Python for Developers

Learn to Develop Efficient Programs using Python

MOHIT RAJ

bpb

Python for Developers

Learn to Develop Efficient Programs using Python

by

Mohit Raj

FIRST EDITION 2020

ISBN: 978-81-94401-872

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My Mother and Father

About the Author

Mohit Raj is a Python programmer with a keen interest in the field of information security. He has completed his Bachelor's degree (B.tech)in Computer Science from Kurukshetra University, Kurukshetra, and a Master's in Engineering (2012) in Computer Science from Thapar University, Patiala. He is a CEH, ECSA from EC-Council USA. He has worked in IBM, Teramatrix (Startup), and Sapient. He has been pursuing a Ph.D. degree in Blockchain from Thapar Institute of Engineering & Technology under Dr. Maninder Singh for two years. Mohit has published several articles in national and international magazines. He is the author of Python Penetration Testing Essentials, Python: Penetration Testing for Developers, and Learn Python in 7 days. Apart from studies, he has a green belt in Karate-do. He is a gym lover and a passionate learner of the Vedic astrology.

His blog link: **https://learningpie.club/**

LinkedIn Profile: **https://www.linkedin.com/in/mohit-990a852a/**

About the Reviewers

- **Bhaskar Das** worked for 10 years with IBM. He has experience of different technologies and tools. He has worked with Java and Python. Now he is working as a freelancer. He has authored "Learn Python in 7 days". He has also authored several technical articles.
- **Afsana Atar** is an accomplished test engineer with over 10 years of extensive experience in software testing. She extends her thought leadership to teams in a variety of domains from digital advertising, education & healthcare to the financial sector in banking, insurance & trading. She has worked for various organizations including Google, IBM, Principal Financial Group and The Children's Hospital of Philadelphia. Currently, she works for "Susquehanna International Group" a financial trading firm. She is a Certified Scrum Master (CSM), an Agile scrum practitioner and part of the scrum alliance community. She has managed and worked on projects worth over one million dollars in various capacities as Quality Assurance Engineer to QA Manager. She believes in sharing her experiences with the testing community to help foster greater learning & innovation.

Acknowledgement

First of all, I am grateful to the Almighty for helping me complete this book. I would like to thank my mother for her love and encouraging support and my father for raising me in a house with desktops and laptops. A big thanks to the CEO of BPB publication Manish Jain for giving me the opportunity to write this book.

I would also like to thank everyone who has contributed to the publication of this book, including the publisher, especially the technical reviewers and also the editor Priyanka. Last but not least, I'm grateful to my i7 Dell laptops, without which it would not have been possible to write this book. Finally, I thank Guido Van Rossum, the creator of Python.

Preface

The primary goal of this book is to provide information and skills that are necessary to be a core Python developer. The book begins with basics and goes beyond basics. After reading this book, you'll be able todevelop cool Python applications. Over the 20 chapters in this book, you will learn the following:

Chapter 1 introduces the installation of Python and shows how to write and execute the first program. This chapter gives the knowledge of basic syntax such astriple quotes, escape sequence, and formatted output.

Chapter 2 discusses the different types of operators offered by Python.

Chapter 3 elaborates on the control statement, uses of the for and while loop, and then how to control aloop with the help of break and continue statements.

Chapter 4 discusses the string,the first data structure of Python, features ofstring, string methods and the functions can be applied on the string.

Chapter 5 gives you the knowledge of two main containers of Python: list and tuple. This chapter mentions the function and method of list and tuple in a step-by-step manner.

Chapter 6 describes another important data structure dictionary. The dictionary contains its own index of values; the index is called as key. The chapter also gives you the details of the dictionary's methods.

Chapter 7 talks about the creation and execution of a function and the removal of aredundancy using the function. The chapter then focuses on the local and global variables.

Chapter 8 focuses on organizing the code and creation of modules and packages.

Chapter 9 teaches how to handle an error and make our own customize exceptions.

Chapter 10 describes the reading and writing of a file. The chapter also mention the pickle and JSON files.

Chapter 11 describes a special type of contain called collections which offers counter, named tuple, defaultdict, ordereddict, and deque.

Chapter 12 teaches how to create a random number with the help of random modules. The chapter also describes some built-in functions such as filter, map, and reduce, which gives greater flexibility to create code.

Chapter 13 introduces the time-related activities and explains how to generate time of different time zones.

Chapter 14 describes the regular expression, which helps to find the desired pattern from the text.

Chapter 15 mentions the os modules that help in running the OS command and howthe os modules help in creating a directory and checking access of a file.

Chapter 16 gives you a detailed description of class and object. This chapter will teach you about instance variable, class variable, method, class method, static method, and so on.

Chapter 17 focuses on parallel programming with the help of threads. The threading module has been used to create the threads.

Chapter 18 describes the significance of queue and types of queue and how a queue can help to communicate two threads.

Chapter 19 mentions the real parallel programming through multiprocessing. The chapter also focuses on the subprocess that could replace the os module.

Chapter 20 describes the modules such as argparse, logging, config parser, and PDB, which helps creating and debugging the code efficiently.

Downloading the code bundle and coloured images:

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

https://rebrand.ly/f483c

Errata

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

errata@bpbonline.com

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

Table of Contents

1. Introduction to Python 1
Structure 1
Objective 1
What is Python? 2
Reasons to choose Python 2
Multi-purpose 2
Vast library and module support 3
Readability 3
Object-Oriented 3
Platform independent 3
Python is dynamically typed and strongly typed 3
Python installation 4
Installation of Python 3.x in Windows 10 4
Installation of Python in Linux 6
Basic Python syntax 9
Print statement 9
Saving the program 10
Triple quotes 10
Python Back Slash \ 11
Escape sequence of string 11
Python formatted output 12
Conclusion 12
Questions 13

2. Python Operators 15
Structure 15
Objective 15
Variables 15
Assignment statement 16
Multiple assignment 16
Numeric data types 17

Integer numbers 18
Floating-point numbers 18
Python character sets 18
Conversion functions 19
Operators 20
Arithmetic operators 20
Type conversions 21
Comparison operators 22
Assignment operators 23
Bitwise operators 24
Logical operators 25
Membership operators 26
Identity Operators 27
Operator precedence 28
Advance part 29
Conclusion 30
Questions 30

3. Control Statement and Loop 31
Structure 31
Objective 32
Conditional statement 32
If and if-else statements 32
if statement 33
Tabs or Spaces? 34
if-else statement 34
if-elif-else structure 35
Range() 37
Loops 39
For loop 39
Printing number 0 to 9 40
Printing the numbers in horizontal fashion 40
For loop with string 41
Exercise 1 42

Exercise 2 ..... *43*
while loop 43
Break statement 45
Break statement with the while loop *46*
Continue statement 47
else statement 48
pass statement 49
Conclusion 50
Questions 50

4. Strings 51
Structure 51
Objective 51
Indexing using subscript operator 53
Slicing for substrings 54
String methods 56
count() *56*
find() *57*
Justify methods *57*
ljust() *57*
rjust() *58*
center() *59*
zfill() *59*
Case methods *60*
lower() *60*
upper() *60*
capitalize() *61*
swapcase() *61*
Strip methods *61*
replace() *62*
Split methods *63*
Partition methods *64*
Join method *65*
String Boolean methods *65*
startswith() *65*

18+

WHAT IS AVAXHOME?

endswith() *66*
isdigit() *67*
isalpha() *67*
isalnum() *68*
isspace() *68*
format() *68*
String functions 69
Max() *69*
min() *70*
Conclusion 70
Question 70

5. Tuple and List 73
Structure 73
Objective 74
Tuple 74
Creating tuple *74*
Empty tuple *74*
Creating tuple with the items *75*
Indexing tuple 75
Slicing of tuple *76*
Tuple methods 78
index() *79*
Tuple functions 79
len() *79*
max() *79*
Min() *80*
Operations of Tuples 80
Addition of tuples *80*
Multiplication of tuple *80*
In operator *81*
for loop with tuple 81
Unpacking of tuple 82
List 83
Creating a list *83*

Empty list ... *83*
List with the elements ... *83*
List operations ... 84
Accessing item of a list ... *84*
Updating list ... *84*
Deleting list item ... *85*
Addition of the lists ... *85*
Multiplication of List ... *86*
in operator ... *86*
List with for loop ... *86*
List functions ... 87
len() ... *87*
max() ... *87*
List methods ... 89
Insert methods ... 89
append() ... *89*
insert() ... *91*
Deletion ... 91
remove() ... *91*
pop() ... *92*
count() ... *93*
index() ... *93*
Copy() ... *94*
Sort() ... *95*
reverse() ... *98*
List comprehensions ... 99
Exercise ... 100
Conclusion ... 100
Questions ... 101

6. Dictionary and Sets ... 103
Structure ... 103
Objective ... 103
Dictionary ... 104

Creating a dictionary *104*
Features of dictionary *104*
Operations on dictionary *104*
Accessing the values of dictionary *105*
Deleting item from dictionary *105*
Updating and adding in the dictionary *106*
Adding item to dictionary *106*
Dictionary functions *107*
len(dict) *107*
Max(dict) *107*
Dictionary methods *107*
copy() *108*
get() *108*
setdefault() *109*
items() *110*
keys() *111*
values() *111*
update() *113*
Exercise *114*
Set 115
With item *115*
Without items *115*
add() *116*
remove() *116*
Conclusion 117
Questions 117

7. Function 119
Structure 119
Objective 120
What is function? 120
Defining a Python function 120
Function with positional arguments 121
Function with the arguments and return value 122
Function with default argument 123

Function with variable-length arguments 124
Function with keyworded arguments 125
Argument pass by reference or value 126
Scope 127
Types of scope 127
Local scope 128
Enclosing scope 128
Global scope 128
Built-in scope 128
Memory management 129
Scope of variables 129
Conclusion 133
Questions 133

8. Module 135
Structure 135
Objective 136
Module 136
The import statement 136
The from statement 137
Locating Python modules 140
Compiled Python files 142
dir() 142
The __name__ statement 143
Python package 145
Importing the modules from different path 147
Conclusion 148
Questions 148

9. Exception Handling 149
Structure 149
Objective 149
Exception 150
Try statement with an except clause 150

Multiple exception block 151
else in exception 152
finally statement 153
Program find its exception type 154
Raising an exception 155
Advance section 157
User-defined exceptions 157
Exercise 161
Conclusion 162
Questions 162

10. File Handling 163
Structure 163
Objective 164
Text files 164
Reading text from a file 164
Writing text to a file 167
The with statement 172
Pickle 174
Reading data from file and unpickling 175
JSON with Python 176
Exercise 178
Conclusion 180
Questions 180

11. Collections 181
Structure 181
Objective 182
Counter 182
Counter methods 183
update() 183
Counter operations 185
Addition 185
Subtraction 186

Union *186*
Intersection *187*
Deque 187
Deque populating *188*
deque consuming *189*
deque rotating *190*
Namedtuple 191
The default dictionary 193
Function as default_factory *193*
int as default factory *194*
list as default_factory *195*
The ordered dictionary 195
Sorted of dictionary based upon key *196*
Sort the dictionary based upon values *197*
Conclusion 198
Question 198

12. Random Modules and Built-in Function 199
Structure 199
Objective 200
The random module 200
Random functions for integers *200*
randint() *200*
randrange() *201*
Random functions for sequence *202*
Choice(rnlist) *202*
shuffle() *203*
Sample() *203*
Random functions for floats *204*
random() *204*
Uniform(start, end) *204*
Exercise 204
The Tombola game *205*
The OTP generator *205*
Python special functions 206

Lambda *206*
filter() *207*
map() *208*
reduce() *209*
isinstance() *210*
eval() *211*
repr() 212
Conclusion 213
Questions 214

13. Time 215
Structure 215
Objective 216
The time module 216
Current Epoch time *216*
Current time *217*
Creating time difference *219*
Conversion of human-readable date to epoch time *219*
time.sleep(second) *220*
The datetime module 221
datetime.datetime.now() *221*
datetime.timedelta class *221*
Dealing with Timezone with the pytz module 222
The calendar module 226
Printing a full month *227*
Printing a year *228*
Curious case of 1752 ..... *229*
Checking the leap year *231*
Conclusion 232
Questions 232

14. Regular Expression 233
Structure 233
Objective 234

Regular expression ... 234
Regular expression functions ... 234
match() ... *234*
search() ... *235*
sub() ... *236*
findall() ... *236*
re.compile(pattern) ... *237*
Special characters ... 238
. (dot) ... *238*
^ (Caret) ... *239*
$ (dollor) ... *239*
** (star)* ... *240*
+ (plus) ... *240*
? ... *241*
{m} ... *242*
{m,n} ... *242*
[] ... *243*
[^] ... *244*
\w ... *245*
Exercise ... 245
Conclusion ... 246
Questions ... 247

15. Operating System Interfaces ... 249
Structure ... 249
Objective ... 249
Getting the OS name ... 250
os.environ ... *251*
Directory and file accessing functions ... 252
os.access() ... *252*
os.rename(old, new) ... *253*
os.stat() ... *254*
Directory functions ... *254*
os.getcwd() ... *254*

os.chdir() *255*
File and folder listing 256
os.listdir(path) *257*
os.walk() *257*
Executing OS command 259
os.popen() *259*
os.system() *261*
Exercise 1 261
Exercise 2 263
Conclusion 264
Questions 264

16. Class and Objects 265
Structure 265
Objective 266
Class 266
Object 267
Instance variable 268
The __init__ method or constructor 269
Regular method *270*
Class variable 274
Class inheritance 278
Multilevel inheritance *286*
Multiple inheritance *288*
Operator overloading 292
Class method 299
Static method 302
Private method and private variable 303
Decorator @property @setter and @deleter 305
Callable objects 308
Conclusion 309
Questions 310

17. Threads 311
Structure 311
Objective 312
Thread 312
Thread creation using class 312
Thread creation using function 313
Important threading methods 313
The join method 315
The join method with time 318
The Daemon thread 320
Lock 322
Problem 1 323
Problem 2 325
Lock versus Rlock 327
GIL 327
Where to use multithreading? 330
What is internal delay? 330
How many threads? 332
Conclusion 338
Questions 339

18. Queue 341
Structure 341
Objective 341
Queue 342
FIFO queue 342
LIFO queue 344
Priority queue 345
Queue with threads 345
Conclusion 352
Questions 352

19. Multiprocessing and Subprocess 353
Structure 353
Objective 354

Python multi-processing 354
The Process class *354*
Killing a Process *357*
The Daemon process *358*
The communication between the processes 358
Shared memory *360*
Value *360*
Array *362*
The Manager class *364*
Exchanging object through the communication channel *365*
Pipe *367*
Subprocess 367
Difference between subprocess and multi-processing *368*
The call() function *368*
Popen() *371*
Conclusion 372
Questions 372

20. Useful Modules 373
Structure 373
Objective 374
Configparser 374
Loggers 377
Argparse 382
The positional argument *383*
Positional arguments with Help message and Type *383*
The Argparse optional argument *384*
nargs *385*
Subparser *387*
Debugging 389
Setting a breakpoint *393*
Conclusion 395
Questions 395

Chapter 1
Introduction to Python

When it comes to rapid development, one language that always comes to mind is the Python programming language. Sometimes, people come up with great ideas, but they are unable to implement it, due to the complexity of the languages learned in the academics. Python has gained a lot of market attention recently. I always say, programming in Python is like programming at the speed of thinking. Python syntaxes are like English syntaxes. According to the IEEE spectrum ranking of 2017 and 2018, Python got the first rank, although different languages have different needs and different domains of interest. But the IEEE spectrum chose Python, which rules the roost. Their selection criteria is described on their website: **https://spectrum.ieee.org/static/ieee-top-programming-languages-2018-methods**.

