close socket

TCP Client

```python
import socket

host = "localhost"
port = 12500

# Create Socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Connect socket
s.connect((host, port))

# Send message to server
msg = input()
s.send(msg.encode('utf-8'))

# Receive message from server
echoMessage = s.recv(1024)
print("Reply from Server: " + str(echoMessage.decode('utf-8
    ')))

# Close socket
s.close()
```

Listing 4.3: *TCP Client Echo*

TCP Server

```
import socket

host = ''
port = 12500


# Create Socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind socket
s.bind((host, port))

# Listen for incoming connections
s.listen(1)

# Connection Accepted
connectionSocket, addr = s.accept()

# Receive message from client
message = connectionSocket.recv(1024)

# Echo message to client
connectionSocket.send(message)

connectionSocket.close()
s.close()
```

Listing 4.4: *TCP Server Echo*

5

Concurrency

Let's start by modifying the TCP Server Echo code from the previous chapter and add the following lines of code:

1. Import the following module

```
from time import sleep
```

2. Add a delay between recv() and send()

```
(...)
message = connectionSocket.recv(1024)
sleep(3) # Wait 3 seconds
connectionSocket.send(message)
(...)
```

Now run the TCP Server Echo and try connecting more than one client at once. You will notice that while the server is attending the first client that connects to it, the remaining clients do not work. What is happening here?

The problem is that the server can only exchange messages with one client at a time. As soon as the first client connects, the server waits for a message from the client, waits for 3 seconds and then sends the echo message back to the client. During this time the server isn't able to attend new clients.

We can temporally fix this issue by increasing the number of connections that the socket can accept by increasing the number passed to **s.listen()**.

```
(...)
s.listen(5) # Accepts up to 5 clients
(...)
```

We still face a couple of challenges with this approach. Sure the client now get the connection establish but it still needs to wait for its turn to exchange messages with the server. And we also have the challenge of choosing the right amount of connections that are going to be accepted.

There are two possible solutions to tackle this problem. We can either use more than one thread or process, or use non-blocking sockets along with an event-driven architecture. We're going to look at both of these approaches, starting with multithreading.

5.1 Multithreading Server

Python has APIs that allow us to write both multithreading and multiprocessing applications. The principle behind multithreading and multiprocessing is simply to take copies of our code and run them in additional threads or processes. The operating system automatically schedules the threads and processes across available CPU cores to provide fair processing time allocation to all the threads and processes. This effectively allows a program to simultaneously run multiple operations. In addition, when a thread or process blocks, for example, when waiting for IO, the thread or process can be de-prioritized by the OS, and the CPU cores can be allocated to other threads or processes that have actual computation to do.

Here is an overview of how threads and processes relate to each other:

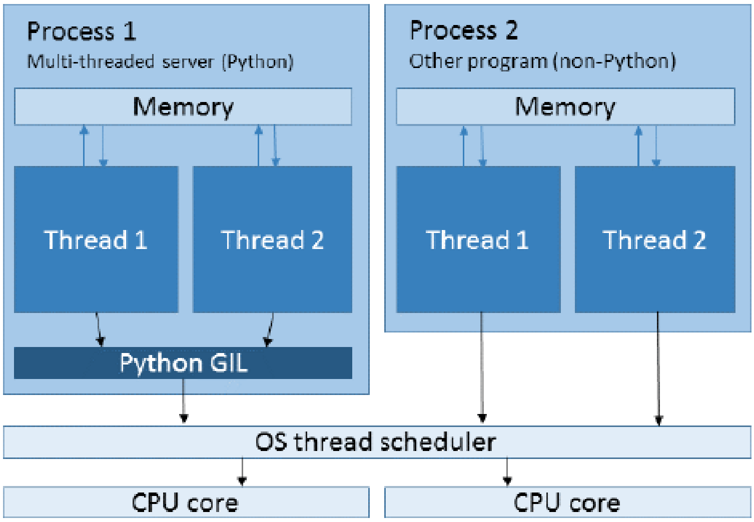


Figure 5.1: *Processes and Threads Comparison*

Threads exist within processes. A process can contain multiple threads but it always contains at least one thread, sometimes called the main thread. Threads within the same process share memory, so data transfer between threads is just a case of referencing the shared objects. Processes do not share memory, so other interfaces, such as files, sockets, or specially allocated areas of shared memory, must be used for transferring data between processes.

When threads have operations to execute, they ask the operating system thread scheduler to allocate them some time on a CPU, and the scheduler allocates the waiting threads to CPU cores based on various parameters, which vary from OS to OS. Threads in the same process may run on separate CPU cores at the same time.

Although two processes have been displayed in Figure 5.1, multiprocessing is not going on here, since the processes belong to different applications. The second process is displayed to illustrate a key difference between Python threading and threading in most other programs. This difference is the presence of the GIL.

Threading and the GIL

The CPython interpreter (the standard version of Python available for download from www.python.org) contains something called the Global Interpreter Lock (GIL). The GIL exists to ensure that only a single thread in a Python process can run at a time, even if multiple CPU cores are present. The reason for having the GIL is that it makes the underlying C code of the Python interpreter much easier to write and maintain. The drawback of this is that Python programs using multithreading cannot take advantage of multiple cores for parallel computation.

This is a cause of much contention; however, for us this is not so much of a problem. Even with the GIL present, threads that are blocking on I/O are still de-prioritized by the OS and put into the background, so threads that do have computational work to do can run instead. The following figure is a simplified illustration of this:

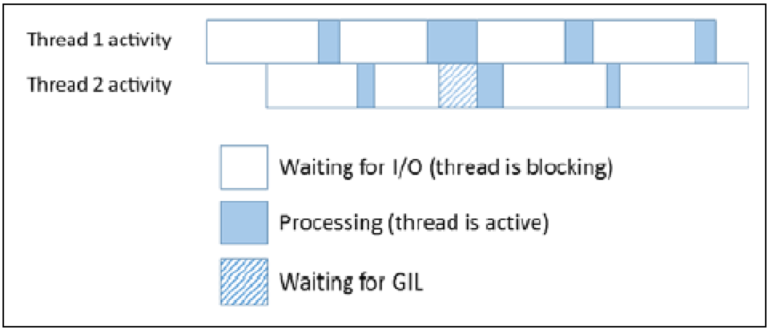

Thread 1 activity
Thread 2 activity
Waiting for I/O (thread is blocking)
Processing (thread is active)
Waiting for GIL

Figure 5.2: *Global Interpreter Lock (GIL) Example*

The Waiting for GIL state is where a thread has sent or received some data and so is ready to come out of the blocking state, but another thread has the GIL, so the ready thread is forced to wait. In many network applications, including our echo and chat servers, the time spent waiting on I/O is much higher than the time spent processing data. As long as we don't have a very large number of connections (a situation we'll discuss later on when we come to event driven architectures), thread contention caused by the GIL is relatively low, and hence threading is still a suitable architecture for these network server applications.

With this in mind, we're going to use multithreading rather than multiprocessing in our echo server. The shared data model will simplify the code that we'll need for allowing our chat clients to exchange messages with each other, and because we're I/O bound, we don't need processes for parallel computation. Another reason for not using processes in this case is that processes are more "heavyweight" in terms of the OS resources, so creating a new process takes longer than creating a new thread. Processes also use more memory.

One thing to note is that if you need to perform an intensive computation in your network server application (maybe you need to compress a large file before sending it over the network), then you should investigate methods for running this in a separate process. Because of quirks in the implementation of the GIL, having even a single computationally intensive thread in a mainly I/O bound process when multiple CPU cores are available can severely impact the performance of all the I/O bound threads.

Multithreaded TCP Echo Server

A benefit of the multithreading approach is that the OS handles the thread switches for us, which means we can continue to write our program in a procedural style. Hence we only need to make small adjustments to our server program to make it multithreaded, and thus, capable of handling multiple clients simultaneously.

Let's add this threading functionality to the TCP Server Echo code from previous chapter. I will describe the code that we are going to add step by step and you can find the all version in ...

1. Import the thread module

```
import threading
```

2. Put **s.accept()**, **connectionSocket.recv()**, **connectionSocket.send()**, and **connectionSocket.close()** under an infinite while loop.

```
(...)
s.listen(1)

while True:
    # Connection Accepted
    connectionSocket, addr = s.accept()

    # Receive message from client
    message = connectionSocket.recv(1024)

    # Echo message to client
    connectionSocket.send(message)

    connectionSocket.close()

s.close()
```

Notice so far that nothing changed in the way that the TCP Server works besides that instead of closing the connection, now it runs forever and after attending one client cycles back to attend a client that connects again to the server.

3. Let's create a function **client_thread()**, copy **connectionSocket.recv()**, **connectionSocket.send()**, and **connectionSocket.close()** inside the function and add connectionSocket and addr as arguments of that function.

```
def client_thread(connectionSocket, addr):
    # Receive message from client
    message = connectionSocket.recv(1024)

    # Echo message to client
    connectionSocket.send(message)

    connectionSocket.close()
```

4. Call that function after **s.accetpt()** and pass connectionSocket and addr to **client_thread()** function.

```
(...)
s.listen(1)

while True:
    # Connection Accepted
    connectionSocket, addr = s.accept()

    client_thread(connectionSocket, addr)

s.close()
```

We are still not dealing with threads yet but we have the code ready to incorporate it with threads.

5. Delete **client_thread()** from the inifite loop. Create a thread and assign **client_thread()** function to target, and connectionSocket and addr as arguments of the thread. Set thread deamon to true and start the thread.

```
(...)
s.listen(1)

while True:
    # Connection Accepted
    connectionSocket, addr = s.accept()

    thread = threading.Thread(
             target=client_thread,
             args=(connectionSocket, addr))

    thread.setDaemon(True)
    thread.start()

s.close()
```

Note: Setting the daemon argument in the thread constructor to True, will allow the program to exit if we hit ctrl - c without us having to explicitly close all of our threads first.

If you try this echo server with multiple clients, then you'll see that the server can now handle multiple connections.

TCP Threading Server

```
from time import sleep
import socket
import threading

host = ''
port = 12500


def client_thread(connectionSocket, addr):
  # Receive message from client
  message = connectionSocket.recv(1024)

  sleep(3)

  # Echo message to client
  connectionSocket.send(message)

  connectionSocket.close()

# Create Socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind socket
s.bind((host, port))

# Listen for incoming connections
s.listen(1)

while True:
  # Connection Accepted
  connectionSocket, addr = s.accept()

  thread = threading.Thread(target=client_thread, args=(
      connectionSocket, addr))

  thread.setDaemon(True)
  thread.start()

s.close()
```

Listing 5.1: *TCP Server Echo with Threads*

5.2 Event Driven Server

For many purposes threads are great, especially because we can still program in the familiar procedural, blocking-IO style. But they suffer from the drawback that they struggle when managing large numbers of connections simultaneously, because they are required to maintain a thread for each connection. Each thread consumes memory, and switching between threads incurs a type of CPU overhead called context switching. Although these aren't a problem for small numbers of threads, they can impact performance when there are many threads to manage. Multiprocessing suffers from similar problems.