Structure

- What is Python
- Python installation
- Basic Python syntax

Objective

With this chapter, you will start your journey to Python programming. You will learn how to install the Python software on Windows, as well as on Linux. After that,

you will write your first program. You will learn the significance of triple quotes and the escape sequence. At the end of the chapter, you will see the formatted output.

What is Python?

Python is a general-purpose, high-level language that is used to solve modern-day, computer problems. These days, people have a misconception about Python. They think python is data analytics and machine learning language. However, Python is actually a general-purpose, programming language. Guido van Rossum invented the Python programming language in the early 1990s.

Where is Python Language used?

The better question to ask would be, what is Python NOT used for. Please see below the demanding fields, where Python is being used.

- Data science
- Machine learning
- Web application development
- Network monitoring
- Game development
- Natural language processing
- IoT

Top companies using Python:

The following companies are using Python in their projects:

- Google
- Facebook
- Instagram
- Spotify
- Quora
- Netflix
- Dropbox
- Reddit

Reasons to choose Python

There are many reasons you should use Python.

Multi-purpose

Python is a multipurpose language. The developers are using python Data Analytics,

Machine Learning, AI, Web Application, Network Monitoring, ETL scripts, Hacking, and much more.

Vast library and module support

Python comes with a huge community support. There are different libraries and modules that are available in Python for two different purposes as shown in the following table:

Field	Library, Module, Framework
Data analytics	Numpy, scipy, scikit learn
Deep learning	Pytorch, TensorFlow
Network Hacking	Scrapy
Web scrapping	Scrapy
Natural language programming	NLTK
Web application	Django, Flask
Multiprocessing	Celery

Table 1.1

Readability

Python code is easy to read and understandable. It does not contain big syntax, like Java. Python uses indentation to manage the blocks of code, so indentation is the indispensable part of python programming. With indentation, the code becomes easy to read. Python's syntax is human readable and concise. As a novice, this will help you pick up the fundamentals quickly, with less mental strain, and level up to advanced topics faster.

Object-Oriented

Python has the power of **object-oriented programming (OOPS)**, although you can write the program without defining any class. Whereas it is mandatory to use OOPS in Java, Python offers object-oriented programming as an option, if you are comfortable with OOPS, then choose it, else it can be avoided.

Platform independent

Python codes are platform-independent; it is a matter of copy and paste.

Python is dynamically typed and strongly typed

In dynamically typed, the data-type of a variable is interpreted at run time. In Python, there is no need to define the data-type like int, float, and such others. the following example shall provide more clarification about a variable:

```
>>> a = 10
>>> type(a)
<class ‘int’>
>>> b = 10.9
>>> type(b)
<class ‘float’>
>>>
```

In strongly typed, the type of variable does not change at run time. If `a = 10`, then it remains 10 throughout the execution, until we reassign the variable.

Python installation

Python comes up with two versions, Python 2.x and Python 3.x. The following are the general steps to install Python, although we will discuss the installation steps in detail as well:

1. Download Windows Python installer from the official website of Python **https://www.python.org/download/**. Run the Python installer; the installation is straightforward.
2. Accept the default configuration.
3. Once you are done with installing Python, you have it on your computer in the `C:/python` folder. If you are a Linux lover, then the good news is that Linux CentOS 6 version has come up with version 2.6. However, you can install Python 3.x.

You can install both versions in Windows as well as in Linux.

Installation of Python 3.x in Windows 10

To begin with, download the suitable, executable file according to your computer configuration from **https://www.python.org/downloads/windows/**.

After downloading the file, run the file, the installation is straightforward, but take care at the step as shown in the *Figure 1.1*.

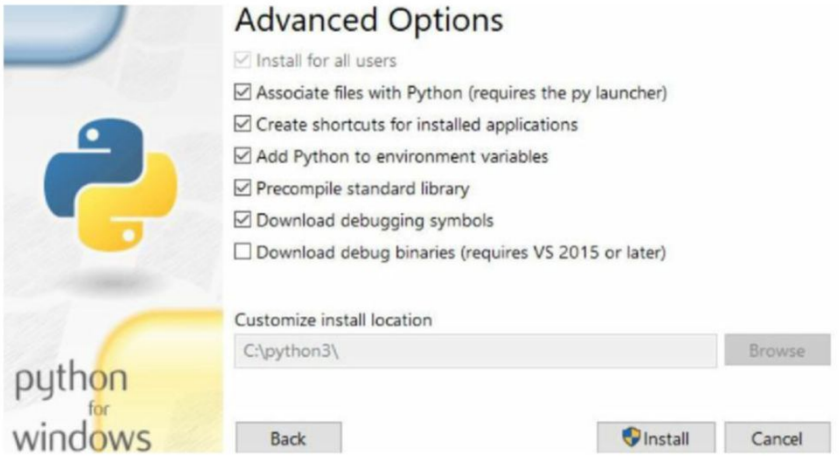

Figure 1.1

Give the customized path of your choice, as shown in *Figure 1.1*. Once you give the path, the installation is done. If you have Python 2.7 in your PC, then go to the installation path of `Python 3` and make Python.exe as Python3.exe. By doing this, you can use both the versions of Python at the same time. After the installation of Python, add the Python path to the environment variable, as shown in the *Figure 1.2*. *Figure 1.2* showcases the steps involved in setting the path of the Python interpreter:

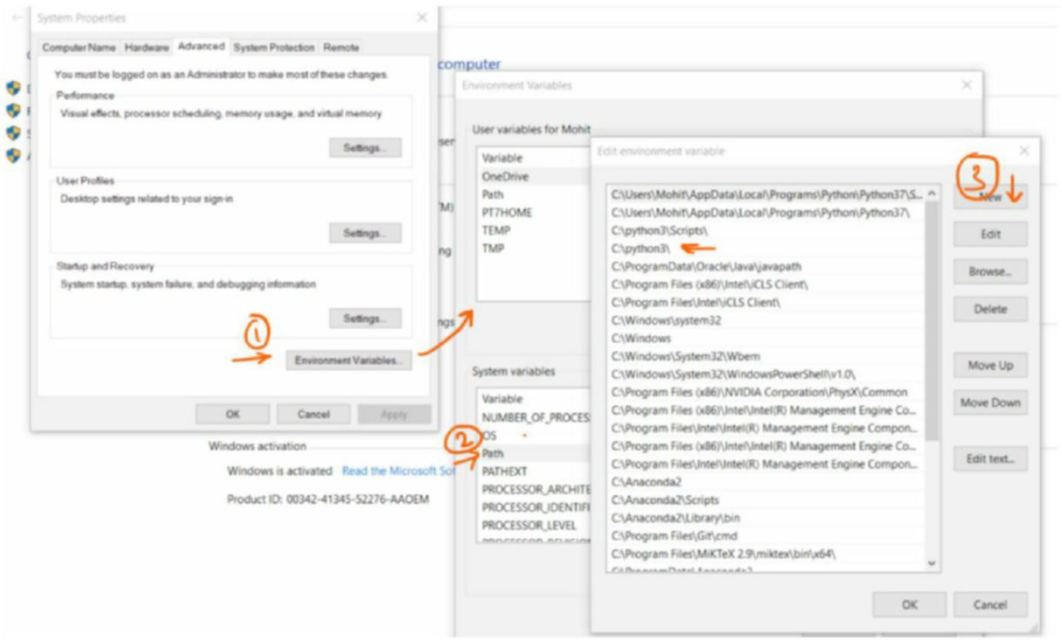

Figure 1.2

After setting the environment variable, you may have to restart the computer. After restarting, open the command prompt. See *Figure 1.3*:

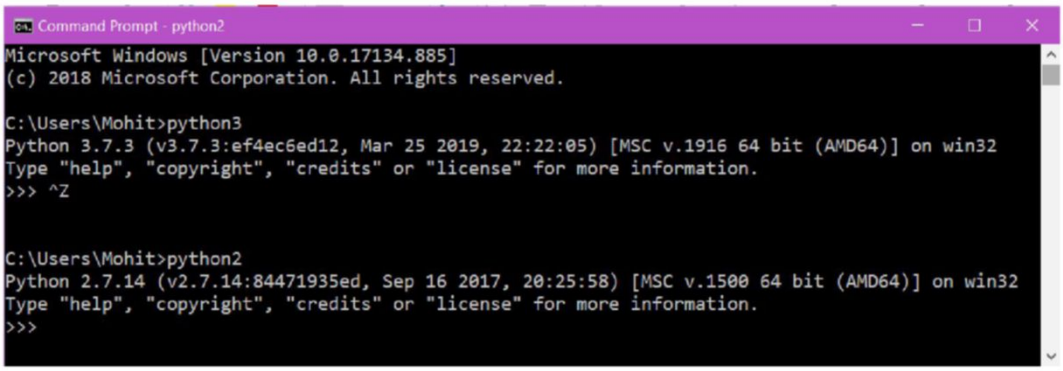

Figure 1.3

The installation of Python in Windows is an easy task.

Installation of Python in Linux

Generally, Python comes with Linux. If python is not pre-installed, then you can download the Python compressed tar file from the official website of Python. Once you download the tar file, you have to extract it.

The command to extract the tar file is as mentioned below:

```
# tar -xvzf Python-3.7.X.tgz
```

After the execution of the command, browse the directory `python-3.7.X` and type the following command:

```
# ./configure
```

The command may need `sudo` permission.

After the successful run of the preceding command, use the following command:

```
# make
```

Then type the following command:

```
# make install
```

If Python is already installed and you want a different version, then you can use the virtual environment to install the Python of your choice.

Let us see how to install using the virtual environment If you are a normal user, then use sudo with commands, as showcased below:

```
$ apt-get install virtualenv
```

The installation process is being displayed in *Figure 1.4:*

```
mohit@node5:~$ sudo apt-get install virtualenv
[sudo] password for mohit:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  snapd-login-service
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  python3-virtualenv
The following NEW packages will be installed:
  python3-virtualenv virtualenv
0 upgraded, 2 newly installed, 0 to remove and 12 not upgraded.
Need to get 47.6 kB of archives.
After this operation, 171 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://in.archive.ubuntu.com/ubuntu xenial-updates/universe amd64 python3-virtualenv all 15.0.1+ds-3ubuntu1 [43.2 kB]
Get:2 http://in.archive.ubuntu.com/ubuntu xenial-updates/universe amd64 virtualenv all 15.0.1+ds-3ubuntu1 [4,342 B]
Fetched 47.6 kB in 0s (57.3 kB/s)
Selecting previously unselected package python3-virtualenv.
(Reading database ... 213545 files and directories currently installed.)
Preparing to unpack .../python3-virtualenv_15.0.1+ds-3ubuntu1_all.deb ...
Unpacking python3-virtualenv (15.0.1+ds-3ubuntu1) ...
Selecting previously unselected package virtualenv.
Preparing to unpack .../virtualenv_15.0.1+ds-3ubuntu1_all.deb ...
Unpacking virtualenv (15.0.1+ds-3ubuntu1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up python3-virtualenv (15.0.1+ds-3ubuntu1) ...
Setting up virtualenv (15.0.1+ds-3ubuntu1) ...
mohit@node5:~$
```

Figure 1.4

After installation, write the following command:

```
$ virtualenv <project-name>
```

The command makes a virtual environment called book, as shown in *Figure 1.5:*

```
mohit@node1:~$ virtualenv book
Running virtualenv with interpreter /usr/bin/python2
New python executable in /home/mohit/book/bin/python2
Also creating executable in /home/mohit/book/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
mohit@node1:~$ cd book/
```

Figure 1.5

Check the content of the virtual environment, the content should be as shown in *Figure 1.6:*

```
mohit@node1:~/book$ ls
bin  lib  local  pip-selfcheck.json  share
mohit@node1:~/book$
```

Figure 1.6

Then browse the project directory, as showcased below.

From the official website (**https://www.python.org/downloads/release/python-373/**), download the Gzipped source tarball file, and put the file in the virtual environment directory book:

```
mohit@node1:~/book$ ls -ltr
total 22456
drwxrwxr-x 3 mohit mohit     4096 Jul 13 11:23 lib
drwxrwxr-x 3 mohit mohit     4096 Jul 13 11:23 share
drwxrwxr-x 2 mohit mohit     4096 Jul 13 11:23 local
-rw-rw-r-- 1 mohit mohit       61 Jul 13 11:23 pip-selfcheck.json
drwxrwxr-x 2 mohit mohit     4096 Jul 13 11:23 bin
-rwxrw-rw- 1 mohit mohit 22973527 Jul 13 11:30 Python-3.7.3.tgz
mohit@node1:~/book$
mohit@node1:~/book$
```

Figure 1.7

Use the following command to extract the tar-zip file.

```
tar -xvzf Python-3.7.3.tgz
```

After running the command, a directory `Python-3.7.3` will be created, as shown in the following screenshot:

```
mohit@node1:~/book$ ls -ltr
total 22460
drwxrwxr-x  3 mohit mohit     4096 Jul 13 11:23 lib
drwxrwxr-x  3 mohit mohit     4096 Jul 13 11:23 share
drwxrwxr-x  2 mohit mohit     4096 Jul 13 11:23 local
-rw-rw-r--  1 mohit mohit       61 Jul 13 11:23 pip-selfcheck.json
drwxrwxr-x  2 mohit mohit     4096 Jul 13 11:23 bin
-rwxrw-rw-  1 mohit mohit 22973527 Jul 13 11:30 Python-3.7.3.tgz
drwxr-xr-x 18 mohit mohit     4096 Jul 13 11:38 Python-3.7.3
mohit@node1:~/book$
```

Figure 1.8

Use the following command to activate the virtual environment:

```
source bin/activate
```

Once the virtual environment activated, run the following command one-by-one.