An alternative to threading and multiprocessing is using the event-driven model. In this model, instead of having the OS automatically switch between active threads or processes for us, we use a single thread which registers blocking objects, such as sockets, with the OS. When these objects become ready to leave the blocking state, for example a socket receives some data, the OS notifies our program; our program can then access these objects in non-blocking mode, since it knows that they are in a state that is ready for immediate use. Calls made to objects in non-blocking mode always return immediately. We structure our application around a loop, where we wait for the OS to notify us of activity on our blocking objects, then we handle that activity, and then we go back to waiting. This loop is called the event loop.

This approach provides comparable performance to threading and multiprocessing, but without the memory or context switching overheads, and hence allows for greater scaling on the same hardware. The challenge of engineering applications that can efficiently handle very large numbers of simultaneous connections has historically been called the c10k problem, referring to the handling of ten-thousand concurrent connections in a single thread. With the help of event-driven architectures, this problem was solved, though the term is still often used to refer to the challenges of scaling when it comes to handling many concurrent connections.

The following diagram shows the relationship of processes and threads in an event-driven server:

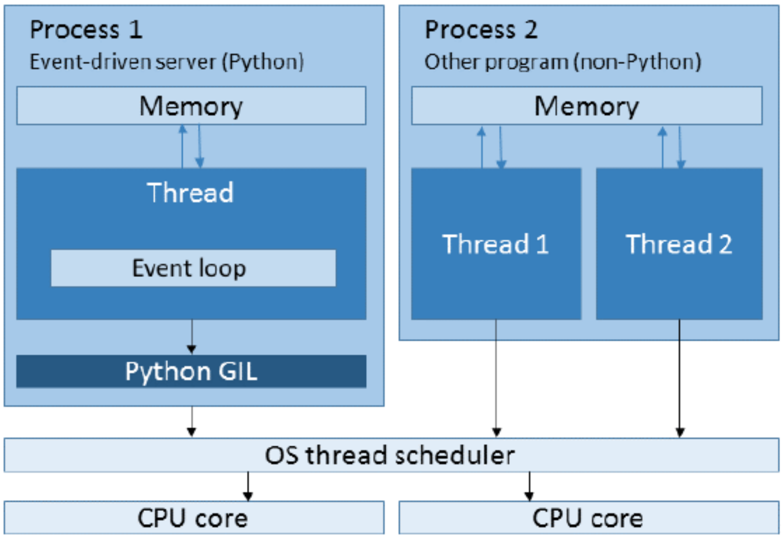


Figure 5.3: *Event Driven Example*

Although the GIL and the OS thread scheduler are shown here for completeness, in the case of an event-driven server, they have no impact on performance because the server only uses a single thread. The scheduling of I/O handling is done by the application.

Event-Driven TCP Echo Server

As mentioned earlier, if the number of connections start to be very large, then we need to start navigating towards a solution that uses multiprocessing instead of multithreading.

Python provides two classes for multiprocessing i.e. Process and Pool class. Though Pool and Process both execute the task parallelly, their way of executing tasks parallelly is different.

The pool distributes the tasks to the available processors using a FIFO scheduling. It works like a map-reduce architecture. It maps the input to the different processors and collects the output from all the processors. After the execution of code, it returns the output in form of a list or array. It waits for all the tasks to finish and then returns the output. The processes in execution are stored in memory and other non-executing processes are stored out of memory.

The process class puts all the processes in memory and schedules execution using FIFO policy. When the process is suspended, it pre-empts and schedules a new process for execution.

When to use Pool or Process? It depends on the task in hand. The pool allows you to do multiple jobs per process, which may make it easier to parallelize your program. If you have a million tasks to execute in parallel, you can create a Pool with a number of processes as many as CPU cores and then pass the list of the million tasks to pool.map. The pool will distribute those tasks to the worker processes(typically the same in number as available cores) and collects the return values in the form of a list and pass it to the parent process. Launching separate million processes would be much less practical (it would probably break your OS).

On the other hand, if you have a small number of tasks to execute in parallel, and you only need each task done once, it may be perfectly reasonable to use a separate multiprocessing.process for each task, rather than setting up a Pool.

Writing code using a lower lever pool API can be can be quite involved, and complicated to manage. There are several libraries and frameworks available for taking some of the leg work out of writing the code that benefits by using pools.

The **eventlet** library provides a high-level API for event-driven programming, but it does so in a style that mimics the procedural, blocking-IO style that we used in our multithreaded example. The upshot is that we can effectively take our multithreaded server code, make a few modifications to it to use eventlet instead, and immediately gain the benefits of the event-driven model.

1. Include eventlet module. Remover all other includes

```
import eventlet
```

2. Substitute **socket.socket()**, **s.setsockopt()**, **s.bind()**, and **s.listen()** with **eventlet.listen()**.

```
server = eventlet.listen(host, port)
```

eventlet.listen() opens a server socket and it sets SO_REUSEADDR on the socket.

3. Create a set of pools to your eventlet.

```
pool = eventlet.GreenPool(10000)
```

enventlet.GreenPool() controls the amount of connections that can be done with eventlet.

4. Remove the lines of code related with threading and add **pool.spawn_n()**.

```
(...)
while True:
    # Connection Accepted
    connectionSocket, addr = s.accept()

    pool.spawn_n(client_thread, connectionSocket)
```

pool.spawn_n() launches multiple greenthreads in parallel. It receives **client_thread()** as the function, and connectionSocket and addr as arguments of that function.

We are almost done. We just need to modify the sleep function under **client_thread()** to be compatible with an eventlet greenthread.

5. Substitute **sleep()** function with **eventlet.greenthread.sleep()**.

```python
def client_thread(connectionSocket, addr):
    # Receive message from client
    message = connectionSocket.recv(1024)

    eventlet.greenthread.sleep(3)

    # Echo message to client
    connectionSocket.send(message)

    connectionSocket.close()
```

We can test this with our multithreaded client to ensure that it works as expected. As you can see, it's pretty much identical to our multithreaded server, with a few changes made so as to use eventlet.

TCP Event Server

```python
import eventlet

host = ''
port = 12500


def client_thread(connectionSocket, addr):
  # Receive message from client
  message = connectionSocket.recv(1024)

  eventlet.greenthread.sleep(3)

  # Echo message to client
  connectionSocket.send(message)

  connectionSocket.close()

# Listen for incoming connections
s = eventlet.listen((host, port))

pool = eventlet.GreenPool(10000)

while True:
  # Connection Accepted
  connectionSocket, addr = s.accept()

  pool.spawn_n(client_thread, connectionSocket, addr)
```

Listing 5.2: *TCP Server Echo with eventlet*

6 Packet Analyzer

The basic tool for observing the messages exchanged between executing protocol entities is called a packet sniffer. As the name suggests, a packet sniffer captures ("sniffs") messages being sent/received from/by your computer; it will also typically store and/or display the contents of the various protocol fields in these captured messages. A packet sniffer itself is passive. It observes messages being sent and received by applications and protocols running on your computer, but never sends packets itself. Similarly, received packets are never explicitly addressed to the packet sniffer. Instead, a packet sniffer receives a copy of packets that are sent/received from/by application and protocols executing on your machine.

Figure 6.1 shows the structure of a packet sniffer. At the right of Figure 6.1 are the protocols (in this case, Internet protocols) and applications (such as a web browser or ftp client) that normally run on your computer. The packet sniffer, shown within the dashed rectangle in Figure 6.1 is an addition to the usual software in your computer, and consists of two parts. The packet capture library receives a copy of every link-layer frame that is sent from or received by your computer.

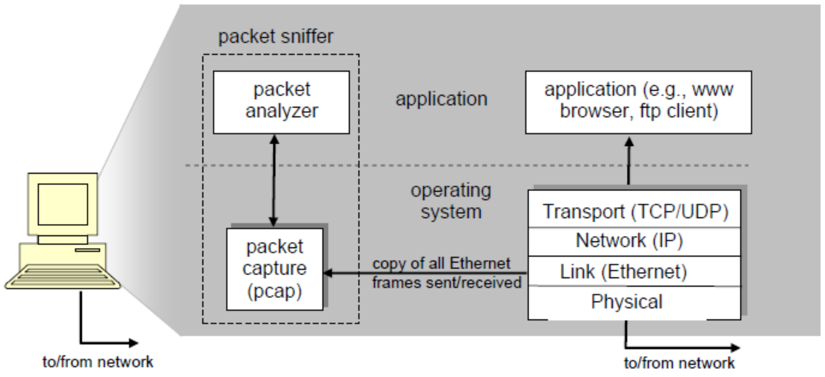


Figure 6.1: *Packet Sniffer Structure*

Remember that messages exchanged by higher layer protocols such as HTTP, FTP, TCP, UDP, DNS, or IP all are eventually encapsulated in link-layer frames that are transmitted over physical media such as an Ethernet cable (Figure 6.2).

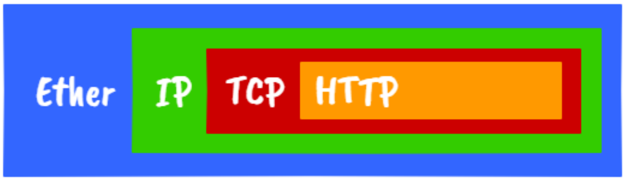


Figure 6.2: *Encapsulation Example*

Going back to Figure 6.1, the assumed physical media is an Ethernet, and so all upper-layer protocols are eventually encapsulated within an Ethernet frame. Capturing all link-layer frames thus gives you all messages sent/received from/by all protocols and applications executing in your computer.

The second component of a packet sniffer is the packet analyzer, which displays the contents of all fields within a protocol message. In order to do so, the packet analyzer must "understand" the structure of all messages exchanged by protocols. For example, suppose we are interested in displaying the various fields in messages exchanged by the HTTP protocol in Figure 6.1. The packet analyzer understands the format of Ethernet frames, and so can identify the IP datagram within an Ethernet frame. It also understands the IP datagram format, so that it can extract the TCP segment within the IP datagram. Finally, it understands the TCP segment structure, so it can extract the HTTP message contained in the TCP segment. Finally, it understands the HTTP protocol and so, for example, knows that the first bytes of an HTTP message will contain the string "GET" "POST" or "HEAD".

Wireshark is a free and open-source packet analyzer that is widely used for network troubleshooting, analysis, software and communications protocol development, and education. This chapter focus on the implementation of a mini version of Wireshark (turtle sniffer), using the terminal as a content display.

Before we go over turtle sniffer, let's cover a basic packet sniffer.

6.1 Basic Sniffer

A simple packet sniffer in Python can be created with the help socket module. We can use the raw socket type to get the packets. A raw socket provides access to the underlying protocols, which support socket abstractions.

As some behaviors of the socket module depend on the operating system socket API and there is no uniform API for using a raw socket under a different operating system, we are going to show two examples: one that uses a Linux OS and another one that uses Windows.

The full code for the basic sniffer can be found under Appendix A.

Basic Sniffer under Linux OS

Here are the steps to create a basic packet sniffer with socket module.

1. Create a new file called ***basic_sniffer_linux.py*** and open it in your editor.
2. Import the required modules:

```
import socket
```

3. Now we can create a PACKET raw socket:

```
s = socket.socket(socket.AF_PACKET,
                  socket.SOCK_RAW,
                  socket.ntohs(3))
```

Both reading and writing to a raw socket require creating a raw socket first. Here we use:

- AF_PACKET: Low-level packet interface
- SOCK_RAW: Raw socket
- ntohs(3): Packet protocol -> Gateway-to-Gateway (captures everything including Ethernet frames)

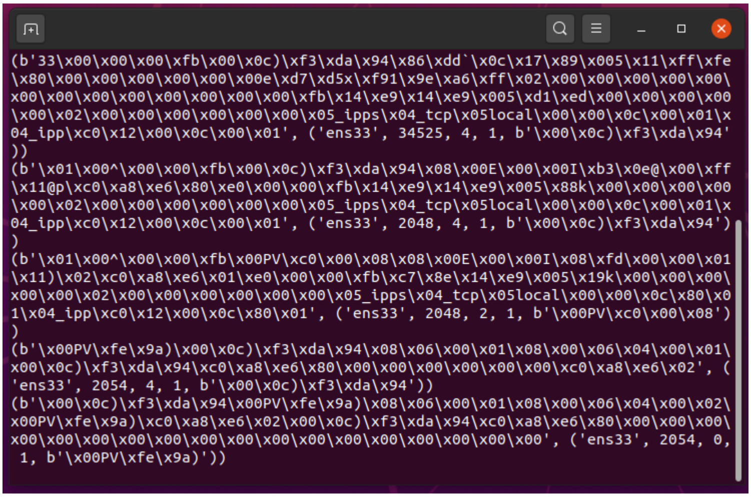
4. Next, start an infinite loop to receive data from the socket:

```
while True:
    print( s.recvfrom(65565) )
```

The ***recvfrom*** method in the socket module helps us to receive all the data from the socket. The parameter passed is the buffer size; 65565 is the maximum buffer size.
5. Now run the program with Python:

```
sudo python3 basic_sniffer_linux.py
```

You should get similar results as shown in Figure 6.3



```
(b'33\x00\x00\x00\xfb\x00\x0c)\xf3\xda\x94\x86\xdd`\x0c\x17\x89\x005\x11\xff\xfe
\x80\x00\x00\x00\x00\x00\x00e\xd7\xd5x\xf91\x9e\xa6\xff\x02\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\xfb\x14\xe9\x14\xe9\x005\xd1\xed\x00\x00\x00\x00
\x00\x02\x00\x00\x00\x00\x00\x00\x05_ipps\x04_tcp\x05local\x00\x00\x0c\x00\x01\x
04_ipp\xc0\x12\x00\x0c\x00\x01', ('ens33', 34525, 4, 1, b'\x00\x0c)\xf3\xda\x94'
))
(b'\x01\x00^\x00\x00\xfb\x00\x0c)\xf3\xda\x94\x08\x00E\x00\x00I\xb3\x0e@\x00\xff
\x11@p\xc0\xa8\xe6\x80\xe0\x00\x00\xfb\x14\xe9\x14\xe9\x005\x88k\x00\x00\x00\x00
\x00\x02\x00\x00\x00\x00\x00\x00\x05_ipps\x04_tcp\x05local\x00\x00\x0c\x00\x01\x
04_ipp\xc0\x12\x00\x0c\x00\x01', ('ens33', 2048, 4, 1, b'\x00\x0c)\xf3\xda\x94')
)
(b'\x01\x00^\x00\x00\xfb\x00PV\xc0\x00\x08\x08\x00E\x00\x00I\x08\xfd\x00\x00\x01
\x11)\x02\xc0\xa8\xe6\x01\xe0\x00\x00\xfb\xc7\x8e\x14\xe9\x005\x19k\x00\x00\x00\
x00\x00\x02\x00\x00\x00\x00\x00\x00\x05_ipps\x04_tcp\x05local\x00\x00\x0c\x80\x0
1\x04_ipp\xc0\x12\x00\x0c\x80\x01', ('ens33', 2048, 2, 1, b'\x00PV\xc0\x00\x08')
)
(b'\x00PV\xfe\x9a)\x00\x0c)\xf3\xda\x94\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01\
x00\x0c)\xf3\xda\x94\xc0\xa8\xe6\x80\x00\x00\x00\x00\x00\x00\xc0\xa8\xe6\x02', (
'ens33', 2054, 4, 1, b'\x00\x0c)\xf3\xda\x94'))
(b'\x00\x0c)\xf3\xda\x94\x00PV\xfe\x9a)\x08\x06\x00\x01\x08\x00\x06\x04\x00\x02\
x00PV\xfe\x9a)\xc0\xa8\xe6\x02\x00\x0c)\xf3\xda\x94\xc0\xa8\xe6\x80\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00', ('ens33', 2054, 0,
 1, b'\x00PV\xfe\x9a)'))
```

Figure 6.3: *Basic Sniffer Capture (Linux OS)*

Basic Sniffer under Windows

When socket protocol is set to capture everything including Ethernet frames (***socket.ntohs(3)***), then all incoming packets will be passed to the packet socket before they are passed to the protocols implemented in the kernel.

For windows raw sockets, due to Win32 limitations the internet socket API doesn't not allow access to receive Ethernet frames. You can only use the API to generate and receive IP packets. We need to find a way to work around this limitation.

Fortunately there are libraries that can overcome this limitation. What you can do is to use a **Winpcap/Npcap** based library such as Scapy, to access Raw low-level sockets.

Scapy has lots of capabilities of handling packets, but for learning purposes we will use it to access raw data and manipulate the data as if we were using the normal socket API.

Let's go over the steps to create a basic packet sniffer under windows.

1. Create a new file called ***basic_sniffer_windows.py*** and open it in your editor.
2. Import the required modules:

```
from scapy.all import *
```

3. We need to select the correct network interface to collect data. The number that is associated to your Network Interface Card (NIC) can vary from system to system. We need to list the available interfaces and select the correct number for your system. For this example, we will let the user enter the desired interface number.

```
IFACES.show()
number = input("NIC #> ")
```

4. Then we get the interface name by index

```
iface = IFACES.dev_from_index( int(number) )
```

5. Create a level 2 socket

```
socket = conf.L2socket(iface=iface)
```

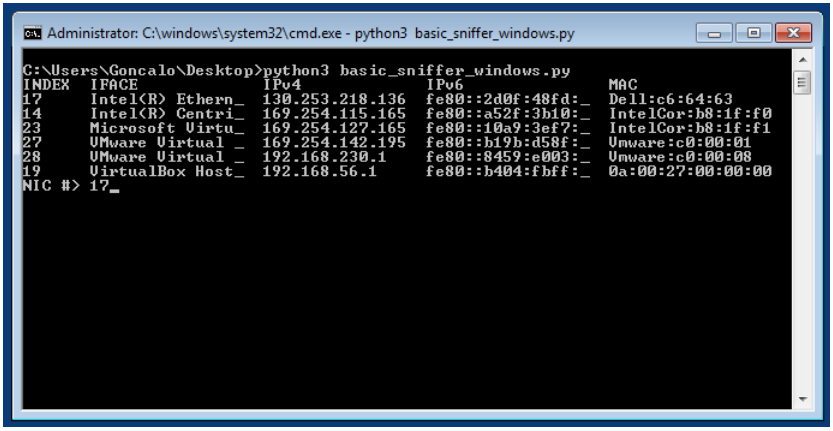
6. Next, start an infinite loop to receive data from the socket:

```
while True:
    packet_raw = socket.recv_raw()
    print(packet_raw[1])
```

7. Now run the program with Python:

```
python3 basic_sniffer_windows.py
```

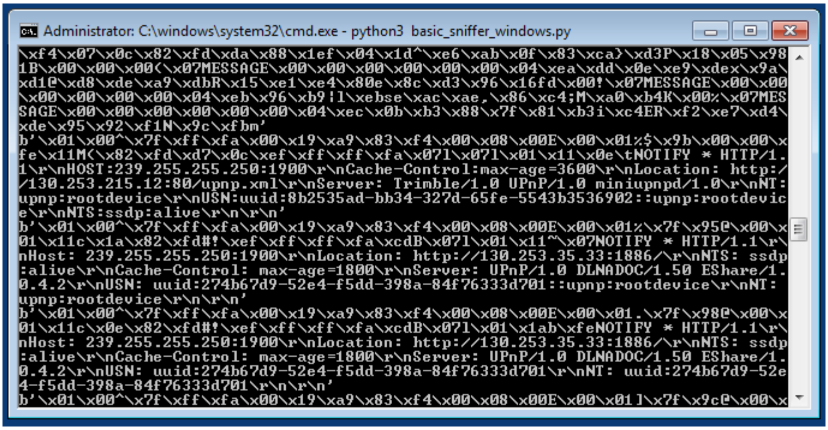
You should get similar results as shown in Figure 6.4 and Figure 6.5



Administrator: C:\windows\system32\cmd.exe - python3 basic_sniffer_windows.py

```
C:\Users\Goncalo\Desktop>python3 basic_sniffer_windows.py
INDEX  IFACE             IPv4              IPv6               MAC
17     Intel(R) Ethern_  130.253.218.136   fe80::2d0f:48fd:_  Dell:c6:64:63
14     Intel(R) Centri_  169.254.115.165   fe80::a52f:3b10:_  IntelCor:b8:1f:f0
23     Microsoft Virtu_  169.254.127.165   fe80::10a9:3ef7:_  IntelCor:b8:1f:f1
27     VMware Virtual _  169.254.142.195   fe80::b19b:d58f:_  Vmware:c0:00:01
28     VMware Virtual _  192.168.230.1     fe80::8459:e003:_  Vmware:c0:00:08
19     VirtualBox Host_  192.168.56.1      fe80::b404:fbff:_  0a:00:27:00:00:00
NIC #> 17_
```