Browse the `Python-3.7.3` directory using the cd command:

```
./configure
```

After the successful run of the preceding command, use the following command:

```
Make
```

Then the following command:

```
make install
```

By doing this, you can install Python in Windows as well as in Linux.

Basic Python syntax

In this section, we will learn a few basic things like saving a program, printing statements and the escape sequence of a string. After installation, open the command prompt and type python. You will get the Python shell, as showcased in the following screenshot:

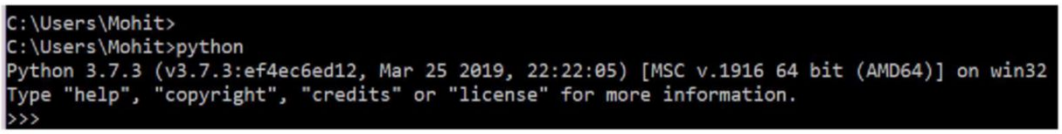

Figure 1.9

Print statement

In Python 3, we use `print` as function, as shown in the following syntax.

```
>>> print (“Hello World”)
Hello World
>>>
>>> print (‘Hello World’)
Hello World
>>>
```

So Hello World is a string, and we shall learn about string, in detail in the string chapter. For now, anything that is in single quotes or double quotes is called a **string**.

> **If you start with single-quotes, then you must end with single quotes. The same holds true for double-quotes.**

See the following example:

```
>>> print (‘Python’s world’)
  File “<stdin>”, line 1
    print (‘Python’s world’)
                   ^
SyntaxError: invalid syntax
```

In the preceding statement, the single quotes are used three times; the single middle quote is a part of the English syntax, not programming syntax. However, the interpreter takes it as a programming syntax. So, you can use double quotes here:

```
>>> print (“Python’s world”)
```

Saving the program

To write the program, you can choose any text editor that can recognize the python syntaxes. There are some lightweight editors are available such as sublime version 3 and notepad++. PyCharm is very efficient for python development.

Save the below mentioned lines in the notepad++ or in any editor:

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

Use a meaningful name with `a.py` extension to save the file.

Run the program as showcased in following screenshot:

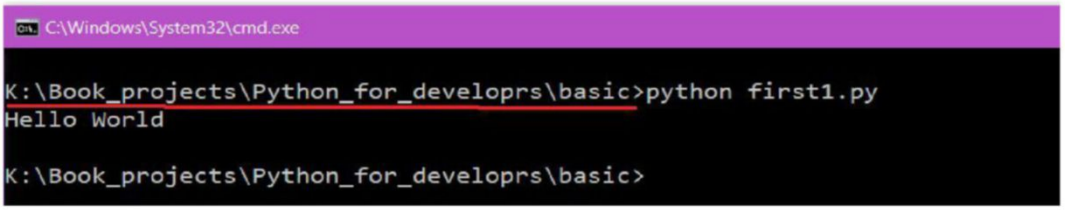

Figure 1.10

Browse the directory that contains your program, then use the command `python <program name>`.

Triple quotes

Whatever you type in triple quotes, the interpreter will print it as is.

Let us see the following example:

```
print (‘’’
,:*’’*;
*;’,%.’;
__)(__ ...
‘’’)
```

The output is showcased in the following screenshot:

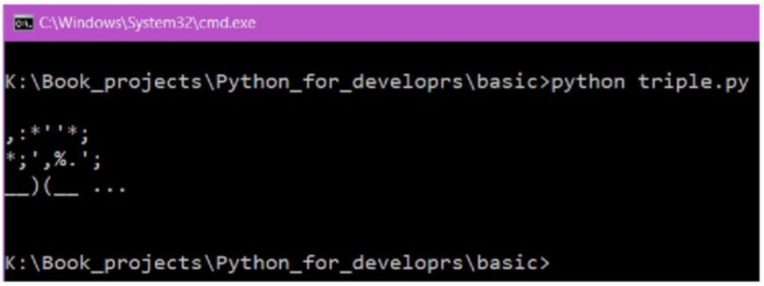

Figure 1.11

You can use double quotes or single quotes to form triple quotes. If you don't use print the statement, then the interpreter takes it as a comment.

Python Back Slash \

The Python backslash \ is used for the continuation of the string. You can stretch a single statement across multiple lines. Have a look at the following example:

```
print ("What you thin \
you become")
```

The output is showcased in the following screenshot:

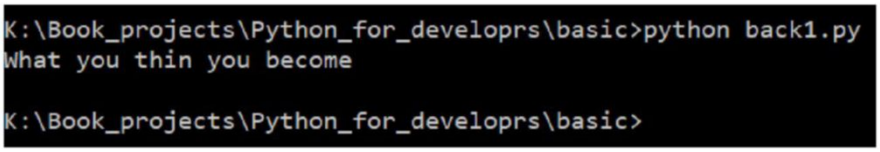

Figure 1.12

Do not use spaces after \.

Escape sequence of string

Escape Sequences allow you to put special characters such as the tab, the new line, and the backspace (delete key) into your strings. The following table showcases the escape sequence character and description.

Escape	Sequence Meaning
\b	Backspace
\a	Sound system bell
\n	Newline
\t	Horizontal tab
\\	The \ character
\'	Single quotation mark
\"	Double quotation mark

Table 1.2

Let us learn using an example:

```
print ('\a' )
print ('\t\tPython' )
print ('i know , you are \'magnificent\' ')
```

The output is showcased in the following screenshot:

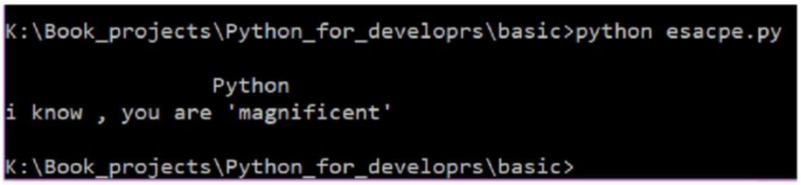

Figure 1.13

The preceding example showcases how to use the single quote as a programming syntax as well as the English language.

Python formatted output

Python allows you to set a formatted output. If you have done some coding in C language, then you must be familiar with %d, %f, %s. To represent int, float and string %d, %f and %s are used respectively.

See the following program.

```
print (“Name Marks Age” )
print ( “%s %14.2f %11d” % (“Mohit”, 80.67, 27))
print ( “%s %12.2f %11d” %(“Bhaskar” ,76.907, 27))
print ( “%s %3.2f %11d” %(“Nitin Shelke”, 56.983, 25))
```

If I use %11d, it means 11 Spaces, if I use, %12.2f, it means 12 spaces and .2 means precision. The decimal part of the number or the precision is set to 2. Let us format new output:

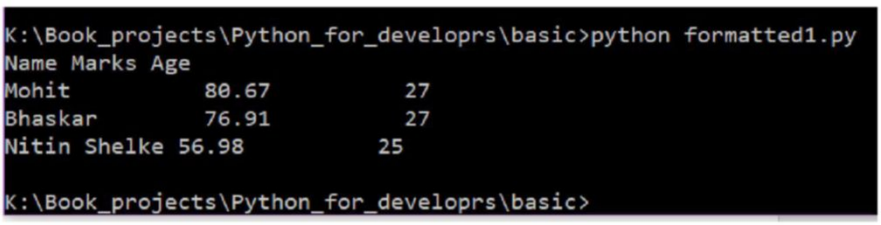

Figure 1.14

We are getting what we expected. The formatted string, sometimes, becomes very useful - like for making SQL queries, message for logger, and so on.

Conclusion

With this chapter, you have started your journey to become a developer. You have learned the Python properties, installation steps for Windows as well for Linux. The

Python shell can be used to run and test the statements. Our focus will be on Python 3. In the basic syntax section, you have learned about the print statement, saving a program, escape sequence, and formatted outputs. In the next chapter, you will learn the variables, assignment statement and different types of Python operators.

Questions

1. What is the extension to save the python program?
2. What the main versions of Python?
3. What is the use of triple quotes?

Chapter 2
Python Operators

Knowledge is power. Knowledge comes from information and information comes from the usable data. In today's modern world, everyone is dealing with data in some form or the other. In the digital world, we store a lot of data in the computer memory and perform operations. In programming, we use variables to store data. With the help of the assignment statement, the variable stores data. Python offers different types of operators to perform the operation on the stored data.

Structure

- Variables
- Assignment statement
- Operators
- Advance part

Objective

In this chapter, you will learn how to declare a variable in Python programming, what are Python operators and how to use them.

Variables

Variable means linking of the data to a name. Data is stored in the memory, and according to the data-type, the interpreter reserves the memory space. The variable

represents that memory location. With the help of a variable, we can easily access the data. We can say that a variable refers to the memory location that contains the data.

Below are the rules to define a variable:

- A keyword cannot be used as a variable. "`if`," "`def`," and "`for`", and such others are the reserved keywords. They cannot be used as variables.
- A variable can contain letters (upper case or lower case), numbers, underscore.
- Python is case sensitive, and hence, variables are also case sensitive.
- A variable cannot start with a number.
- A variable is assigned to data by using the assignment operator.

Examples for variables are as follows:

```
>>> A7 = 10
>>> 7A = 10
 File "<stdin>", line 1
  7A = 10
   ^
SyntaxError: invalid syntax
>>> _10 = 90
>>> _10
90
>>>
```

The "7A" is not a valid variable.

Assignment statement

Now we all understood the concept of variable. With the help of the assignment statement, we can bind a variable to a value or data. You can also reassign a variable to a different value. Refer to the following example:

```
<variable name>= < data>Example
inr = 10000 # An integer assignment
distanc = 10.0 # A floating point
name = "mohit: # A string
```

Multiple assignment

Python allows you to assign a single value to several variables simultaneously.

For example:

```
>>> x = y = z = 10
>>> x
10
>>> y
10
>>> z
10
>>>
```

By using one statement, x = y = z = 10, we have assigned value 10 to the variables *x*, *y*, and *z*.

Numeric data types

Python allows programmers to use various types of numbers. An integer and floating-point numbers are the two types used in python programming.

See the following figure to understand about integer numbers and floating point numbers (real numbers):

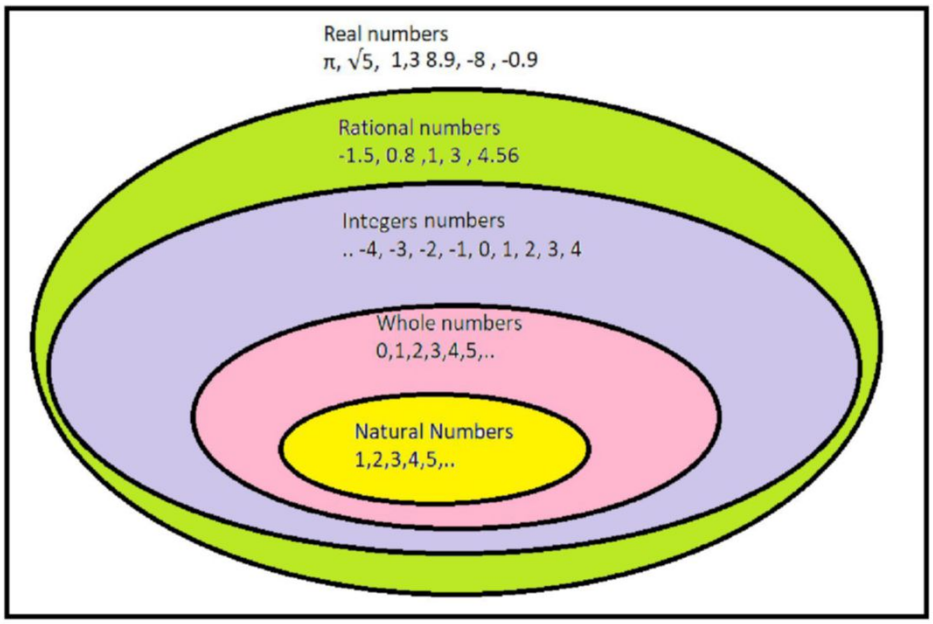

Figure 2.1

Integer numbers

From *Figure 2.1* we can conclude that the integers include 0, all the positive whole numbers, and all the whole negative numbers. The Python interpreter first examines the expression on the right side of the assignment operator and then connects the value with its variable name; it is called defining or initializing the variable.

Floating-point numbers

Python uses floating-point numbers to represent real numbers. Python offers a maximum of 17 digits of decimal precision. As we increase the number before the decimal, the precision value gets decreased. See the following examples:

```
C:\Users\Mohit>python3
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 "license" for more information.
>>> 1.1234567890123456789
1.1234567890123457
>>> 0.1234567890123456789
0.12345678901234568
>>> 05.1234567890123456789
5.123456789012345
```

Figure 2.2

It is possible to write python floating-point numbers using either scientific notation, or decimal notation. Scientific notation is often useful for mentioning a considerable amount, as showcased in the following screenshot:

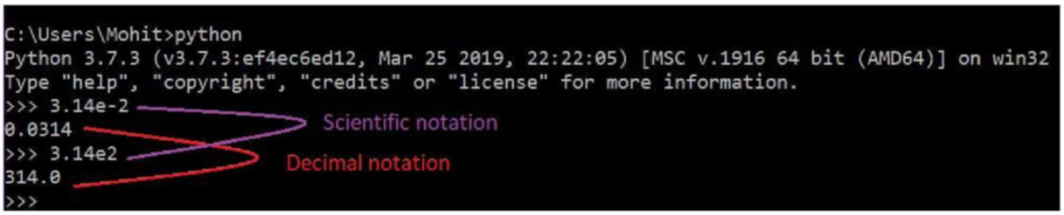

Figure 2.3

We generally use decimal notation in programming.

Python character sets

Python characters look like string, and the following figure shows the mapping of the character set:

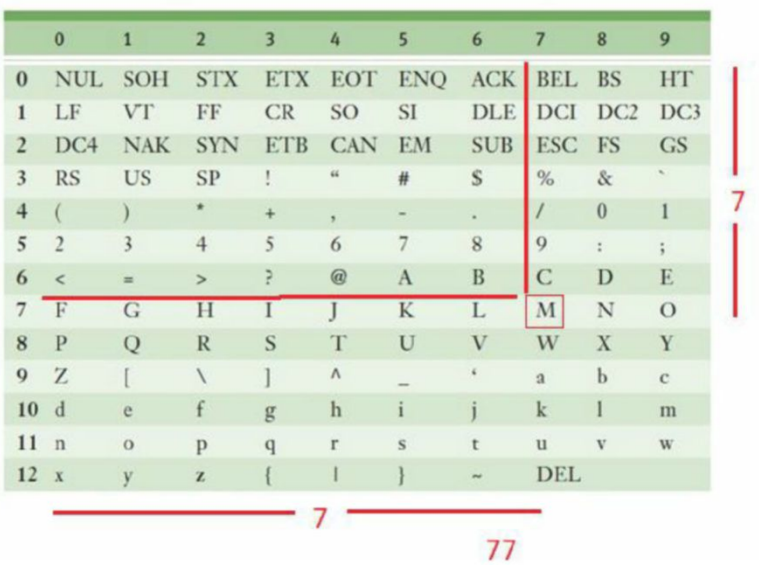

Figure 2.4

The preceding figure showcases the mapping of the first 128 ASCII codes to the character values. Every value, such as letters, special characters associated with its ASCII value and the ASCII value, is a two-digit number.