Figure 6.4: *Select Interface Number (Windows)*



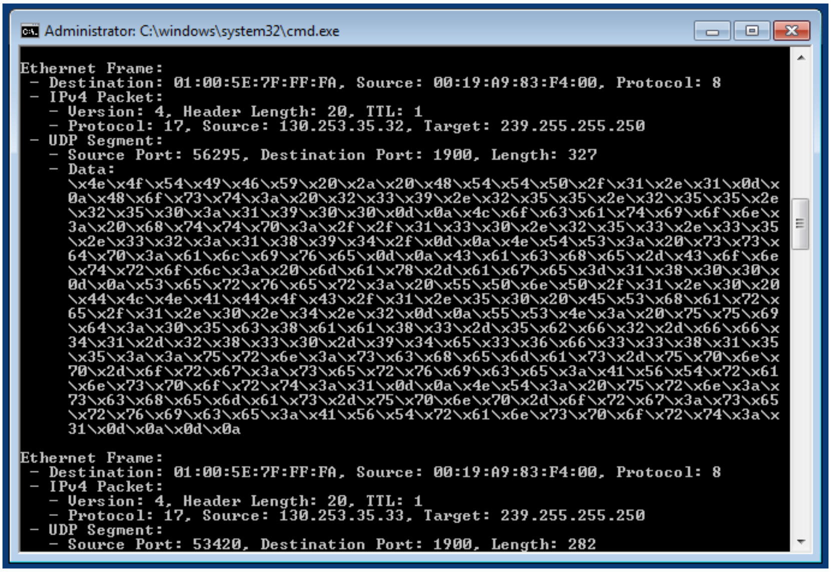
Administrator: C:\windows\system32\cmd.exe - python3 basic_sniffer_windows.py

```
\xf4\x07\x0c\x82\xfd\xda\x88\x1ef\x04\x1d^\xe6\xab\x0f\x83\xca}\xd3P\x18\x05\x98
1B\x00\x00\x00<\x07MESSAGE\x00\x00\x00\x00\x00\x00\x04\xea\xdd\x0e\xe9\xdex\x9a\
xd1@\xd8\xde\xa9\xdbR\x15\xe1\xe4\x80e\x8c\xd3\x96\x16fd\x00!\x07MESSAGE\x00\x00
\x00\x00\x00\x00\x04\xeb\x96\xb9¦l\xebse\xac\xae,\x86\xc4;M\xa0\xb4K\x00%\x07MES
SAGE\x00\x00\x00\x00\x00\x00\x04\xec\x0b\xb3\x88\x7f\x81\xb3i\xc4ER\xf2\xe7\xd4\
xde\x95\x92\xf1N\x9c\xfbm'
b'\x01\x00^\x7f\xff\xfa\x00\x19\xa9\x83\xf4\x00\x08\x00E\x00\x01%$\x9b\x00\x00\x
fe\x11M<\x82\xfd\xd7\x0c\xef\xff\xff\xfa\x07l\x07l\x01\x11\x0e\tNOTIFY * HTTP/1.
1\r\nHOST:239.255.255.250:1900\r\nCache-Control:max-age=3600\r\nLocation: http:/
/130.253.215.12:80/upnp.xml\r\nServer: Trimble/1.0 UPnP/1.0 miniupnpd/1.0\r\nNT:
upnp:rootdevice\r\nUSN:uuid:8b2535ad-bb34-327d-65fe-5543b3536902::upnp:rootdevic
e\r\nNTS:ssdp:alive\r\n\r\n'
b'\x01\x00^\x7f\xff\xfa\x00\x19\xa9\x83\xf4\x00\x08\x00E\x00\x01%\x7f\x95@\x00\x
01\x11c\x1a\x82\xfd#!\xef\xff\xff\xfa\xcdB\x07l\x01\x11^\x07NOTIFY * HTTP/1.1\r\
nHost: 239.255.255.250:1900\r\nLocation: http://130.253.35.33:1886/\r\nNTS: ssdp
:alive\r\nCache-Control: max-age=1800\r\nServer: UPnP/1.0 DLNADOC/1.50 EShare/1.
0.4.2\r\nUSN: uuid:274b67d9-52e4-f5dd-398a-84f76333d701::upnp:rootdevice\r\nNT:
upnp:rootdevice\r\n\r\n'
b'\x01\x00^\x7f\xff\xfa\x00\x19\xa9\x83\xf4\x00\x08\x00E\x00\x01.\x7f\x98@\x00\x
01\x11c\x0e\x82\xfd#!\xef\xff\xff\xfa\xcdB\x07l\x01\x1ab\xfeNOTIFY * HTTP/1.1\r\
nHost: 239.255.255.250:1900\r\nLocation: http://130.253.35.33:1886/\r\nNTS: ssdp
:alive\r\nCache-Control: max-age=1800\r\nServer: UPnP/1.0 DLNADOC/1.50 EShare/1.
0.4.2\r\nUSN: uuid:274b67d9-52e4-f5dd-398a-84f76333d701\r\nNT: uuid:274b67d9-52e
4-f5dd-398a-84f76333d701\r\n\r\n'
b'\x01\x00^\x7f\xff\xfa\x00\x19\xa9\x83\xf4\x00\x08\x00E\x00\x01]\x7f\x9c@\x00\x
```

Figure 6.5: *Basic Sniffer Capture (Windows)*

6.2 Turtle Sniffer

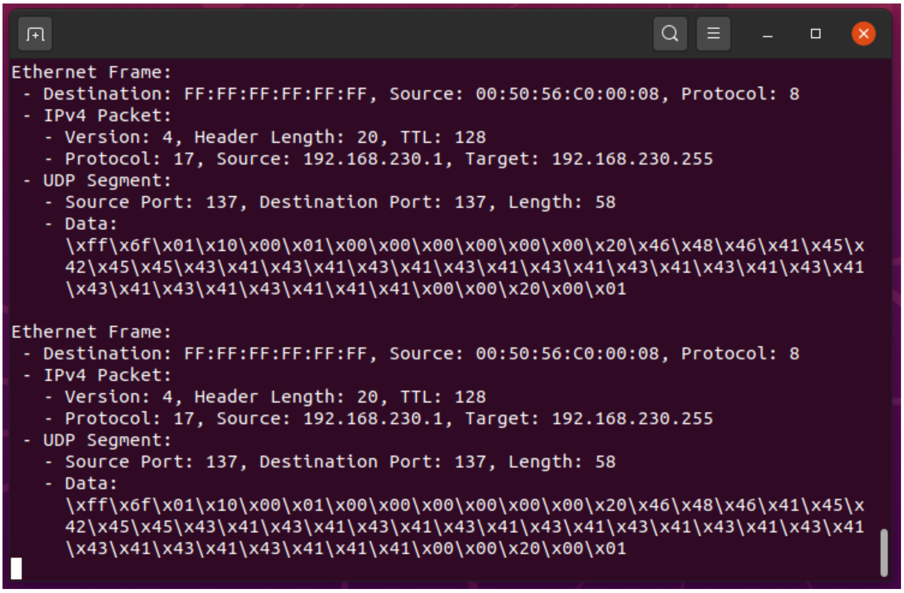
The basic packet sniffer captures all Ethernet frames and displays the raw data on the console or terminal. Turtle sniffer will do a bit more with that raw data. As you can see in Figure 6.6 and Figure 6.7 the data is parsed and displayed in a user friendly manner.



```
Ethernet Frame:
 - Destination: 01:00:5E:7F:FF:FA, Source: 00:19:A9:83:F4:00, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 1
   - Protocol: 17, Source: 130.253.35.32, Target: 239.255.255.250
 - UDP Segment:
   - Source Port: 56295, Destination Port: 1900, Length: 327
   - Data:
     \x4e\x4f\x54\x49\x46\x59\x20\x2a\x20\x48\x54\x54\x50\x2f\x31\x2e\x31\x0d\x
     0a\x48\x6f\x73\x74\x3a\x20\x32\x33\x39\x2e\x32\x35\x35\x2e\x32\x35\x35\x2e
     \x32\x35\x30\x3a\x31\x39\x30\x30\x0d\x0a\x4c\x6f\x63\x61\x74\x69\x6f\x6e\x
     3a\x20\x68\x74\x74\x70\x3a\x2f\x2f\x31\x33\x30\x2e\x32\x35\x33\x2e\x33\x35
     \x2e\x33\x32\x3a\x31\x38\x39\x34\x2f\x0d\x0a\x4e\x54\x53\x3a\x20\x73\x73\x
     64\x70\x3a\x61\x6c\x69\x76\x65\x0d\x0a\x43\x61\x63\x68\x65\x2d\x43\x6f\x6e
     \x74\x72\x6f\x6c\x3a\x20\x6d\x61\x78\x2d\x61\x67\x65\x3d\x31\x38\x30\x30\x
     0d\x0a\x53\x65\x72\x76\x65\x72\x3a\x20\x55\x50\x6e\x50\x2f\x31\x2e\x30\x20
     \x44\x4c\x4e\x41\x44\x4f\x43\x2f\x31\x2e\x35\x30\x20\x45\x53\x68\x61\x72\x
     65\x2f\x31\x2e\x30\x2e\x34\x2e\x32\x0d\x0a\x55\x53\x4e\x3a\x20\x75\x75\x69
     \x64\x3a\x30\x35\x63\x38\x61\x61\x38\x33\x2d\x35\x62\x66\x32\x2d\x66\x66\x
     34\x31\x2d\x32\x38\x33\x30\x2d\x39\x34\x65\x33\x36\x66\x33\x33\x38\x31\x35
     \x35\x3a\x3a\x75\x72\x6e\x3a\x73\x63\x68\x65\x6d\x61\x73\x2d\x75\x70\x6e\x
     70\x2d\x6f\x72\x67\x3a\x73\x65\x72\x76\x69\x63\x65\x3a\x41\x56\x54\x72\x61
     \x6e\x73\x70\x6f\x72\x74\x3a\x31\x0d\x0a\x4e\x54\x3a\x20\x75\x72\x6e\x3a\x
     73\x63\x68\x65\x6d\x61\x73\x2d\x75\x70\x6e\x70\x2d\x6f\x72\x67\x3a\x73\x65
     \x72\x76\x69\x63\x65\x3a\x41\x56\x54\x72\x61\x6e\x73\x70\x6f\x72\x74\x3a\x
     31\x0d\x0a\x0d\x0a

Ethernet Frame:
 - Destination: 01:00:5E:7F:FF:FA, Source: 00:19:A9:83:F4:00, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 1
   - Protocol: 17, Source: 130.253.35.33, Target: 239.255.255.250
 - UDP Segment:
   - Source Port: 53420, Destination Port: 1900, Length: 282
```

Figure 6.6: *Turtle Sniffer Capture (Windows)*

The rest of this chapter will describe how the data from different network layers is parsed. The full code for the turtle sniffer can be found under Appendix B.



```
Ethernet Frame:
 - Destination: FF:FF:FF:FF:FF:FF, Source: 00:50:56:C0:00:08, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 17, Source: 192.168.230.1, Target: 192.168.230.255
 - UDP Segment:
   - Source Port: 137, Destination Port: 137, Length: 58
   - Data:
     \xff\x6f\x01\x10\x00\x01\x00\x00\x00\x00\x00\x00\x20\x46\x48\x46\x41\x45\x
     42\x45\x45\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41
     \x43\x41\x43\x41\x43\x41\x41\x41\x00\x00\x20\x00\x01

Ethernet Frame:
 - Destination: FF:FF:FF:FF:FF:FF, Source: 00:50:56:C0:00:08, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 17, Source: 192.168.230.1, Target: 192.168.230.255
 - UDP Segment:
   - Source Port: 137, Destination Port: 137, Length: 58
   - Data:
     \xff\x6f\x01\x10\x00\x01\x00\x00\x00\x00\x00\x00\x20\x46\x48\x46\x41\x45\x
     42\x45\x45\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41\x43\x41
     \x43\x41\x43\x41\x43\x41\x41\x41\x00\x00\x20\x00\x01
```

Figure 6.7: *Turtle Sniffer Capture (Linux)*

Parsing Data

Now that we know how to capture and display raw data that we sniffed, we are in good shape to start unpack the headers and access payload data.

To make things easier to follow it is important to call proper names to the data that will be parsed. Figure 6.8 shows the names for the data unit and respective network layer that are implemented under turtle sniffer code.

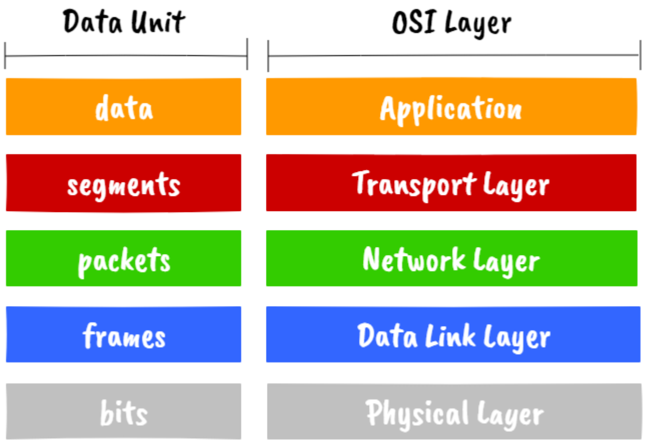


Figure 6.8: *Data Unit and OSI Layers*

To parse raw data, we need to have an idea on how the layers information is structured. The bits received by the Network Card are bundled into frames. Going back to Figure 6.2, frames encapsulate packets, then packets encapsulate segments, and segments encapsulate application data.

Network interface cards handle the physical and data link layers and they provide captured frames to the OS kernel drivers (Figure 6.9). Turtle sniffer code works at the OS level and it will receive frames from Scapy library.

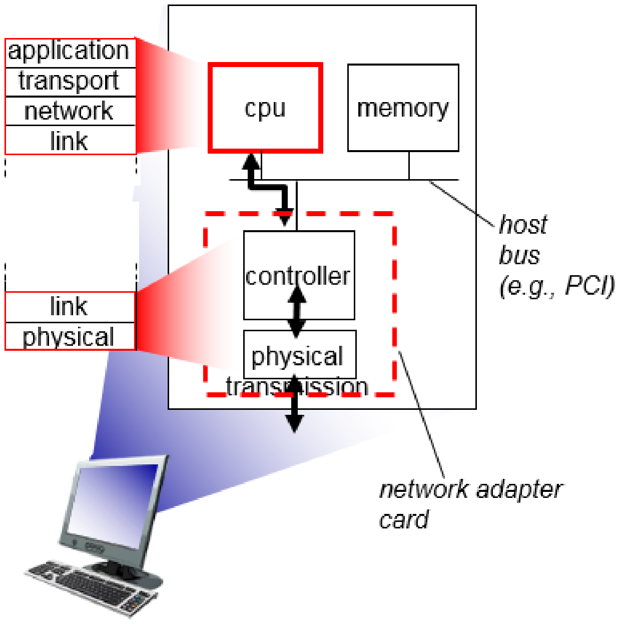
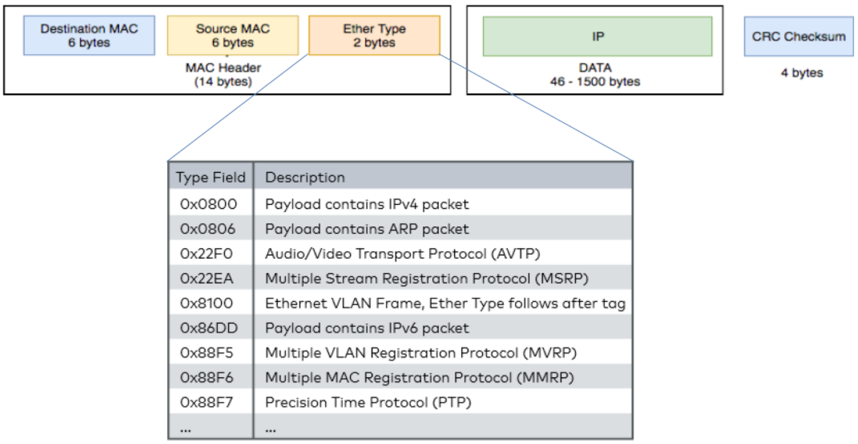


Figure 6.9: *Network Cards and OSI Layers*

Ethernet Frames

Our computers are most likely connected to an Ethernet infrastructure so we will be receiving Ethernet frames. An Ethernet frame structure is shown in Figure 6.10.
The first six bytes are for the Destination MAC address and the next six bytes are for the Source MAC. The last two bytes are for the Ether Type. The rest includes DATA and CRC Checksum.

For the Ether Type, we will focus on receiving IPv4 packets (type field 0x0800).



Type Field	Description
0x0800	Payload contains IPv4 packet
0x0806	Payload contains ARP packet
0x22F0	Audio/Video Transport Protocol (AVTP)
0x22EA	Multiple Stream Registration Protocol (MSRP)
0x8100	Ethernet VLAN Frame, Ether Type follows after tag
0x86DD	Payload contains IPv6 packet
0x88F5	Multiple VLAN Registration Protocol (MVRP)
0x88F6	Multiple MAC Registration Protocol (MMRP)
0x88F7	Precision Time Protocol (PTP)
...	...

Figure 6.10: *Ethernet Frame Structure*

IPv4 Packets

According to RFC 791, an Internet header format looks like Figure 6.11

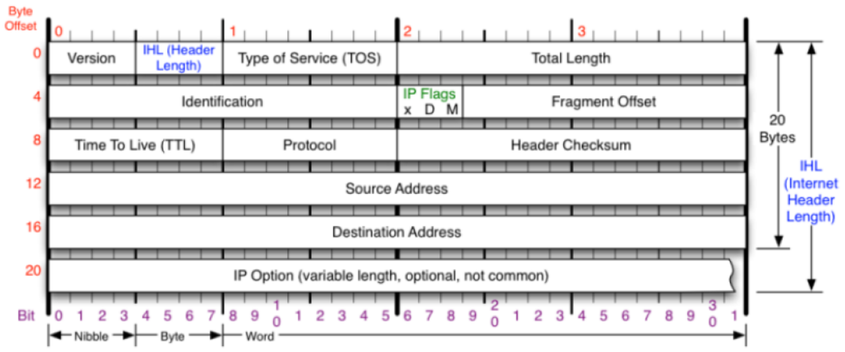


Figure 6.11: *Internet Header Structure*

The IP header includes the following sections:

- Protocol Version (four bits): The first four bits. This represents the current IP protocol.

- Header Length (four bits): The length of the IP header is represented in 32-bit words. Since this field is four bits, the maximum header length allowed is 60 bytes. Usually the value is 5, which means five 32-bit words: 5 * 4 = 20 bytes.
- Type of Service (eight bits): The first three bits are precedence bits, the next four bits represent the type of service, and the last bit is left unused.
- Total Length (16 bits): This represents the total IP datagram length in bytes. This a 16-bit field. The maximum size of the IP datagram is 65,535 bytes.
- Flags (three bits): The second bit represents the Don't Fragment bit. When this bit is set, the IP datagram is never fragmented. The third bit represents the More Fragment bit. If this bit is set, then it represents a fragmented IP datagram that has more fragments after it.
- Time To Live (eight bits): This value represents the number of hops that the IP datagram will go through before being discarded.
- Protocol (eight bits): This represents the transport layer protocol that handed over data to the IP layer.
- Header Checksum (16 bits): This field helps to check the integrity of an IP datagram.
- Source and destination IP (32 bits each): These fields store the source and destination address, respectively.

How to do it...

Following are the steps to parse an Ethernet frame using python under Linux OS:
1. Import the modules required to parse the data.

```
import socket
import struct
import textwrap
```

2. Now we can create a function to parse the Ethernet header:

```
# Unpack ethernet frame
def ethernet_frame(data):
    dest_mac, src_mac, proto =
           struct.unpack('! 6s 6s H', data[:14])
    return get_mac_addr(dest_mac),
           get_mac_addr(src_mac),
           socket.htons(proto), data[14:]
```

Here we use the unpack method from struct module to unpack the headers (the first 14 bytes of data argument that corresponds to frame information). From the Ethernet frame structure, the first six bytes are for the destination MAC, the second 6 bytes are for the source MAC, and the last unsigned short is for the Ether Type. Finally, the rest is payload information (packet information). So, this function returns the destination MAC, source MAC, protocol, and data (remaining payload).