The figure contains rows and columns. The column's digit represents the first digit of the ASCII code. The row's digit denotes the second digit. The obtained ASCII value of "M" is 77.

Conversion functions

Python offers two built-in functions ord() and chr() for the interconversion of the ASCII value to the character:

- The `ord()` function converts a character to an ASCII value.
- The `chr()` function converts the ASCII value to the corresponding character as showcased in the following screenshot:

```
C:\Users\Mohit>python
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 "license" for more information.
>>> ord('L')
76
>>> chr(76)
'L'
>>> ord('M')
77
>>> chr(77)
'M'
>>>
```

Figure 2.5

In the next section, we will see the operators that Python supports.

Operators

The Python language supports the following types of operators.

- Arithmetic operators.
- Comparison operators
- Assignment operators
- Bitwise operators
- Logical operators
- Membership operators
- Identity operators

Before jumping to the details of all the operators, let us clarify what are the operator and operands in an expression, See the following expression:

```
4+ 6 = 10
```

The input `4` and `6` are the operands and + is the operator.

Arithmetic operators

Arithmetic expressions comprise of operands and operators.

The following table describes the operator's symbol with their description:

Operator	Description
**	Exponent - Performs exponential (power) calculation on operators
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
//	Floor division: Floor division means that after performing the division it returns the lower integer value as the result.

Table 2.1

The division in Python 3.x always returns a float type.

The following screenshot showcases the example of a floor division. It returns the lower integer value of the division result:

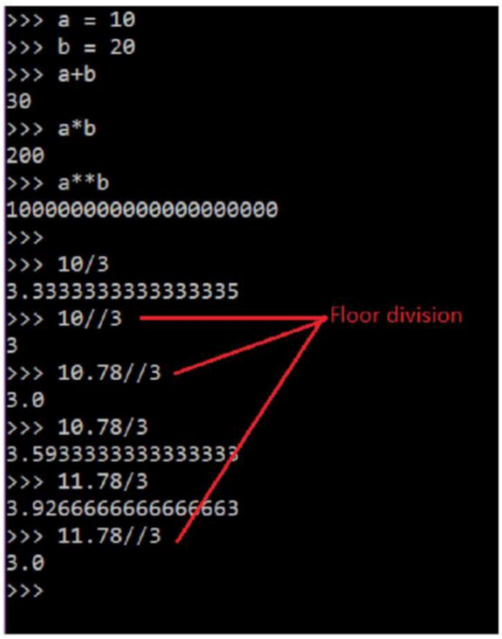

Figure 2.6

It is now clear that the floor division always returns the integer or float without the precision numbers.

Type conversions

Type conversion function facilitates the programmer to change the data type of the operand. See the following examples:

```
>>> a
10
>>> str(a)
'10'
>>> c = 12.565
>>> int(c)
12
>>> str1 = "90"
>>> int(str1)
90
>>>
```

For more clarification see the following screenshot:

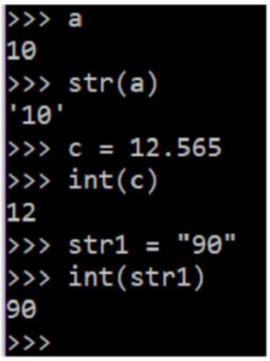

Figure 2.7

In *Figure 2.7*, 12.565 converts into 12. If you want to convert 10 into `str` use `str()`. To convert a string to an integer, use `int()`.

Comparison operators

Python supports different types of comparison operators. Comparison operators return a Boolean value, `True` or `False`:

Operator	Description
==	The operator returns `True` if the right-side operand and the left-side operand are equal.
<	The "Less than" operator returns `True` if the right-side operand is less than the left-side operand.
>	The "Greater than" operator returns `True` if the right-side operand is greater than the left-side operand.
<=	The "Less than or equal to " operator returns `True` if the right-side operand is less than or equal to the left-side operand.
>=	The "Greater than or equal to" operator returns `True` if the right-side operand is greater than or equal to the left-side operand.
!=	The "Not equal to" operator returns `True` if the right-side operand is not equal to the left-side operand.

Table 2.2

Let us consider the following example:

```
>>> A = 20
>>> B = 30
>>> A < B
```

```
True
>>>
>>> A > B
False
>>> A == B
False
>>> A != B
True
>>>
```

Numbers are compared according to their arithmetic priority. Strings are compared in alphabetical order using the numeric equivalents.

Assignment operators

We saw the declaration of assignment. Let us now see some assignment operator versions, mixed with arithmetic operators.

Operator	Description
=	a=b , b is assigned to a
+=	a+=b is equal to a=a+b
-=	a-=b is equal to a=a-b
*=	a*=b is equal to a=a*b
/=	a/=b is equal to a=a/b
**=	a**=b is equal to a=a**b

Table 2.3

Let us see some examples of assignment statements that will help you understand how to use them:

```
>>> A = 20
>>> B = 3
>>> A+=B
>>> A
23
>>> A-=B
>>> A
```

```
20
>>> B
3
>>>
```

New programmers often make mistakes while writing the *x+=y*. If you are not sure about the syntax, use a simple one, like *x= x+y*.

Bitwise operators

In Python programming, you can also perform binary bitwise operations.

The following table describes the symbols of the operators with their description.

Operator	Description
|	Bitwise OR operation
&	Bitwise AND operation
~	Binary one's Complement
^	Bitwise XOR operation
<<	Binary Left shift operator, the left-hand operand bit is shifted to the left by the number on the right
>>	Binary Right shift operator, The right-hand operand bit is shifted to the right by the number on the right

Table 2.4

The following example will showcase how to use bitwise "And" operator.

```
>>> a = 242
>>> b = 12
>>> a|b
254
>>>
```

See the following figure for better understanding:

Figure 2.8

In the preceding figure, the "OR" operation has been performed on every bit.

The truth table of OR and AND operators has been explained in the following figure:

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

OR Operation

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR Operation

Figure 2.9

Logical operators

Python offers "or", "and", and "not" as logical operators. These operators are beneficial for conditional statements:

Operator	Description
And	If both the sides, right and left, are True, then the AND operator returns True.
Or	If any side is True, then the OR operator returns True.
Not	The "not" operator inverts the outcome of the condition. If a condition in the not operator returns true, then the "not" operator makes it False, vice versa.

Table 2.5

See the following example to learn about the use of logical operators:

```
>>> 6>10 and 20>12
False
>>> 6>10 or 20>12
True
>>>
>>> 5>1
True
>>> not 5>1
False
```

```
>>> not 1>5
True
>>>
```

Let us discuss the different cases, the expression $x < y <= z$ is equal to $x<y$ and $y<=z$. In this situation, the "and" operator is used, the expression x and y is assessed only once, and $y<=z$ would be not assessed if $x < y$ is found `false`:

```
>>> 6>10>12
False
>>> 6>10
False
>>> 6>10>mohit
False
>>>
>>> 6<10>mohit
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'mohit' is not defined
>>>
```

The interpreter examines the second expression only if the first one is true.

Membership operators

To put it simply, the membership operator checks whether the given item exists in the sequence or not. Python offers “in” and “not in” operators to examine the membership of the operands in the given sequence.

The following table described the operators in detail:

Operator	Description
`in`	If the specified operand is found in the sequence, then the "`in`" operator returns `True`, otherwise, it returns `False`.
`not in`	If the specified operand is not found in the sequence, then the "`not in`" operator returns `True`, otherwise, it returns `False`.

Table 2.6

Let’s see the following example:

```
>>> str1 = "Greatwisdom"
>>> "m" in str1
True
```

```
>>> "m" not in str1
False
>>> "x" in str1
False
>>>
```

The character 'm' is the member of the string "`Greatwisdom`" and that is why it returned `True`. The character "`x`" is not a member of the `str1` and that is why it returned `False`.

Identity Operators

The identity operators determine whether the given operands point to the same memory address.

The following table describes the two identity operators, "is" and "is not", in detail.

Operator	Description
`Is`	If two variables refer to the same memory location, then the "is" operator returns `True`, otherwise, it returns `False`.
`is not`	If two variables refer to the same memory location, then the "is not" operator returns `False`, otherwise, it returns `True`.

Table 2.7

In the following example, the assigned values may be the same, but memory location is different. The operator returns True if the memory location is the same:

```
>>> X= 10
>>> Y = 10
>>> X is Y
True
>>> X is not Y
False
>>> id(X)
140715372274608
>>> id(Y)
140715372274608
>>>
>>> list1 = [9,4,1]
>>> list2 = [9,4,1]
>>> list1 is list2
```

```
False
>>>
>>> id(list1)
2472095605448
>>> id(list2)
2472096098824
>>>
```

The `id()` function returns the memory address (location) of an object. It is like memory addresses in C language.

Operator precedence

The following table shows that the highest priority is at the top and lowest priority is at the bottom. All the operators that have the same precedence, their expression is evaluated from left to right, except for comparisons and exponentiation. Comparisons can be chained arbitrarily:

Operator	Description
( )	Parentheses
A[i],A[i1:i2],fun(arg...),k.attribute	Sequence Subscription, slicing, function call, attribute reference
**	Exponentiation
-a, +a, ~a	Negative, Positive, bitwise NOT
/, *, %	Division, Multiplication, remainder
+, -	Addition and subtraction
<<, >>	Shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
is, is not , in, not in, <, <=, >, >=, !=, ==	Comparisons, membership , identity tests
not x	Boolean NOT
And	Boolean AND
Or	Boolean OR
if-else	Conditional expression
Lambda	Lambda expression

Table 2.8

Advance part

In this section, we will learn some advance things.

See the following example:

```
>>> a = 10
>>> id(a)
140704055780272
>>> b = 10
>>> id(b)
140704055780272
>>>
```

When you assign b = `10`, the interpreter checks the memory to see whether we have 10 or not. If yes, then the interpreter assigns 10 to the variable b.

Let us see some interesting things:

```
>>> b = b +10
>>> b
20
>>> id(b)
140704055780592
>>> b = b -10
>>> b
10
>>> id(b)
140704055780272
>>>
```

When we added 10 to b, the b points to 20 at a different memory address. When we subtract 10 from b, consequently, b points to 10 again at the same memory address.

Let us see the series of action again:

```
>>> a = 257
>>> id(b)
140704055780272
>>> b = 257
>>> id(b)
2506386851728
>>>
```

When the interpreter sees that the value is higher than 256, it assigns the value at a different address. Let us see the comparison operator again.

See the following example:

```
C:\Users\Mohit>python
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 "license" for more information.
>>> 10 > "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'int' and 'str'
>>>
>>> ^Z
```

Figure 2.10

In Python 3.7, you can compare different data types. Let us check the Python 2.7:

```
C:\Users\Mohit>python2
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit (
Type "help", "copyright", "credits" or "license" for more information.
>>> 10 > "a"
False
>>>
```

Figure 2.11

Though Python 2.7 is answering, logically that does not make any sense. So be careful while writing the code. Always check the datatype of the object before doing any operation.

Conclusion

In this chapter, you have learned about the assignment statement and the different types of operators of Python. In the beginning, it is difficult to remember all the operators, but as we continue practicing and solving the exercises, it will not be challenging to memorize the operators. In the beginning, you can skip the advance part. Once you complete all the core Python parts, you can learn the advance part. In the next chapter, you will learn about conditional statements, loops and control statements.

Questions

1. What is the operator to describe the power of the operands?
2. What is the difference between "is" and "="?
3. Find the answer of expression 9/10*9 in Python 3.x and Python 2.x

CHAPTER 3
Control Statement and Loop

We always keep making a lot of plans for our lives, such as if I do this, then I will become a developer. If I prepare for the exam, then I will succeed. If I learn Python, then I will be able to do projects. There are so many examples where we act based on a specific condition. Sometimes we are also prepared with a plan B, for example, if I crack the interview, then I will take up the job, else I will start a business. The same thing happens in programming; the action of logic depends on the condition. To apply the conditional statement in programming, we use the 'if, else' structure. To repeat a step multiple times, we use loops. In this chapter, we will learn about the conditional statements and loops that Python supports.

Structure

- Conditional statement
- Range
- For loop
- While loop
- Break statement
- Continue statement
- Pass statement

Objective

This chapter will help you learn about the decision-making statements. By using loops, you will learn how to perform repetitive tasks. Through break, continue and pass statement, you will get better control over the loops.

Conditional statement

We use control statements when there is an anticipation of a condition in the logic. The control statements allow a programmer to choose a specific path among all paths. *Figure 3.1* showcases a basic example of the control statement. If *a>b* is true, then it takes "YES" path, else it will take the "NO" path:

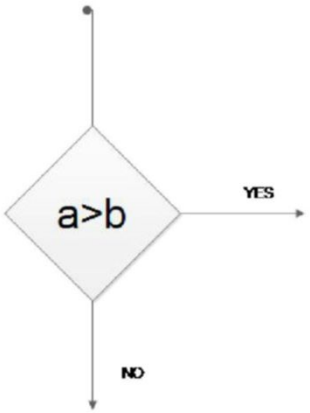

Figure 3.1

If and if-else statements

The if-else statement allows us to take the decision based on the condition, as showcased in *Figure 3.2*:

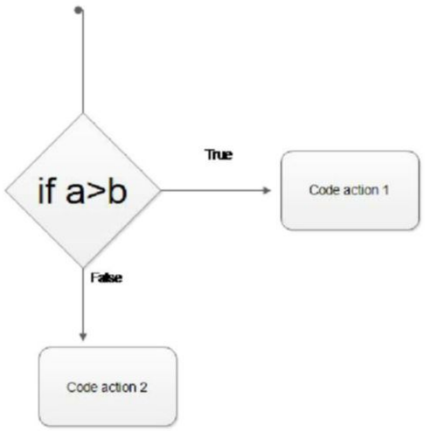

Figure 3.2

In *Figure 3.2*, based on the `True` and `False`, the algorithm chooses the path.

if statement

Through `if` statement, a program can branch to a section of code or skip it.