MAC addressess can be properly formatted using **get_mac_addr()** function.

```
# Return properly formatted
# MAC address (ie AA:BB:CC:DD:EE:FF)
def get_mac_addr(bytes_addr):
    bytes_str = map('{:02X}'.format, bytes_addr)
    return ':'.join(bytes_str).upper()
```

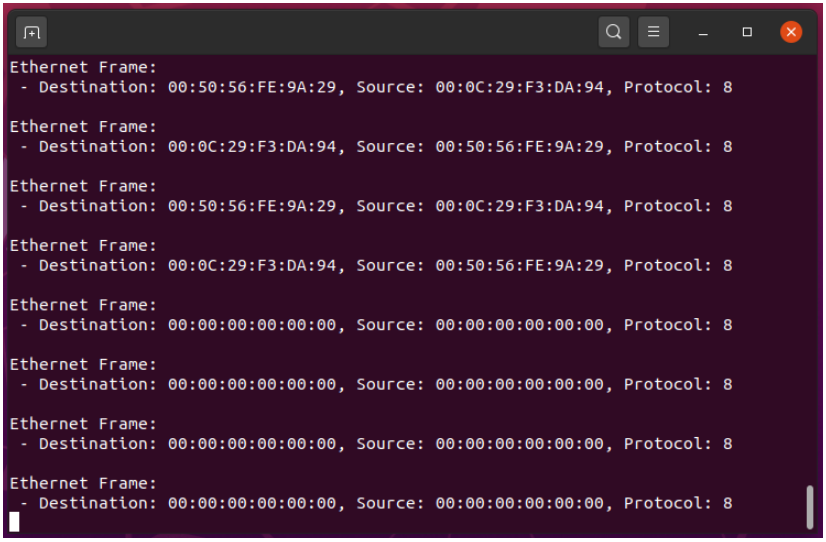
3. Now we can create **turtle_sniffer()** function

```
TAB_1 = ' - '

def turtle_sniffer():
    # Get host
    host = socket.gethostbyname(
                socket.gethostname())
    print("IP: {}".format(host))

    # Linux Version
    conn = socket.socket(socket.AF_PACKET,
                         socket.SOCK_RAW,
                         socket.ntohs(3))

    while True:
        raw_data, addr = conn.recvfrom(65535)
        dest_mac, src_mac, eth_proto, data =
                    ethernet_frame(raw_data)

        print("\nEthernet Frame:")
        print(TAB_1 + "Destination: {},
              Source: {}, Protocol: {}".format(
              dest_mac, src_mac, eth_proto))
```

4. And call it on our main function

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

Up to this step you will be able to capture, parse and display Ethernet frames.



```
Ethernet Frame:
 - Destination: 00:50:56:FE:9A:29, Source: 00:0C:29:F3:DA:94, Protocol: 8

Ethernet Frame:
 - Destination: 00:0C:29:F3:DA:94, Source: 00:50:56:FE:9A:29, Protocol: 8

Ethernet Frame:
 - Destination: 00:50:56:FE:9A:29, Source: 00:0C:29:F3:DA:94, Protocol: 8

Ethernet Frame:
 - Destination: 00:0C:29:F3:DA:94, Source: 00:50:56:FE:9A:29, Protocol: 8

Ethernet Frame:
 - Destination: 00:00:00:00:00:00, Source: 00:00:00:00:00:00, Protocol: 8

Ethernet Frame:
 - Destination: 00:00:00:00:00:00, Source: 00:00:00:00:00:00, Protocol: 8

Ethernet Frame:
 - Destination: 00:00:00:00:00:00, Source: 00:00:00:00:00:00, Protocol: 8

Ethernet Frame:
 - Destination: 00:00:00:00:00:00, Source: 00:00:00:00:00:00, Protocol: 8
```

Figure 6.12: *Turtle Sniffer - Frames Displayed*

5. Now we can check the data section in the Ethernet frame and parse the IP headers. We can create another function to parse IPv4 headers.

```
# Unpack IPv4 Packet
def ipv4_packet(data):
    version_header_len = data[0]
    version = version_header_len >> 4
    header_len = (version_header_len & 15) * 4
    # Start unpacking header
    ttl, proto, src, target =
         struct.unpack('! 8x B B 2x 4s 4s',
                        data[:20])
    return version, header_len, ttl, proto,
           ipv4(src), ipv4(target),
           data[header_len:]
```

As per the IP headers, we will unpack the headers using the unpack method in struct, and return the version, header lentgth, time to live (ttl), transport layer protocol, source and destination IPs.

IP addressess can be properly formatted using **ipv4()** function.

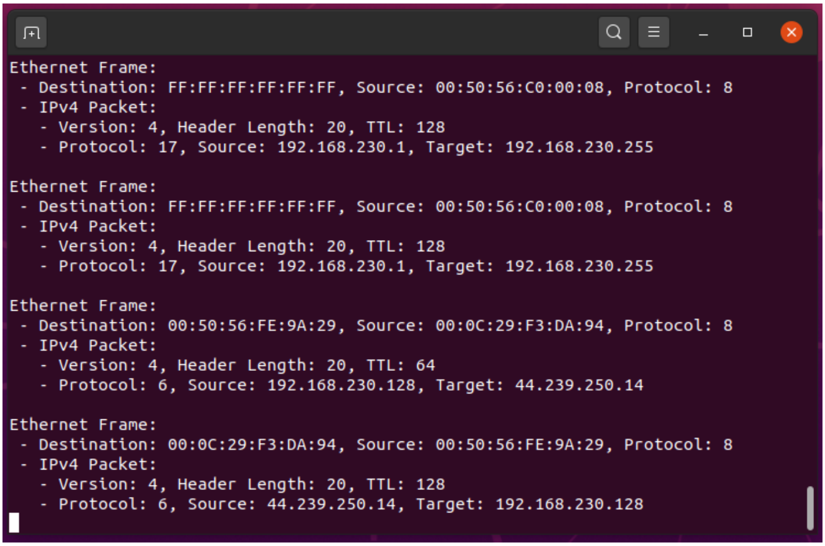
```
# Return properly format IPv4 address
def ipv4(addr):
    return '.'.join(map(str, addr))
```

6. Now update **turtle_sniffer()** function to print IP headers.

```
TAB_1 = ' - '
TAB_2 = '   - '

def turtle_sniffer():
    (...)
    while True:
        (...)
        print("\nEthernet Frame:")
        print(TAB_1 + "Destination: {},
              Source: {}, Protocol: {}".format(
              dest_mac, src_mac, eth_proto))
```

```
        # 8 for IPv4
        if eth_proto == 8:
            (version, header_length, ttl, proto,
            src, target, data) = ipv4_packet(data)

            print(TAB_1 + 'IPv4 Packet:')
            print(TAB_2 + 'Version: {},
                  Header Length: {},
                  TTL: {}'.format(
                  version, header_length, ttl))

            print(TAB_2 + 'Protocol: {},
                  Source: {},
                  Target: {}'.format(
                  proto, src, target))
```

Up to this step you will be able to capture, parse and display Ethernet frames plus IPV4 packets.



```
Ethernet Frame:
 - Destination: FF:FF:FF:FF:FF:FF, Source: 00:50:56:C0:00:08, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 17, Source: 192.168.230.1, Target: 192.168.230.255

Ethernet Frame:
 - Destination: FF:FF:FF:FF:FF:FF, Source: 00:50:56:C0:00:08, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 17, Source: 192.168.230.1, Target: 192.168.230.255

Ethernet Frame:
 - Destination: 00:50:56:FE:9A:29, Source: 00:0C:29:F3:DA:94, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 64
   - Protocol: 6, Source: 192.168.230.128, Target: 44.239.250.14

Ethernet Frame:
 - Destination: 00:0C:29:F3:DA:94, Source: 00:50:56:FE:9A:29, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 6, Source: 44.239.250.14, Target: 192.168.230.128
```

Figure 6.13: *Turtle Sniffer - Frames + Packets Displayed*

Now that we have the network layer unpacked, we are in good shape to start unpack transport layer data.

Segments

Turtle sniffer will focus on the following protocols: UDP, TCP and ICMP.

> [1] ICMP is the Internet Control Message Protocol, a helper protocol that helps Layer 3. ICMP is used to troubleshoot and report error conditions. Without ICMP to help, IP would fail when faced with routing loops, ports, hosts, or networks that are down, etc. ICMP has no concept of ports, as TCP and UDP do, but instead uses types and codes. Commonly used

[1]Bryan Simon, "CISSP Study Guide", 3rd Edition, 2016

ICMP types are echo request and echo reply (used for ping) and time to live exceeded in transit (used for traceroute).

"Which protocol runs at which layer" is often a subject of fierce debate. We call this the “bucket game.” For example, which bucket does ICMP go into: Layer 3 or Layer 4? ICMP headers are at Layer 4, just like TCP and UDP, so many will answer “Layer 4.” Others argue ICMP is a Layer 3 protocol, since it assists IP (a Layer 3 protocol), and has no ports.

This shows how arbitrary the bucket game is: a packet capture shows the ICMP header at Layer 4, so many network engineers will want to answer “Layer 4:” never argue with a packet. The same argument exists for many routing protocols: for example, BGP is used to route at Layer 3, but BGP itself is carried by TCP (and IP). This book will cite clear-cut bucket game protocol/layers in the text and self tests, but avoid murkier examples (just as the exam should).

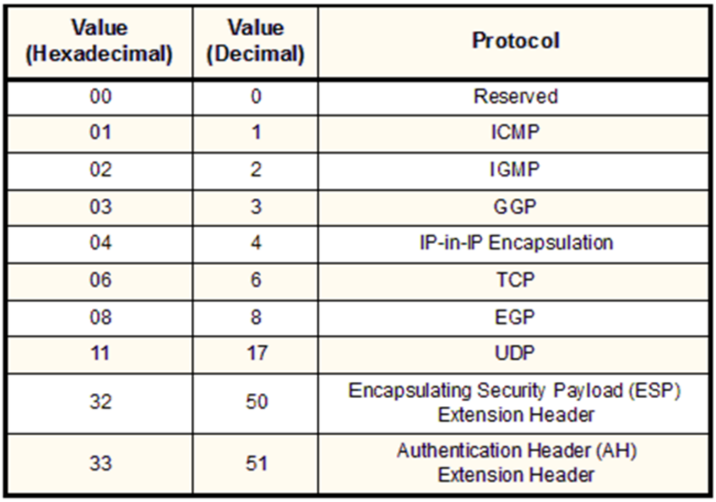
For simplicity of the code, turtle sniffer considers ICMP a layer 4 protocol (transport layer protocol).

From the packet header field **protocol** (Figure 6.11), we can determine which protocol is being encapsulated under the transport layer (Figure 6.14). This information can be found under RFC 1700.

ICMP

An ICMP segment structure is shown in Figure 6.15. It consists of a type (1 bytes), a code (1 byte), and checksum (2 bytes).

7. Let’s create a function to unpack ICMP segments.



Value (Hexadecimal)	Value (Decimal)	Protocol
00	0	Reserved
01	1	ICMP
02	2	IGMP
03	3	GGP
04	4	IP-in-IP Encapsulation
06	6	TCP
08	8	EGP
11	17	UDP
32	50	Encapsulating Security Payload (ESP) Extension Header
33	51	Authentication Header (AH) Extension Header

Figure 6.14: *IPv4 Packet - Protocol Field Options (RFC 1700)*

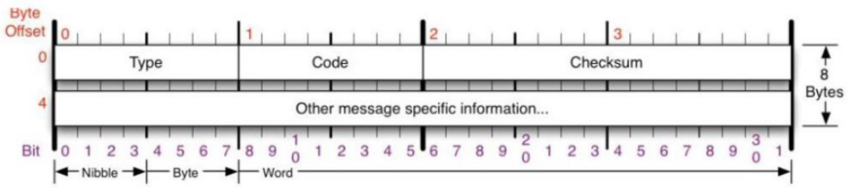


Figure 6.15: *ICMP Segment Structure*

```
# Unpack ICMP packet
def icmp_packet(data):
    icmp_type, code, checksum =
           struct.unpack('! B B H', data[:4])
    return icmp_type, code, checksum, data[4:]
```

Include **format_multi_line()** function to format binary data in a more user friendly way. Any payload data or data that doesn't have a protocol type implemented can be displayed using this function.

```
# Format data display
def format_multi_line(prefix, string, size=80):
    size -= len(prefix)
    if isinstance(string, bytes):
        string = ''.join(r'\x{:02x}'.format(byte)
            for byte in string)
        if size % 2:
            size -= 1
    return '\n'.join([prefix + line for line in
            textwrap.wrap(string, size)])
```

8. Update **turtle_sniffer()** function to print ICMP headers.

```
(...)
DATA_TAB_2 = '   '
DATA_TAB_3 = '     '

def turtle_sniffer():
    (...)
    while True:
        (...)
        # 8 for IPv4
        if eth_proto == 8:
            (...)
            # Check protocols
            #**********************
            # ICMP
            if proto == 1:
                icmp_type, code,
                checksum, data = icmp_packet(data)

                print(TAB_1 + 'ICMP Packet:')
                print(TAB_2 + 'Type: {},
                      Code: {},
                      Checksum: {}'.format(
                      icmp_type, code, checksum))

                print(TAB_2 + 'Data:')
                print(format_multi_line(
                      DATA_TAB_3, data))

            # Other
            else:
                print(TAB_1 + 'Data:')
                print(format_multi_line(
                      DATA_TAB_2, data))
```

UDP

A UDP segment structure is shown in Figure 6.16. It consists of a source (2 bytes) and destination (2 bytes) port, length (2 bytes), and checksum (2 bytes).

9. According to the diagram, we can unpack the UDP segment using the following code:

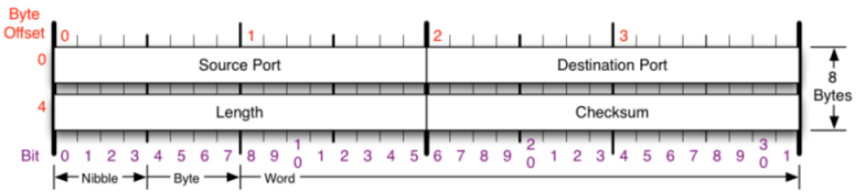


Figure 6.16: *UDP Segment Structure*

```
# Unpack UDP segment
def udp_segment(data):
    src_port, dest_port, length =
          struct.unpack('! H H H 2x', data[:8])
    return src_port, dest_port, length, data[8:]
```

10. Update **turtle_sniffer()** function to print UDP headers.

```
def turtle_sniffer():
    (...)
    while True:
        (...)
        # 8 for IPv4
        if eth_proto == 8:
            (...)

            # Check protocols
            #***********************
            # ICMP
            if proto == 1:
                (...)
            # UDP
            elif proto == 17:
                src_port, dest_port,
                length, data =
                      udp_segment(data)

                print(TAB_1 + 'UDP Segment:')
                print(TAB_2 + 'Source Port: {},
                      Destination Port: {},
                      Length: {}'.format(
                      src_port, dest_port,
                      length))

                print(TAB_2 + 'Data:')
                print(format_multi_line(
                DATA_TAB_3, data))

            # Other
            else:
                (...)
```

TCP

A TCP segment structure is shown in Figure 6.17. We will be parsing source (2 bytes) and destination (2 bytes) port, sequence (4 bytes), acknowledgement number (4 bytes), offset (4 bits) and TCP flags (1 byte).

11. According to the diagram, we can unpack the TCP segment using the following code:

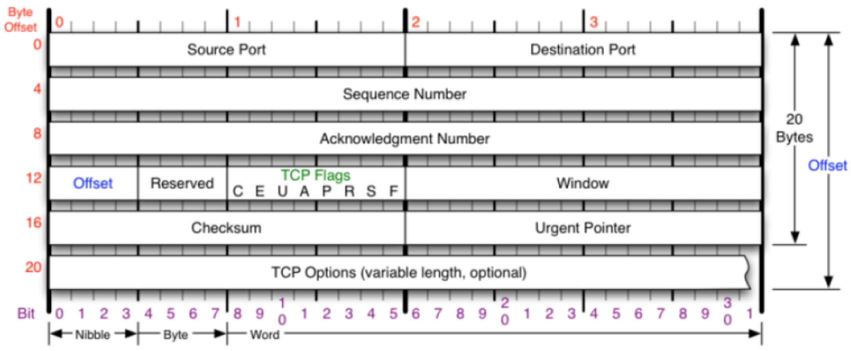


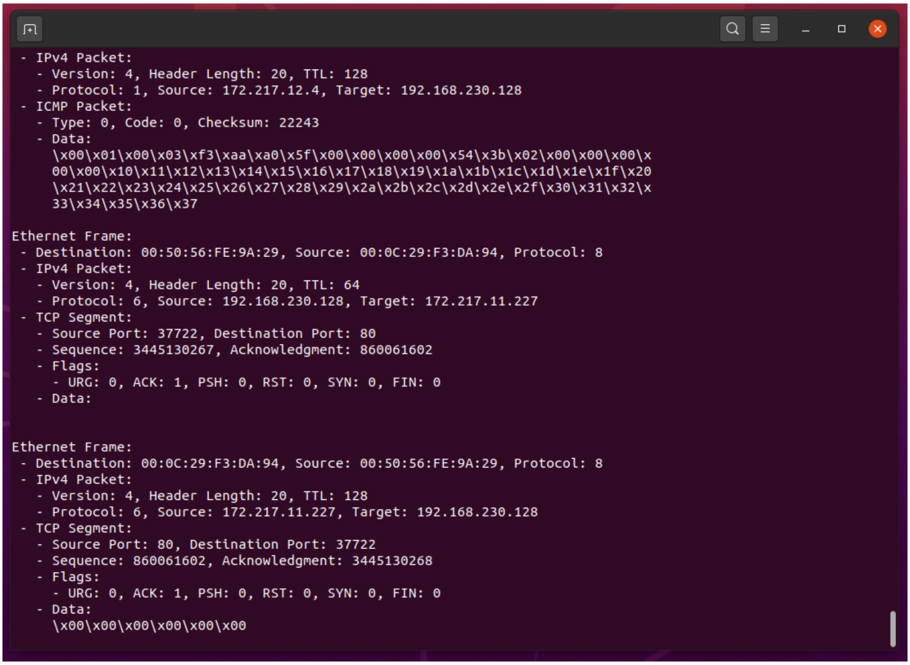
Figure 6.17: *TCP Segment Structure*

```
# Unpack TCP segment
def tcp_segment(data):
    (src_port, dest_port,
    sequence, acknowledgment,
    offset_reserved_flags) =
         struct.unpack('! H H L L H', data[:14])

    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1

    return src_port, dest_port, sequence,
           acknowledgment,
           offset_reserved_flags,
           flag_urg, flag_ack, flag_psh,
           flag_rst, flag_syn, flag_fin,
           data[offset:]
```

12. Update **turtle_sniffer()** function to print TCP headers.

```
def turtle_sniffer():
    (...)
    while True:
        (...)
        # 8 for IPv4
        if eth_proto == 8:
            (...)

            # Check protocols
            #************************
            # ICMP
            if proto == 1:
                (...)

            # UDP
            elif proto == 17:
                (...)
            # TCP
            elif proto == 6:
                src_port, dest_port, sequence,
                acknowledgment,
                offset_reserved_flags,
                flag_urg, flag_ack, flag_psh,
                flag_rst, flag_syn, flag_fin,
                data = tcp_segment(data)

                print(TAB_1 + 'TCP Segment:')

                print(TAB_2 + 'Source Port: {},
                  Destination Port: {}'.format(
                  src_port, dest_port))

                print(TAB_2 + 'Sequence: {},
                  Acknowledgment: {}'.format(
                  sequence, acknowledgment))

                print(TAB_2 + 'Flags:')
                print(TAB_3 + 'URG: {}, ACK: {},
                      PSH: {}, RST: {}, SYN: {},
                      FIN: {}'.format(flag_urg,
                      flag_ack, flag_psh,
                      flag_rst, flag_syn,
                      flag_fin))

                print(TAB_2 + 'Data:')
                print(format_multi_line(
                DATA_TAB_3, data))
            # Other
            else:
                (...)
```

Running the script you will get something similar to Figure 6.18



```
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 1, Source: 172.217.12.4, Target: 192.168.230.128
 - ICMP Packet:
   - Type: 0, Code: 0, Checksum: 22243
   - Data:
     \x00\x01\x00\x03\xf3\xaa\xa0\x5f\x00\x00\x00\x00\x54\x3b\x02\x00\x00\x00\x
     00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20
     \x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x
     33\x34\x35\x36\x37

Ethernet Frame:
 - Destination: 00:50:56:FE:9A:29, Source: 00:0C:29:F3:DA:94, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 64
   - Protocol: 6, Source: 192.168.230.128, Target: 172.217.11.227
 - TCP Segment:
   - Source Port: 37722, Destination Port: 80
   - Sequence: 3445130267, Acknowledgment: 860061602
   - Flags:
     - URG: 0, ACK: 1, PSH: 0, RST: 0, SYN: 0, FIN: 0
   - Data:


Ethernet Frame:
 - Destination: 00:0C:29:F3:DA:94, Source: 00:50:56:FE:9A:29, Protocol: 8
 - IPv4 Packet:
   - Version: 4, Header Length: 20, TTL: 128
   - Protocol: 6, Source: 172.217.11.227, Target: 192.168.230.128
 - TCP Segment:
   - Source Port: 80, Destination Port: 37722
   - Sequence: 860061602, Acknowledgment: 3445130268
   - Flags:
     - URG: 0, ACK: 1, PSH: 0, RST: 0, SYN: 0, FIN: 0
   - Data:
     \x00\x00\x00\x00\x00\x00
```

Figure 6.18: *Turtle Sniffer - Frames + Packets + Segments Displayed*

A

Basic Sniffer Code

Basic Sniffer (Linux)

```python
import socket

# Create PACKET raw socket
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket
    .ntohs(3))

# Capture data
while True:
    print( s.recvfrom(65565) )
```

Listing A.1: *Basic Sniffer (Linux)*

Basic Sniffer (Windows)

```
from scapy.all import *

# List available interfaces
IFACES.show()

# Get interface number
number = input("NIC #> ")

# Get interface name by index
iface = IFACES.dev_from_index( int(number) )

# Create a level two socket
socket = conf.L2socket(iface=iface)

while True:
    packet_raw = socket.recv_raw() # raw data
    print(packet_raw[1])
```