Let us see the example:

```
print ("Welcome Avengers" )
AV= input("Who is the strongest Avenger \t")
if AV=="Thor":
   print ("..............................")
   print ("Welcome Thor ")
```

The output is showcased in the following screenshot:

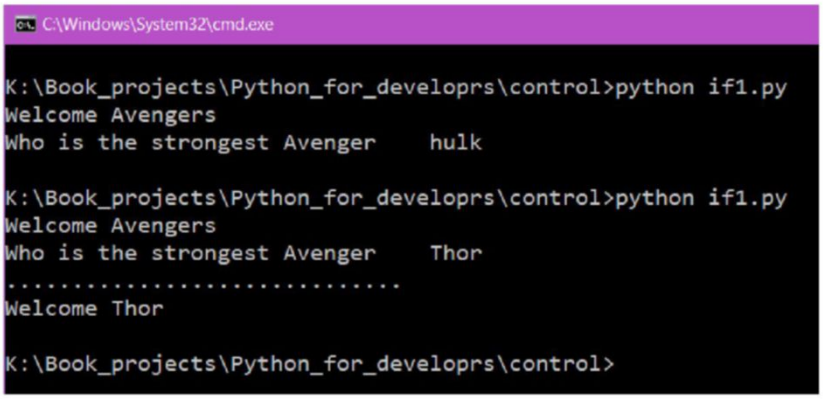

Figure 3.3

The preceding example shows branching; if you write the right password, you will enter in the 'if' statement, as showcased in *Figure 3.3*. The branching diagram is showcased in *Figure 3.4*:

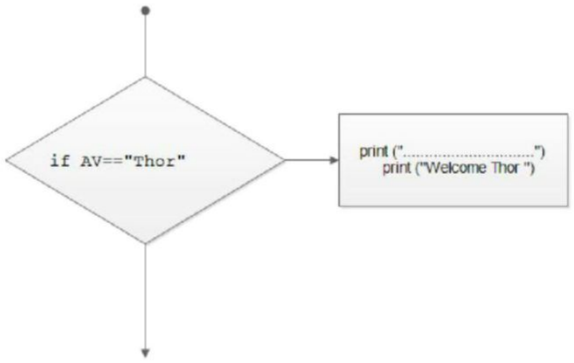

Figure 3.4

If you type the wrong password, then it would skip the 'if' statement. You may have observed that the lines after the 'if' statement, print "....." and print "Welcome Thor", are indented. When you indent the line, it becomes a block. This block is executed if the input string satisfies the 'if' condition. To make a block, ":" is used at the end of the 'if' statement. All indented lines that are written after ":", make a block, as showcased in the *Figure 3.5*:

Figure 3.5

Tabs or Spaces?

This depends on personal style. You may apply whatever you want to but be consistent. If you are going to use a two spaces indentation, then abide with this throughout the program. Don't mix spaces and tabs. Although you can use both, but this can lead to a a lot of issues later.

if-else statement

It is also called a two-way choice, as it allows the program to choose between two alternative courses of action.

Following is the syntax of the if-else statement.

```
if condition :
  sequence of statements-1
else:
  sequence of statements-2
```

Let us understand through the following example:

```
print ("Welcome Avengers" )
AV= input("Who is the strongest Avenger \t")
if AV=="Thor":
   print ("..............................")
   print ("Welcome Thor ")
```

```
else :
   print ("Access Denied")
```

The output is showcased in the following screenshot:

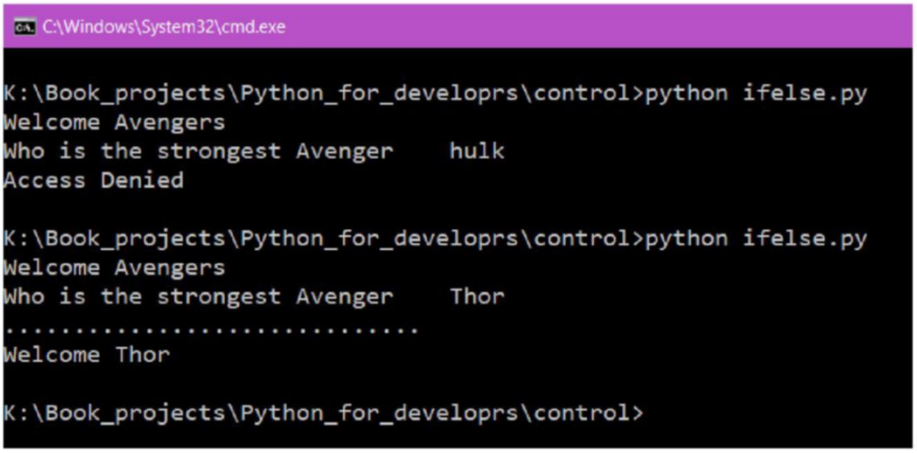

Figure 3.6

If the user provides "Thor", then the "if" block gets executed. If a user offers a value other than "Thor", then the "else" block gets executed and a message "Access Denied" is printed.

if-elif-else structure

Occasionally, there are several test cases in a program that comprise of more than two alternative courses of action. In multi-way if statements the program examines each condition until one decides to `true` or all decide to `false`. When a condition evaluates to `True`, the corresponding action of the condition takes place. If no condition satisfies or computes to `true`, then the respective else trailing action will be conducted.

The multiway `if` declaration syntax is showcased below:

```
if condition-1:
  sequence of statements-1
elif condition-n:
  sequence of statements-n
else:
  default sequence of statements
```

Let's consider a grading system as an example. The following table shows grade A, grade B, grade C, grade D ranging from 1 to 100:

Letter	Marks Range
A	All grades above 89
B	All grades above 79 and below 90
C	All grades above 69 and below 80
D	All grades below 70

Table 3.1

See the implementation of this in the following program:

```
num =int(input("Enter the number: "))
if num > 89:
    letter = 'A'
elif num > 79:
    letter = 'B'
elif num > 69:
    letter = 'C'
else :
    letter = 'D'
print ("The Grade is " , letter)
```

The output is showcased in the following screenshot:

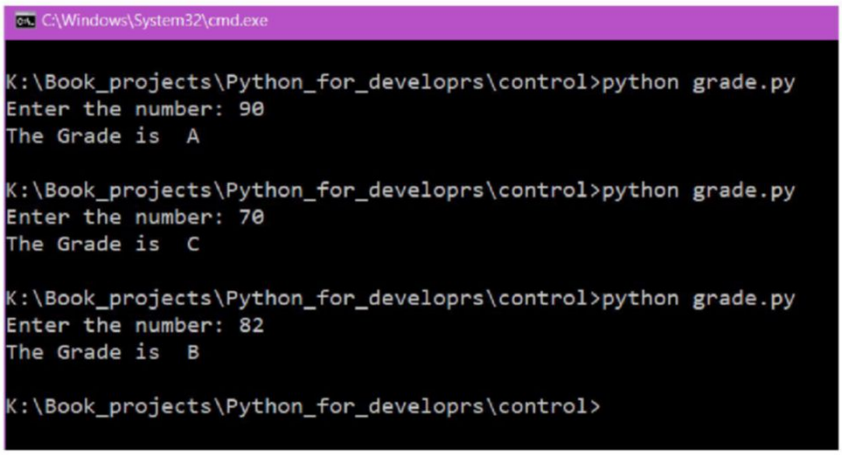

Figure 3.7

The following figure showcases the sequence of actions. The number 90 evaluated `True` in the first `if` condition. Similarly, 82 and 70 evaluated `True` at their appropriate condition:

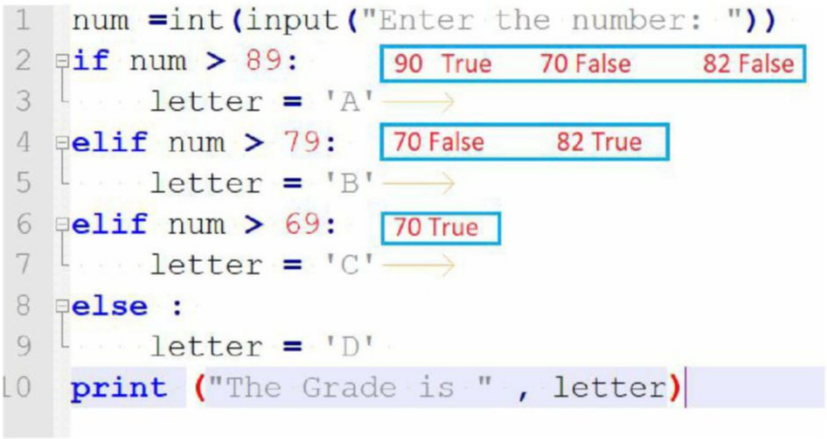

Figure 3.8

Range()

The `range()` function populates its range list whenever it is employing such as for loop.

See the following syntax of `range()`:

```
range(start-value, end-value, difference between the values)
```

Let us understand it using examples:

```
C:\Users\Mohit>python3
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> range(1,30,1)
range(1, 30)
>>>
```

We can use the list function to receive the values from the range:

```
>>> list(range(1,30,1))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
```

```
>>>
>>> list(range(1,30,3))
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28]
>>>
>>> list(range(0,30,3))
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>>
>>> list(range(0,30))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
>>>
>>> list(range(30))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
>>>
>>> list(range(-1,30))
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
>>> list(range(-1,30,-2))
[]
>>>
```

The `range()` function does not consume the memory.

Let us see the examples of it:

```
C:\Users\Mohit>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)
Type "help", "copyright", "credits" or "license" for more information.
>>> r1 = range(1,10)
>>> r1
range(1, 10)
>>> r2 = range(1,300)
>>> r2
range(1, 300)
>>> import sys
>>> sys.getsizeof(r1)
48
>>> sys.getsizeof(r2)
48
>>>
```

Figure 3.9

The `sys.getsizeof()` shows the memory utilized by the passing argument. The above example concludes that `r1` and `r2` contain different arguments, but the `sizr` of the memory used is the same. If you are using Python 2.7, then always remember the following diagram:

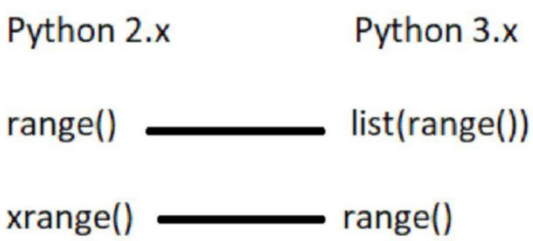

Figure 3.10

The `range()` and `xrange()` of python 2 are similar to `list(range())` and `range()` of Python 3.

Loops

Sometimes you may experience a scenario where you need to execute a block of code multiple times. Programming languages provide control statements with repetition statements, called loops, which repeat an action.

There are two types of loops:

Definite loop: In this type, the action is repeated a predefined number of times.

- **Indefinite loop:** In this type, the action is repeated until the program determines that it needs to stop.

For loop

In this section, we will learn about the `for` loop. Suppose that you want to repeat some course of action a definite number of times; in that case, we use for loop.

Please see below the syntax of `for` loop:

```
for <variable> in range(<an integer expression >):
  <statement-1 >
  <statement-n >
```

The above syntax is a fundamental type of syntax. The syntax may vary when we use list, tuple, dictionary string, and so on. The first line of the code in the `for` loop is called the loop header. An integer expression specifies the number of repetitions of the loop. The colon (:) ends the loop header. After the header for loop, the body

consists of the statements in the remaining lines of code. Always remember, the for loop body must be indented. The statements in the `for` loop body are executed sequentially on each pass through the loop. Let us understand the working of the for loop using examples.

Printing number 0 to 9

```
for each in range(10):
  print (each)
```

In the preceding example, each is a variable.

The output is showcased in the following screenshot:

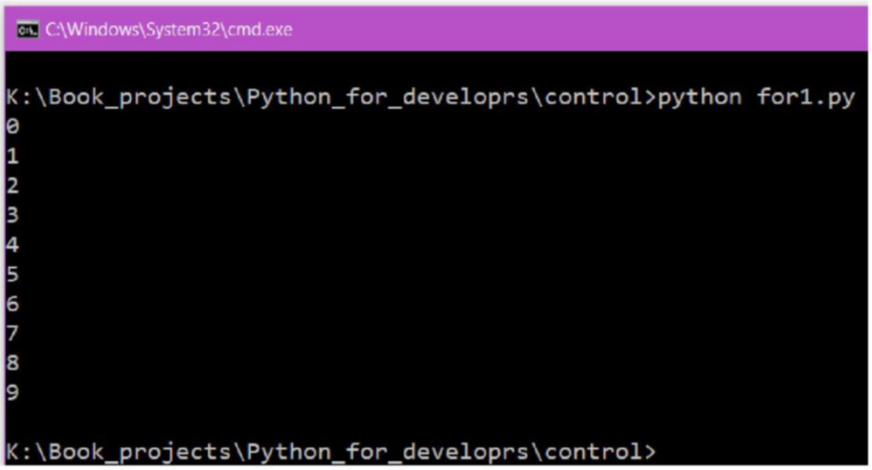

Figure 3.11

For each iteration, the `for` loop demands the next number from the `range()` function, the `range()` function calculates the next value and supplies it to the "`each`" variable

Printing the numbers in horizontal fashion

See the following code:

```
for each in range(10):
  print (each, end = " ") # python 3.7
  # print each,    # python 2.7
```

In the preceding code, the "#" makes a line as a comment. In Python 3.x, we need to use syntax end=" " in the `print()` function. In Python 2.x, we have to use "," at the end of the syntax.

Output:

```
K:\Book_projects\Python_for_developrs\control>python for2.py
0 1 2 3 4 5 6 7 8 9
K:\Book_projects\Python_for_developrs\control>
```

Figure 3.12

Let us see the next example. We are printing a statement multiple times:

```
for each in range(10):
  print ("Hello") # python 3.7
print ("Done")
```

The output is shown in the following screenshot:

```
K:\Book_projects\Python_for_developrs\control>python for3.py
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Done

K:\Book_projects\Python_for_developrs\control>
```

Figure 3.13

In the above example, the statement `print` ("`Hello`") comes under the `for` loop, but the statement `print` ("`Done`") is not indented, hence the interpreter executed it only once.

For loop with string

Let us see the use of the `for` loop with string:

```
str1 = "Python programming"
for each in str1:
  print (each)
```

The output is showcased in the following screenshot:

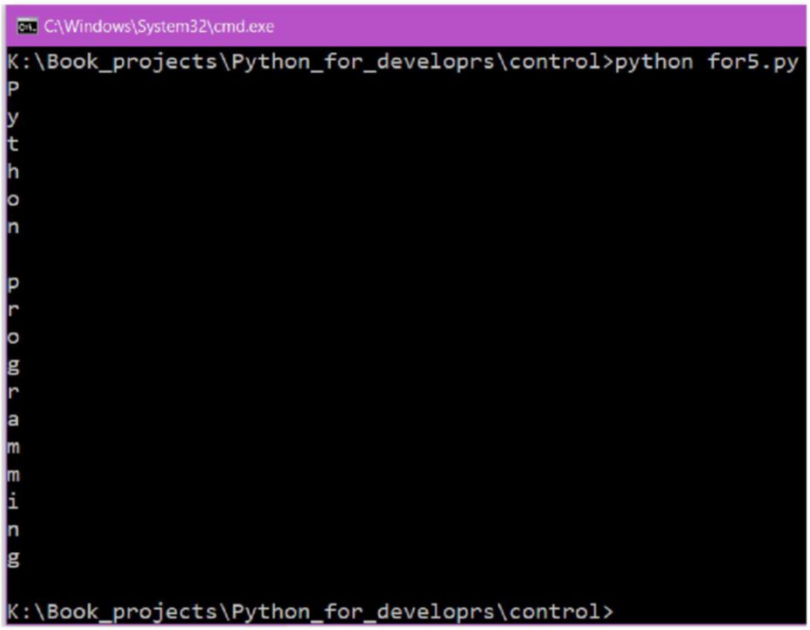

Figure 3.14

String supplies each character to the "`each`" variable for every iteration. We will see the use of the "`for`" loop with list, tuple, and dictionary, in the corresponding chapters.

Let us work on some exercises.

Exercise 1

Obtain the sum of first 10 numbers.

See the following code:

```
sum = 0
for each in range(1,11):
  sum = sum+each
print ("total number is ", sum)
```

The output is showcased in the following screenshot:

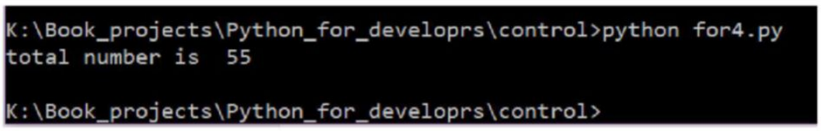

Figure 3.15

The Sum of the numbers 1 to 10 is 55. The variable sum is initialized to 0. We use `range(1,11)` because the last value in the `range()` is not included, which means that if we use 11, then it goes up to 10.

Exercise 2

Calculate the frequency of the letter in the given string:

```
str1 = "Python programming by mohit"
i = 0
arg1 = input("Enter the char to be checked ")
for each in str1:
  if each==arg1:
    i = i+1
print ("Occurrence of %s is %d"%(arg1, i))
```

The output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\control>python for6.py
Enter the char to be checked o
Occurrence of o is 3

K:\Book_projects\Python_for_developrs\control>python for6.py
Enter the char to be checked m
Occurrence of m is 3

K:\Book_projects\Python_for_developrs\control>python for6.py
Enter the char to be checked p
Occurrence of p is 2

K:\Book_projects\Python_for_developrs\control>
```

Figure 3.16

The program is straightforward. It gets each character from the given string and matches it against the entered character. If it matches, then increment the variable i with one.

while loop

Sometimes it is uncertain to determine, in how many iterations can a program complete its task. The loop ultimately ends its job, but only when a condition changes. In some situations, you use an infinite loop and inside the loop, some condition determines when to terminate the loop. This process is called conditional iteration.

Please see below the syntax of the `while` loop:

```
while <condition>:
  <sequence of statements>
```

There needs to be at least one statement in the loop body, which updates the variable that affects the condition that leads to the termination of the `while` loop. "Sentry" variable or "loop control" variable control the while loop. Let us discuss the first example:

```
name = input("Enter your name ")
while name!="point break":
  print ("Wrong password")
  name = input("Enter the password again ")
print ("Welcome Thor")
```

The output is showcased in the following screenshot:

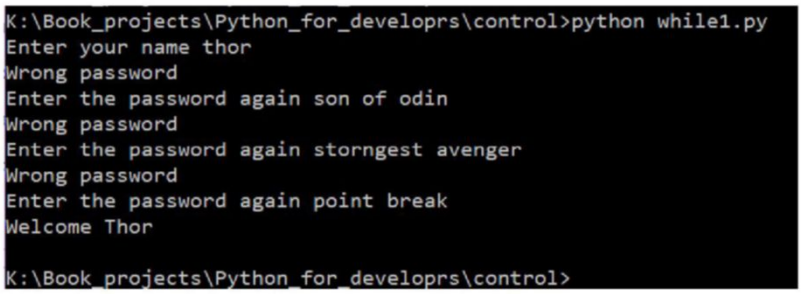

Figure 3.17

If you have seen the movie "Thor Ragnarok", then you might remember the "Thor" gives a voice password to quinjet's computer. We have used the same thing here with the help of the "while" loop. But here we used a text password, and not a voice password. For the `while` loop, we have used a condition - if the input variable `name`, is not equal to point break, the condition becomes `True`. Here `name` is called the control variable, asking for the password. When you enter the wrong password, the program continues to ask for the right password until you type the correct number. So, this is the power of the "while" loop - unless you enter right number program, it will never end.

Let us calculate the of the first 10 numbers:

```
sum1 = 0
n = int(input("Enter the number "))
i=1
sum2 = 0
while i<=n :
```

```
  sum2 = sum2+i
  print (sum2)
  i = i+1
print ("Total sum is ", sum2)
```

The output is showcased in the following screenshot:

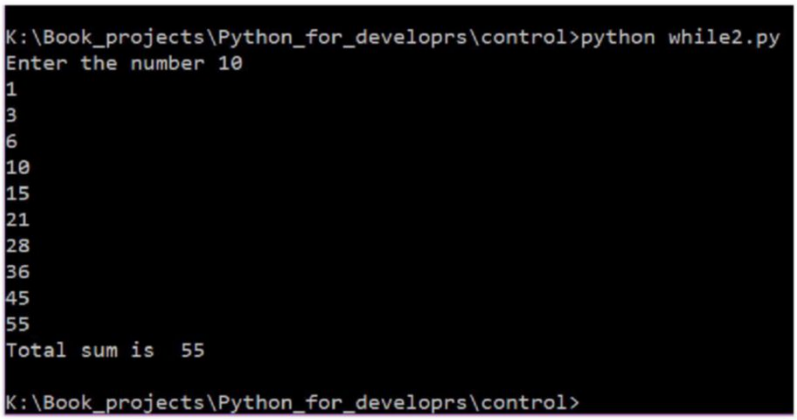

Figure 3.18

Both, the `for` and `while` codes produce the same result. But the `while` loop code segment is noticeably more complex, containing extra statements. The i is the loop control variable. It must be explicitly initialized before the loop header. Although, the use of the Python while loop results in more labour for the programmer, you will soon see the problems for which the Python "while" loop is the only solution.

Break statement

Sometimes, depending on the situation, we want to terminate the loop in the middle of the iteration. The break statement allows us to break the loop at any point of the iteration.

Exercise:

Find the sum of the first n number, but if the sum is greater than 100, then stop the iteration.

Code:

```
sum = 0
num = int(input("Enter the number "))
for each in range(1,num):
  sum = sum+each
```

```
  if sum > 100:
    break
print ("total number is ", sum)
```

In the preceding code, it is clear that if the sum becomes greater than 100, then the interpreter executes the break statement and the break statement terminates the loop.

The output is showcased in the following screenshot:

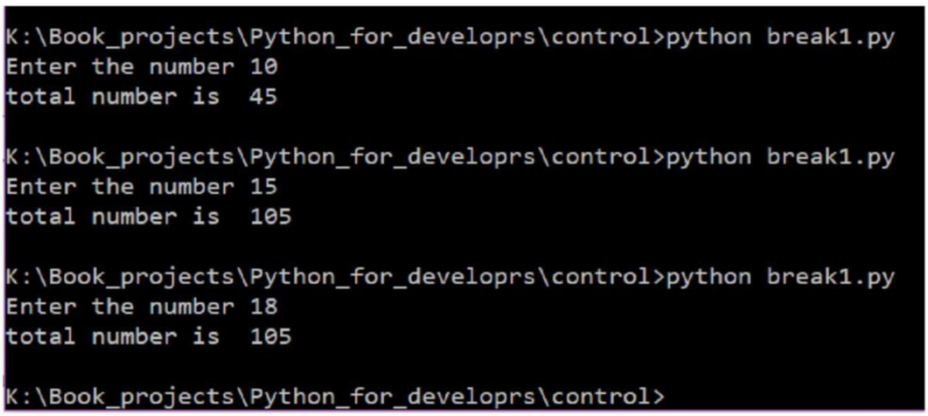

Figure 3.19

The output shows that the maximum value of the sum is 105.

Break statement with the while loop

Let us use the "true while" loop and find the sum of the number given at the run time.

Please see below the code:

```
sum1 = 0
while True:
  n = int(input("Enter the number to add :or press 0 to exit "))
  sum1 = sum1 + n
  print ("sum1: ", sum1)
  if n ==0:
    break
print ("Total is ", sum1)
```

In the preceding code, we have used the `while True` statement. Thus implies that there is no condition check, we have explicitly fixed the `True`. Consequently, the

interpreter has to enter the `while` loop. If the entered number is not equal to 0, then the number is added to the variable sum1. But, if the entered number is 0, then interpreter executes the break statement, which breaks the loop immediately.

The output is showcased in the following screenshot:

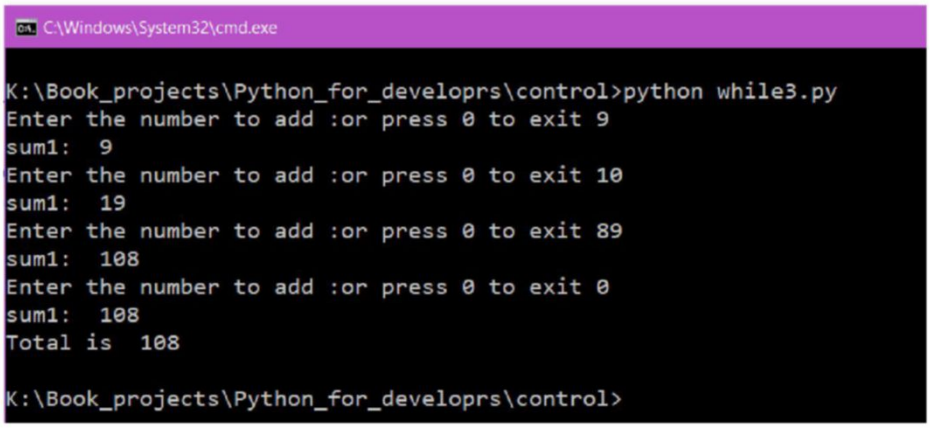

Figure 3.20

In the case of nested loops (one loop contains another loop), the break statement only terminates the loop which contained it.

Continue statement

The "continue" statement is used to skip the current iteration. With the `continue` statement, the loop does not terminate, but continues with the next iteration.

Consider a list containing some integers, and we want to perform some calculations on those numbers.

Please see the below code:

```
list1 = [1,2,3,0,4,5,0]


for i in list1:
  c = 10/i*100
  print ("Percentage is ", c)
```

The output is showcased in the following screenshot:

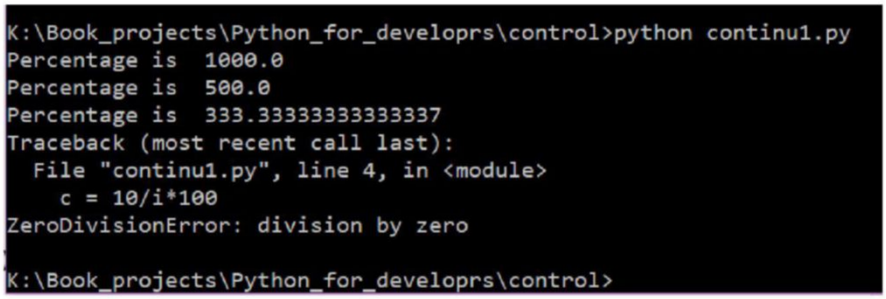

Figure 3.21

In the preceding output, we are getting an error due to the zero division error. We can avoid these errors with the help of the `continue` statement.

See the following modified code:

```
list1 = [1,2,3,0,4,5,0]
for i in list1:
  if i == 0:
    continue
  c = 10/i*100
  print ("Percentage is ", c)
```

The output is showcased in the following screenshot:

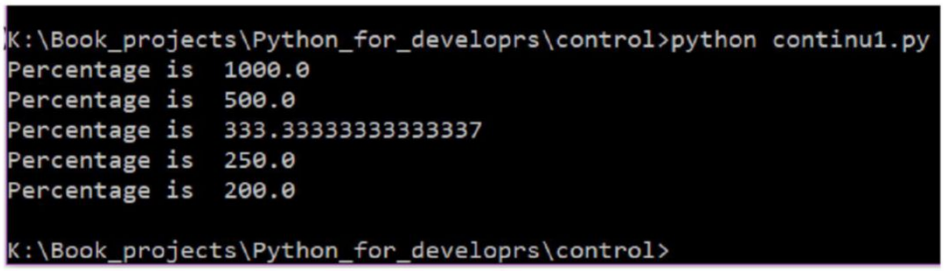

Figure 3.22

The continue statement is used to skip that particular iteration.

else statement

The "else" in the loop is considered as a no break statement. If we use the "else" block after the loop, then the interpreter executes the else block only if it encounters no a break statement. See the following example:

```
for each in [1,2,3,4]:
  print (each)
  if each == 5:
    break
else :
  print ("All done")
```

See the output in the following figure

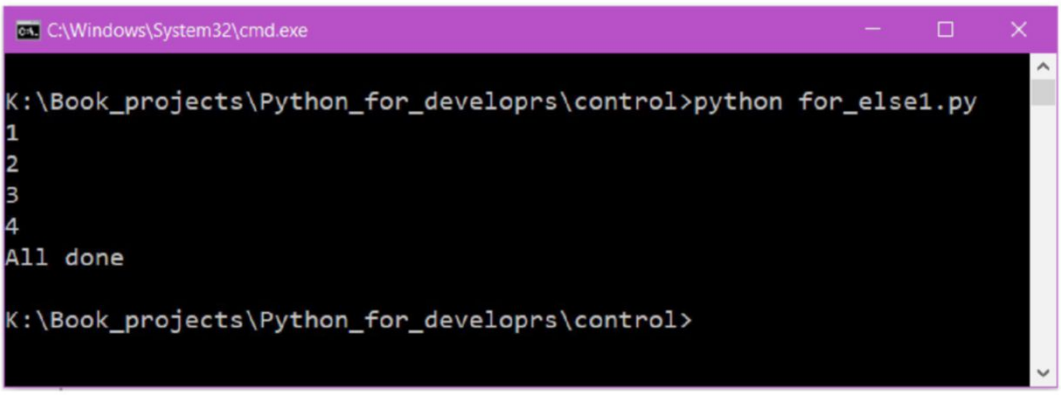

Figure 3.23

You can see that since the interpreter has encountered the no break statement, the else block has been executed. If we change the statement "`each==3`", then the else block would not be executed. See the following output:

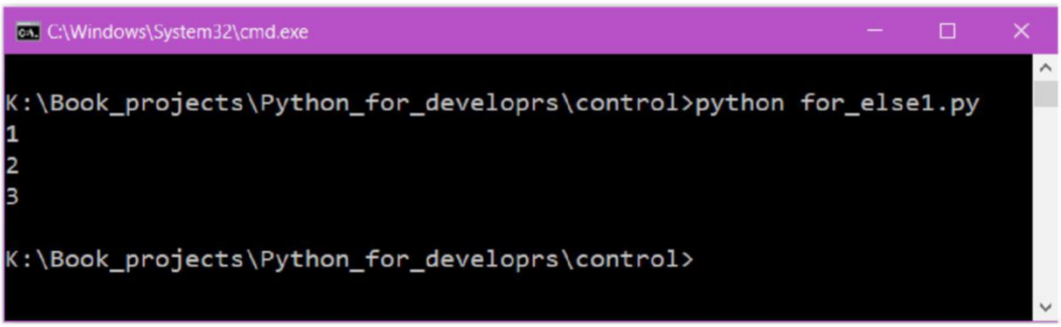

Figure 3.24

pass statement

Consider a scenario where you want to reserve a loop for future use. You don't want to write anything in the loop, but the loop cannot have an empty body. So, we use the `pass` statement to construct a body that does nothing.

Please see the following syntax of `pass`:

Example 1:

```
def load_balancing():
  pass
Example 2
for each in [12,3]:
  pass
```

In the above examples, the pass statement acts as a body and does nothing.

Conclusion

In this chapter, you have learned conditional statements that help in taking decisions. In Python, for taking decisions, we use the `if-else` and `elif` statement. The loop facilitates the program to repeat an action multiple times. The `for` loop has been used as a definite loop, and the while loop is often used as an indefinite loop. The `for` and `while` loop have their own significance. The break and `continue` statements give more over the program. In the next chapter we will learn about the string data type. in that we shall learn the properties of string, methods and functions of string.

Questions

1. What is the difference between a “continue” and a “break” statement?
2. What is the return type of the range()?
3. From the following statement, which one will consume more memory: range(10) or range(20)?

Chapter 4
Strings

The string plays a vital role in programming. In Python, string is a data type. With the help of a string, you can store a character, a word, a sentence, or a complete text. Python offers several methods and functions for string operations. In programming as well as in day-to-day life, we use strings daily; for example, the name of an employee, the name of a newspaper, the address of any place, and so on. There are numerous examples of strings.

Structure

- String
- Indexing using the Subscript operator
- Slicing
- String methods
- String functions

Objective

In this chapter, we will learn about the string, its properties, indexing, slicing method, and the functions that can be applied on the string.

String

A string is a sequence of zero or more characters. Let us see the example of a string:

```
>>> a = ""
>>> type(a)
<class 'str'>
>>> a = "abc"
>>> type(a)
<class 'str'>
>>> a = "123"
>>> type(a)
<class 'str'>
>>>
```

From the preceding example, we can conclude that whatever is inside the quotes, becomes a string. The string is an immutable data structure. Although we can access the internal data elements, we cannot modify the string.

Python sequences fall into one of the two categories - mutable or immutable.

Mutable means changeable. The mutable sequence is the one that can be changed. Immutable means unchangeable. Python strings are immutable sequences, which means that they can't change. So, for example, the string "`INDIA`" will always be the string "`INDIA`".

Let us see the immutability in the following examples:

```
>>> str1 = "INDIA"
>>> type(str1)
<class 'str'>
>>>
>>> id(str1)
1957360731448
```

We can reassign the variable str1, but cannot change the value at the address 1957360731448:

```
>>> import ctypes
>>> ctypes.cast(1957360731448, ctypes.py_object).value
'INDIA'
>>>
```

```
>>> str1 = "Game"
>>> id(str1)
1957360734024
>>> ctypes.cast(1957360731448, ctypes.py_object).value
<cparam 'P' (000001C7BBCA8538)>
>>> ctypes.cast(1957360731448, ctypes.py_object).value
```

After a few seconds, the `ctypes.cast(1957360731448, ctypes.py_object).value` would show the error, because the data is not present at the address `1957360731448`.

In order to find the length of Python string, function `len(string)` is used. See the following example:

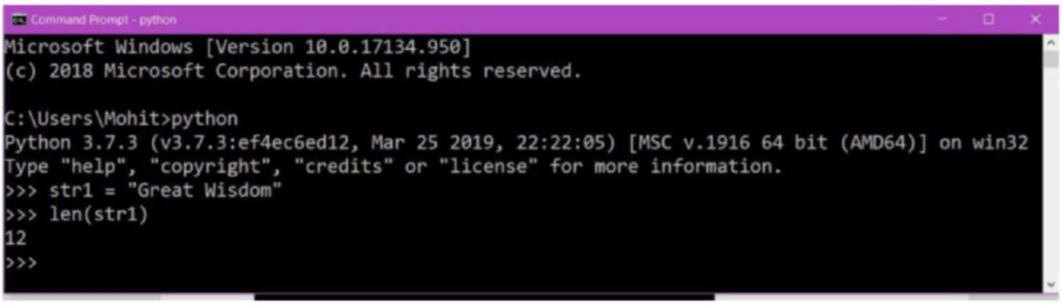

Figure 4.1

Indexing using subscript operator

Sometimes a user needs to examine one string character at a specified place without analysing them all. This is possible by using the subscript operator. A subscript operator's syntax and examples are provided below:

```
<given string>[<index>]
```

The `given` string indicates that you want to examine, the `index` is an integer that indicates the position of a particular character, as showcased in the following example.

In the following cases, the subscript operator is used to access the characters in the string "`Great Wisdom`":

```
>>> str1 = "Great Wisdom"
>>> len(str1)
12
>>> str1[0]
'G'
>>> str1[1]
'r'
>>> str1[11]
'm'
>>> str1[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> str1[-1]
'm'
>>> str1[-12]
'G'
>>>
```

Figure 4.2

Now we have understood the subscript operator. The string, "`Great Wisdom`", is `12` characters long. The syntax `str1[0]` represents the character `'G', str1[-1]` represents the last character of the string, and `str1[-12]` represents the first character of the string. The `str1[12]` generates an error, "out of range" as can be seen in the following figure:

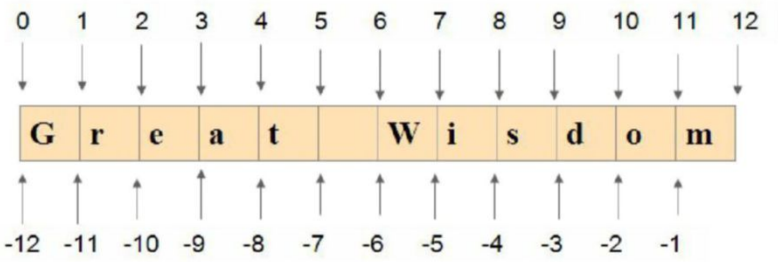

Figure 4.3

Slicing for substrings

The extracted portions of the string are called substrings. A lot of times, you need some part of the string, such as the first two characters of the string. The subscript operator uses a process called slicing. In slicing, the colon (:) is used. An integer value will appear on either side of the colon.

Please see below the syntax:

```
Given string[start : End: Step]
```

- `Given string:` The string that you want to examine.
- `Start:` The starting index of the string.
- `End:` The last index of the string.
- `Step:` the difference between each character.

See the following example:

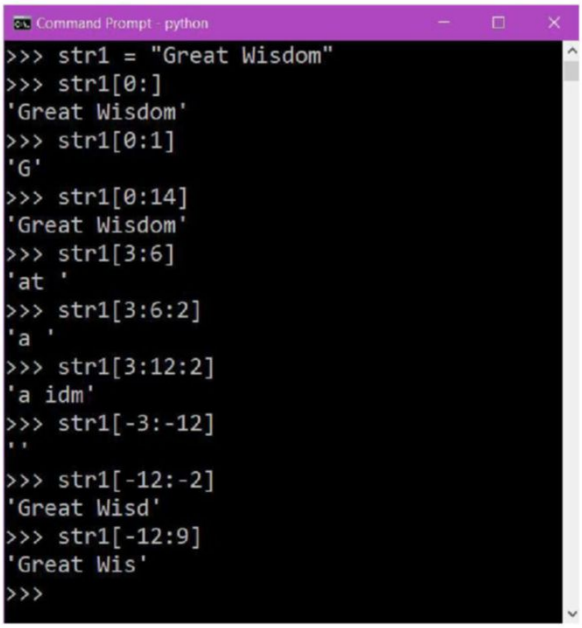

Figure 4.4

You can use the negative indexing, mixing of a positive and negative index. In slicing, the interpreter does not give any error if the index is greater than the length of a string.

For more clarification, see the following figure:

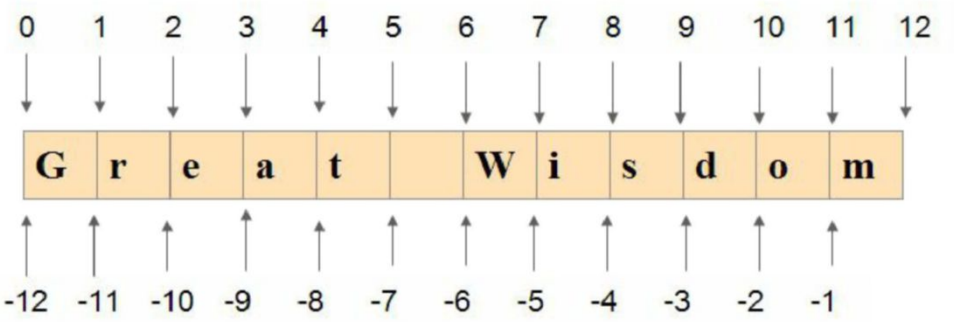

Figure 4.5

If we use a negative step, then string will be traversed from the right to the left. See the following examples.

```
>>> str1 = "Great Wisdom"
>>> str1[0:10]
'Great Wisd'
>>> str1[0:10:-1]
''
>>> str1[10:0:-1]
'odsiW taer'
>>> str1[:0:-1]
'modsiW taer'
>>> str1[::-1]
'modsiW taerG'
>>>
```

In the preceding examples, you can see that if we use a negative step, then the "start" must be greater than the "end". The syntax **`str1[::-1]`** prints the reverse of the string

String methods

Now we will learn about the Python string methods. In order to see all the string methods, use `dir(str object)` as showcased in *Figure 4.6:*

Figure 4.6

There are so many methods that are associated with the string; we will study most of them.

count()

Please see the following syntax:

```
str.count(substr , start , end)
```

The method `str.count(substr , start , end)` returns the number of occurrences of the string `substr` in the string `str`. By using the parameters `start` and `end`, you can give a slice of the string `str`:

```
>>> str1 = "python programming"
>>> str1.count('p')
2
>>> str1.count('p',0,6)
1
>>>
```

find()

Please see the following syntax:

```
str.find(given_str, beg=0 end=len(string))
```

The `find()` method is used to find out the index of the `given_str` in string `str`. The `find` method only finds the index of the first occurrence of `given_str` from the left side.

For example:

```
>>> str1 = "The mind is everything. What you think you become"
>>> str1.find("you")
29
```

If you want to find from the right side, then use the method `rfind`:

```
>>> str1.rfind("you")
39
>>>
```

Justify methods

Python offers four methods for string justification. Let us see the methods one-by-one.

ljust()

Syntax:

```
str.ljust(given_length, fillchar)
```

The `ljust` method is used to justify the string `str` from the left side. The total length of the string is defined in the first parameter of method `given_length`. The `fillchar` character is used to fill the remaining space in the string. (default is space). The remaining space only exists if the `given_length` is greater than the length of the string `str`:

```
>>> "intel".ljust(10, "$")
'intel$$$$$'
>>>
>>> "intel".ljust(10)
'intel     '
>>>
>>> "intel".ljust(5)
'intel'
>>>
>>> "intel".ljust(10,"#$")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: The fill character must be exactly one character long
>>>
>>> "intel".ljust(10,"")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: The fill character must be exactly one character long
>>>
```

From the preceding example, it is clear that you cannot use the null string as `fillchar`.

rjust()

Please see the below syntax:

```
str.rjust(width[, fillchar])
```

The `rjust` is the brother of ljust, used to justify from the right side.

```
>>> "python".rjust(10,"#")
'####python'
>>>
```

center()

Please see the following syntax:

```
str.center(width[, fillchar])
```

The method `center()` makes the `str` centred, by taking the width parameter into account. Padding is specified by the `fillchar` parameter. The default filler is a space.

```
>>> "python".center(10,"#")
'##python##'
>>>
```

zfill()

Please see the following syntax:

```
str.zfill(width)
```

This method is the special case of `rjust`. This method pads the string on the left with zeros to fill the width:

```
>>> "01".zfill(10)
'0000000001'
>>> "03132".zfill(10)
'0000003132'
>>>
```

The `zfill` method can be used to make a binary, fixed-length number.

Let us see one exercise of the `ljust` method.

In the following code, `ljust` is used to print the statement elegantly:

```
name = ["Mohit", "Ravender", "Bhaskar Narayan Das"]
print ("Name\t\t  ", "Age","marks")
for each in name:
  print (each.ljust(19), 30, 89.56)
```

The output is showcased in the following screenshot:

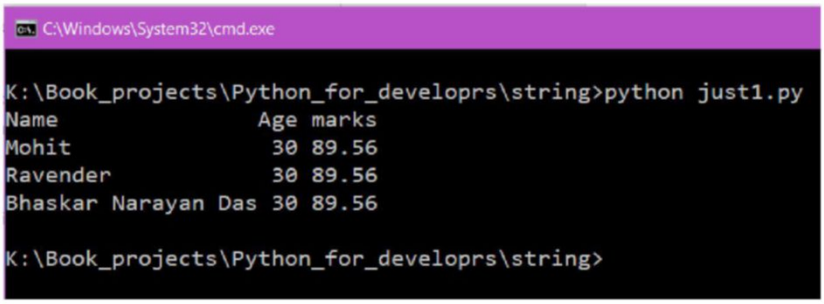

Figure 4.7

Case methods

We will now discuss the method that deals with the cases.

lower()

Please see the below syntax:

```
str.lower()
```

The method `lower()` returns the string in lower case. The following example showcases the rest:

```
>>> "ABCabc".lower()
'abcabc'
>>> "ABC112&*".lower()
'abc112&*'
>>>
```

upper()

Please see the below syntax:

```
str.upper()
```

The method `upper()` returns a copy of the string, in which all the case-based characters have been converted to upper case:

```
>>> "abc".upper()
'ABC'
>>> "abc123".upper()
'ABC123'
>>>
```

capitalize()

This function capitalizes the first letter of the string:

```
>>> "what we think we become".capitalize()
'What we think we become'
>>>
```

title()

Please see the below syntax:

```
str.title()
```

The `method title()` returns a copy of the string, in which the first characters of all the words of the string are capitalized:

```
>>> "what we think we become".title()
'What We Think We Become'
>>>
```

swapcase()

Please see the below syntax:

```
str.swapcase()
```

The method `swapcase()` returns a copy of the string, in which all the cased based characters swap their cases:

```
>>> "abc123ABC".swapcase()
'ABC123abc'
>>>
```

Strip methods

Sometimes, we want to remove some unwanted character or sub-string from the given string. To remove it, we have three methods. Let's see them one-by-one:

Please see the below syntax:

```
Str1.rstrip("characters to be removed")
```

The `lstrip` method removes the characters from the right-hand side.

See the following example:

```
>>> str1 = "hello"
>>> str1.lstrip("h")
```

```
'ello'
>>> str1.lstrip("he")
'llo'
>>> str1.lstrip("eh")
'llo'
```

You can see that whether we use "eh", or "he", the result is the same.