Listing A.2: *Basic Sniffer (Windows)*

B

Turtle Sniffer Code

Turtle Sniffer (Linux)

```
import socket
import struct
import textwrap

TAB_1 = ' - '
TAB_2 = '   - '
TAB_3 = '     - '
TAB_4 = '       - '

DATA_TAB_1 = ' '
DATA_TAB_2 = '   '
DATA_TAB_3 = '     '
DATA_TAB_4 = '       '

def turtle_sniffer():
    # Get host
    host = socket.gethostbyname(socket.gethostname())
    print("IP: {}".format(host))

    # Linux Version
    conn = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

    while True:
        raw_data, addr = conn.recvfrom(65535)
        dest_mac, src_mac, eth_proto, data = ethernet_frame(raw_data)
        print("\nEthernet Frame:")
        print(TAB_1 + "Destination: {}, Source: {}, Protocol: {}".format(
             dest_mac, src_mac, eth_proto))

        # 8 for IPv4
        if eth_proto == 8:
            (version, header_length, ttl, proto, src, target, data) =
                 ipv4_packet(data)
            print(TAB_1 + 'IPv4 Packet:')
            print(TAB_2 + 'Version: {}, Header Length: {}, TTL: {}'.format(
                 version, header_length, ttl))
            print(TAB_2 + 'Protocol: {}, Source: {}, Target: {}'.format(
                 proto, src, target))

            # Check protocols
            #***********************
            # ICMP
            if proto == 1:
                icmp_type, code, checksum, data = icmp_packet(data)
                print(TAB_1 + 'ICMP Packet:')
                print(TAB_2 + 'Type: {}, Code: {}, Checksum: {}'.format(
                     icmp_type, code, checksum))
                print(TAB_2 + 'Data:')
                print(format_multi_line(DATA_TAB_3, data))

            # TCP
            elif proto == 6:
                src_port, dest_port, sequence, acknowledgment,
                     offset_reserved_flags, \
                flag_urg, flag_ack, flag_psh, flag_rst, flag_syn, flag_fin,
                     data = tcp_segment(data)
                print(TAB_1 + 'TCP Segment:')
                print(TAB_2 + 'Source Port: {}, Destination Port: {}'.format
                     (src_port, dest_port))
                print(TAB_2 + 'Sequence: {}, Acknowledgment: {}'.format(
                     sequence, acknowledgment))
                print(TAB_2 + 'Flags:')
                print(TAB_3 + 'URG: {}, ACK: {}, PSH: {}, RST: {}, SYN: {},
                     FIN: {}'.format(flag_urg, flag_ack, flag_psh, flag_rst
                     , flag_syn, flag_fin))
                print(TAB_2 + 'Data:')
                print(format_multi_line(DATA_TAB_3, data))

            # UDP
            elif proto == 17:
                src_port, dest_port, length, data = udp_segment(data)
                print(TAB_1 + 'UDP Segment:')
                print(TAB_2 + 'Source Port: {}, Destination Port: {}, Length
                     : {}'.format(src_port, dest_port, length))
                print(TAB_2 + 'Data:')
                print(format_multi_line(DATA_TAB_3, data))

            # Other
            else:
                print(TAB_1 + 'Data:')
                print(format_multi_line(DATA_TAB_2, data))

# Unpack ethernet frame
def ethernet_frame(data):
    dest_mac, src_mac, proto = struct.unpack('! 6s 6s H', data[:14])
    return get_mac_addr(dest_mac), get_mac_addr(src_mac), socket.htons(proto
         ), data[14:]

```

```
# Return properly formatted MAC address (ie AA:BB:CC:DD:EE:FF)
def get_mac_addr(bytes_addr):
    bytes_str = map('{:02X}'.format, bytes_addr)
    return ':'.join(bytes_str).upper()

# Unpack IPv4 Packet
def ipv4_packet(data):
    version_header_len = data[0]
    version = version_header_len >> 4
    header_len = (version_header_len & 15) * 4
    # Start unpacking header
    ttl, proto, src, target = struct.unpack('! 8x B B 2x 4s 4s', data[:20])
    return version, header_len, ttl, proto, ipv4(src), ipv4(target), data[
         header_len:]

# Return properly format IPv4 address
def ipv4(addr):
    return '.'.join(map(str, addr))

# Unpack ICMP packet
def icmp_packet(data):
    icmp_type, code, checksum = struct.unpack('! B B H', data[:4])
    return icmp_type, code, checksum, data[4:]

# Unpack TCP segment
def tcp_segment(data):
    (src_port, dest_port, sequence, acknowledgment, offset_reserved_flags) =
          struct.unpack('! H H L L H', data[:14])
    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1

    return src_port, dest_port, sequence, acknowledgment,
         offset_reserved_flags, \
           flag_urg, flag_ack, flag_psh, flag_rst, flag_syn, flag_fin, data[
                offset:]

# Unpack UDP segment
def udp_segment(data):
    src_port, dest_port, length = struct.unpack('! H H H 2x', data[:8])
    return src_port, dest_port, length, data[8:]

# Format data display
def format_multi_line(prefix, string, size=80):
    size -= len(prefix)
    if isinstance(string, bytes):
        string = ''.join(r'\x{:02x}'.format(byte) for byte in string)
        if size % 2:
            size -= 1
    return '\n'.join([prefix + line for line in textwrap.wrap(string, size)
         ])
```

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

Listing B.1: *Turtle Sniffer (Linux)*

Turtle Sniffer (Windows)

```
from scapy.all import *
import struct
import textwrap

TAB_1 = ’ - ’
TAB_2 = ’   - ’
TAB_3 = ’     - ’
TAB_4 = ’       - ’

DATA_TAB_1 = ’ ’
DATA_TAB_2 = ’   ’
DATA_TAB_3 = ’     ’
DATA_TAB_4 = ’       ’

def turtle_sniffer():
    # Show interfaces available
    IFACES.show()

    number = input("Network Interface #: ")
    iface = IFACES.dev_from_index(int(number))

    # Create a level two socket
    socket = conf.L2socket(iface=iface)

    try:
        while True:

            # socket is now an ethernet socket
            packet_raw = socket.recv_raw() # raw data

            dest_mac, src_mac, eth_proto, data = ethernet_frame(packet_raw
                 [1])
            print("\nEthernet Frame:")
            print(TAB_1 + "Destination: {}, Source: {}, Protocol: {}".format
                 (dest_mac, src_mac, eth_proto))

            # 8 for IPv4
            if eth_proto == 8:
                (version, header_length, ttl, proto, src, target, data) =
                     ipv4_packet(data)
                print(TAB_1 + ’IPv4 Packet:’)
                print(TAB_2 + ’Version: {}, Header Length: {}, TTL: {}’.
                     format(version, header_length, ttl))
                print(TAB_2 + ’Protocol: {}, Source: {}, Target: {}’.format(
                     proto, src, target))

                # Check protocols
                # ***********************
                # ICMP
                if proto == 1:
                    icmp_type, code, checksum, data = icmp_packet(data)
                    print(TAB_1 + ’ICMP Packet:’)
                    print(TAB_2 + ’Type: {}, Code: {}, Checksum: {}’.format(
                         icmp_type, code, checksum))
                    print(TAB_2 + ’Data:’)
                    print(format_multi_line(DATA_TAB_3, data))

                # TCP
                elif proto == 6:
                    src_port, dest_port, sequence, acknowledgment,
                         offset_reserved_flags, \
                    flag_urg, flag_ack, flag_psh, flag_rst, flag_syn,
                         flag_fin, data = tcp_segment(data)
                    print(TAB_1 + 'TCP Segment:')
                    print(TAB_2 + 'Source Port: {}, Destination Port: {}'.
                         format(src_port, dest_port))
                    print(TAB_2 + 'Sequence: {}, Acknowledgment: {}'.format(
                         sequence, acknowledgment))
                    print(TAB_2 + 'Flags:')
                    print(TAB_3 + 'URG: {}, ACK: {}, PSH: {}, RST: {}, SYN:
                         {}, FIN: {}'.format(flag_urg, flag_ack,
                           flag_psh, flag_rst,
                           flag_syn, flag_fin))
                    print(TAB_2 + 'Data:')
                    print(format_multi_line(DATA_TAB_3, data))

                # UDP
                elif proto == 17:
                    src_port, dest_port, length, data = udp_segment(data)
                    print(TAB_1 + 'UDP Segment:')
                    print(TAB_2 + 'Source Port: {}, Destination Port: {},
                         Length: {}'.format(src_port, dest_port, length))
                    print(TAB_2 + 'Data:')
                    print(format_multi_line(DATA_TAB_3, data))

                # Other
                else:
                    print(TAB_1 + 'Data:')
                    print(format_multi_line(DATA_TAB_2, data))

    except KeyboardInterrupt:
        print("turtle> Bye.")

# Unpack ethernet frame
def ethernet_frame(data):
    try:
        dest_mac, src_mac, proto = struct.unpack('! 6s 6s H', data[:14])
        return get_mac_addr(dest_mac), get_mac_addr(src_mac), socket.htons(
             proto), data[14:]
    except:
        dest_mac = ""
        src_mac = ""
        proto = ""
        return dest_mac, src_mac, proto, ""

# Return properly formatted MAC address (ie AA:BB:CC:DD:EE:FF)
def get_mac_addr(bytes_addr):
    bytes_str = map('{:02X}'.format, bytes_addr)
    return ':'.join(bytes_str).upper()

# Unpack IPv4 Packet
def ipv4_packet(data):
    version_header_len = data[0]
    version = version_header_len >> 4
    header_len = (version_header_len & 15) * 4
    # Start unpacking header
    ttl, proto, src, target = struct.unpack('! 8x B B 2x 4s 4s', data[:20])
    return version, header_len, ttl, proto, ipv4(src), ipv4(target), data[
         header_len:]

# Return properly format IPv4 address
def ipv4(addr):
    return '.'.join(map(str, addr))

# Unpack ICMP packet
def icmp_packet(data):
    icmp_type, code, checksum = struct.unpack('! B B H', data[:4])
    return icmp_type, code, checksum, data[4:]

# Unpack TCP segment
def tcp_segment(data):
    (src_port, dest_port, sequence, acknowledgment, offset_reserved_flags) =
          struct.unpack('! H H L L H', data[:14])
    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1

    return src_port, dest_port, sequence, acknowledgment,
         offset_reserved_flags, \
           flag_urg, flag_ack, flag_psh, flag_rst, flag_syn, flag_fin, data[
                offset:]

# Unpack UDP segment
def udp_segment(data):
    src_port, dest_port, length = struct.unpack('! H H H 2x', data[:8])
    return src_port, dest_port, length, data[8:]

# Format data display
def format_multi_line(prefix, string, size=80):
    size -= len(prefix)
    if isinstance(string, bytes):
        string = ''.join(r'\x{:02x}'.format(byte) for byte in string)
        if size % 2:
            size -= 1
    return '\n'.join([prefix + line for line in textwrap.wrap(string, size)
         ])

#------------------------------------
if __name__ == "__main__":
    turtle_sniffer()
```

Listing B.2: *Turtle Sniffer (Windows)*