```
>>> str1.lstrip("ehl")
'o'
>>>
```

From the preceding example, we can easily conclude that it checks all the characters one-by-one:

```
>>> str1.lstrip("ek")
'hello'
>>> str1.lstrip("hk")
'ello'
>>> str1.lstrip("kh")
'ello'
>>>
```

Now we can easily say that the interpreter is checking the "`h`" character first. If you want to remove something from the right-hand side, you can use `rstrip()` method. Similarly, if you want to remove the characters from both the sides in one go, you can use the `strip` method.

If you want to remove or replace characters from the string, you can use the `replace` method.

replace()

Please see the below syntax:

```
str.replace(old, new,max)
```

The `replace()` method returns the string, in which the occurrences of the string specified by the '`old`' parameter have been replaced with the string specified by the '`new`' parameter new. The '`max`' parameter defines how many occurrences have been replaced. If `max` is not specified, then all occurrences will be replaced.

For example:

```
>>> str1 = "time is great and time is money"
>>> str1.replace("is", "was")
'time was great and time was money'
>>> str1.replace("is", "was",1)
'time was great and time is money'
>>>
```

Split methods

Sometimes we want just a specific part of the string. Although we can use slicing for that, sometimes slicing does not fit the requirement. See the following example:

```
>>> date = '2018-12-31'
```

Let's say, we want the day of the month:

```
>>> date[8:]
'31'
```

We are getting the desired result, but for the following example, the above expression will not work:

```
>>> date = '2018-2-31'
>>> date[8:]
'1'
>>>
```

So, we will use the `split` method to achieve the same thing.

Please see below the syntax of `split`:

```
Str1.split(delimiter, max)
```

The `split` method is used to split the string, based on the delimiter. The `split` method returns a list of split sub-strings. The `max` argument is an integer, which signifies how many splits are to be performed. Let us see a different example for the split.

Let us split based on "-":

```
>>> date1 = "2018-9-2"
>>> date1.split("-")
['2018', '9', '2']
>>> date1.split("-",1)
['2018', '9-2']
>>>
```

Let us consider a different example. Suppose that, we have an IP address - for example "`192.168.0.1`", and we want to print extract "`192.168.0`":

Let us write the piece of code:

```
>>> ip1 = "192.168.10.1"
>>> str1 = ip1.split(".")
>>> str1
['192', '168', '10', '1']
>>>
>>> str1[0]+"."+str1[1]+"."+str1[2]
'192.168.10'
>>>
```

We can accomplish the same task using the `rsplit` method.

```
>>> ip1.rsplit(".",1)[0]
'192.168.10'
>>>
```

Partition methods

The following methods are used to make partitions of the string.

- `Str.partition("separator"):`

  The `partition` method returns a tuple containing three things - before `separator`, separator and after `separator`. If the string `str` contains more than one occurrence of a separator, then the first occurrence will be used:

  ```
  >>> str1 = "what we think we become"
  >>> str1.partition("we")
  ('what ', 'we', ' think we become')
  >>>
  ```

- `Str.rpartition("separator"):`

  The `rpartition` is the right brother of partition method. If the string `str` contains more than one occurrence of separator, then the last occurrence will be used:

  ```
  >>> str1.rpartition("we")
  ('what we think ', 'we', ' become')
  >>>
  ```

Join method

The `join` method makes a string by joining the items of a sequence.

lease see the below syntax:

```
str.join(seq)
```

- `seq:` It contains the sequence of the separated strings.
- `str:` It is the string that is used to replace the separator of the sequence.

The `join()` method returns a string, which is the concatenation of the given sequence and the string, as showcased in the following example:

```
>>> str1 = "-"
>>> seq = ["Hello","Jarvis", "!"]
>>> str1.join(seq)
'Hello-Jarvis-!'
>>> "".join(seq)
'HelloJarvis!'
>>> " ".join(seq)
'Hello Jarvis !'
>>>
```

String Boolean methods

The Boolean method returns `True` or `False`:

startswith()

Please see the following syntax:

```
str.startswith(str1, beg=0,end=len(string));
```

The `startswith()` method returns `true`, if a string `str` starts with the string specified by the `str1` parameter. The "beg" and "end" parameters are used to slice the string `str`:

```
>>> str1 = "Time is money"
>>> str1.startswith("Ti")
True
>>> str1.startswith("is")
False
```

```
>>> str1.startswith("is",5,8)
True
>>>
```

endswith()

This method returns `True` if the string ends with the specified substring, otherwise, it returns `False`.

Please see the following syntax:

```
str.endswith(suffix[, start[, end]])
```

Please see the following example to understand the use of `start` and `end`, to generate a slice of string `str`.

```
>>> str1 = "it is not easy to play another man's game"
>>> str1.endswith("is")
False
>>> str1.endswith("is",2,5)
True
>>> str1.endswith("game")
True
>>>
```

```
islower()
Please see the following syntax:
str.islower()
```

The `islower()` method returns true if the string contains only lower cased character(s), else it will return false. See the following examples:

```
>>> str1 = "Hellojarvis"
>>> str1.islower()
False
>>> str1 = "hellojarvis"
>>> str1.islower()
True
>>> str1 = "hellojarvis "
>>> str1.islower()
True
```

```
>>> str1 = "hellojarvis!"
>>>
>>> str1.islower()
True
>>>
```

The method only concerns with the upper and lower case, not with the special symbols.

Similarly, we can use `isupper()` and `istitle().`

isdigit()

Please see the following syntax:

```
str.isdigit()
```

The `isdigit()` method returns `true` if the string contains only digit(s), otherwise it returns `false`. See the following example:

```
>>> "123".isdigit()
True
>>> "123 ".isdigit()
False
>>> "123!@".isdigit()
False
>>>
```

isalpha()

Please see the following syntax:

```
str.isalpha()
```

The `isalpha()` method returns `true` if the Python string contains only alphabetic character(s), otherwise it returns `false`:

```
>>> "ABCxy".isalpha()
True
>>> "ABCxy12".isalpha()
False
>>>
```

isalnum()

Please see the following syntax:

```
str.isalnum()
```

The `isalnum()` method is used to determine whether the string consists of alphanumeric characters, otherwise it returns `false`:

```
>>> "ABCxy12".isalnum()
True
>>> "ABCxy12!@".isalnum()
False
>>> "ABCxy".isalnum()
True
>>>
```

isspace()

Please see the following syntax:

```
str.isspace()
```

The `isspace()` method returns true if the Python string contains only white space(s). See the following examples:

```
>>> " ".isspace()
True
>>> "".isspace()
False
>>> "sda ".isspace()
False
>>>
```

format()

The `format()` method is used to format the string in a better way. The method allows multiple substitutes and configuration of values. This method allows us to concatenate items through positional formatting within a sequence. The method uses the { } curly brackets as a place holder. The place holder in the format method can take positional as well as key worded parameters. Let us see the examples:

```
>>> Str1 = "Student {name} got {marks}"
>>> Str1
```

```
'Student {name} got {marks}'
>>> Str1.format(name= "Mohit", marks= 90)
'Student Mohit got 90'
```

Let us see how we can make SQL query

```
>>> "Select * from {table} where id = {num}".format(table= "student", num=3)
'Select * from student where id = 3'
>>>
You can also use positional parameters.
>>> "Select * from {0} where id = {1}".format("student", 3)
'Select * from student where id = 3'
>>>You can also leave the curly brackets blank.
>>> "Select * from {} where id = {}".format("student", 3)
'Select * from student where id = 3'
>>>
```

The format method is similar to the f-string. In order to make an f-string, we put "f" as a prefix. F-strings provide a straightforward and convenient way to integrate python expressions into the layout of string literals. See the following example.

```
>>> table = "student"
>>> num = 4
>>> q = f"Select * from {table} where id = {num}"
>>> q
'Select * from student where id = 4'
>>>
```

String functions

There are some functions that can be applied on strings.

Max()

The `max()` method returns the max character from the string `str` according to the ASCII value. In the first print statement, y is the maxed character, because the ASCII code of y is 121. In the second print statement s is the maxed character, since the ASCII code of s is 115:

```
>>> str1 = "Time is money"
```

```
>>> max(str1)
'y'
>>> str1 = "Time is moneY"
>>> max(str1)
's'
>>>
```

min()

Please see the following syntax:

```
min(str)
```

The `min()` method returns the `min` character from string `str` according to the ASCII value:

```
>>> str1 = "Time is moneY"
>>> min(str1)
' '
>>> str1 = "hello!@"
>>> min(str1)
'!'
>>>
```

Conclusion

In this chapter, we have learned about the string. The string is a very important data-type of programming. We have seen the string methods, which are defined in the string class, only meant for the string. There are some functions that can be applied on the string or other data-types. All the methods return a new string the original string remains the same due to the immutability of string. The string method can be known by using the `dir()` function. In the next chapter, you will learn about the tuple and list sequence.

Question

1. See the series of commands and give the answer

```
>>> name = "Hello Jarvis"
>>> name.upper()
'HELLO JARVIS'
```

```
>>> name
```

Answer:

Hello Jarvis

2. Write the code snippet

```
  ip = "192.168.0.1"
<code>
```

```
Output: ['192.168.0', '1']
```

Answer: ip.rsplit(".",1)

3. See the following code snippet:

```
>>> name = "Hello Jarvis"
>>> name[-3:-9]
```

Chapter 5
Tuple and List

In today's world *data is everything*. A person or an organization having huge data can become the king. In programming, to store data of different data types, we need containers. In Python, tuple and list act as container s that can store data without any limitation (depending on the RAM). In this chapter, we will learn about the tuple and list.

Structure

- Tuple
- Indexing of tuple
- Slicing of tuple
- Tuple methods
- Tuple function
- List
- List operation
- List functions
- List methods
- List comprehension

Objective

In this chapter, we will learn about the tuple and the list data structure. Both are containers. We will learn about both individually. First, we will learn about tuple, creation of tuple, indexing of tuple, slicing of tuple, method, and the function that can be applied on a tuple. After tuple, we will learn about the list. The indexing and slicing of the list and tuple are the same. The list contains more methods than a tuple. In the end, we will learn about list comprehension.

Tuple

Tuples are a type of sequence, like strings. But unlike strings, tuples can contain elements of any kind, which means that you can have a tuple that can store name, number, and score, and so on. A tuple is like a container, which can contain heterogeneous elements. Tuples are the immutable sequence; it means once we have defined the tuple, we don't have methods to update it.

Creating tuple

In this section, you will learn how to create a Python tuple.

Empty tuple

```
Tuple = ()
```

The empty tuple is written as two parentheses containing nothing.

Please the following example of the "If" condition with an empty tuple:

```
t1 = ()
if t1:
    print ("Something in Tuple")
else :
    print ("Tuple is empty")
```

The output is showcased in the following screenshot:

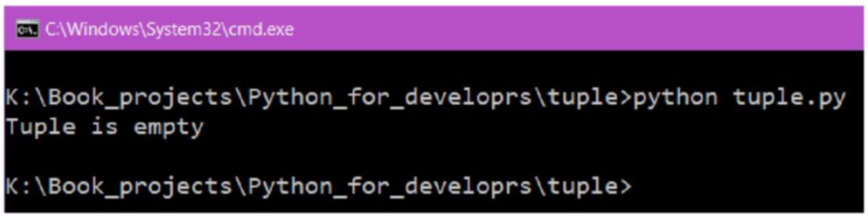

Figure 5.1

Creating tuple with the items

Create a tuple, fill the items in the tuple separated by commas. See the following examples:

```
tup1 = ('Python ', 'PHP', 1900, 799)
tup2 = (1,7,9,5 )
```

If you don't use parentheses, the interpreter will still take the items in a tuple.

For Example:

```
>>> tup2 = 1,3,4,8,"a"
>>> tup2
(1, 3, 4, 8, 'a')
>>> type(tup2)
<class 'tuple'>
>>>
```

Indexing tuple

You can specify a position number in the bracket, to access a particular element. Let us check this through an example:

```
>>> tup1 = ('Thor', 'Cap-America','Iron-man','Hulk','Spider-man')
>>> tup1
('Thor', 'Cap-America', 'Iron-man', 'Hulk', 'Spider-man')
>>> tup1[0]
'Thor'
>>> tup1[4]
'Spider-man'
>>> tup1[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

If the given index value is not present, then interpreter raises the "`IndexError`":

```
>>> tup1[-1]
'Spider-man'
```

```
>>> tup1[-4]
'Cap-America'
>>> tup1[-5]
'Thor'
>>> tup1[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>>
```

Negative indexing also works. To obtain the first item of the tuple, we use index `0`, for the last item, we use index `-1`.

The following figure shows the indexing of the tuple:

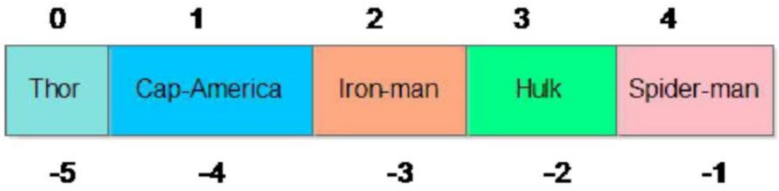

Figure 5.2

Slicing of tuple

You can assign a beginning and the end position. The result is a tuple containing every element between those positions. The following figure illustrates the pictorial representation of slicing:

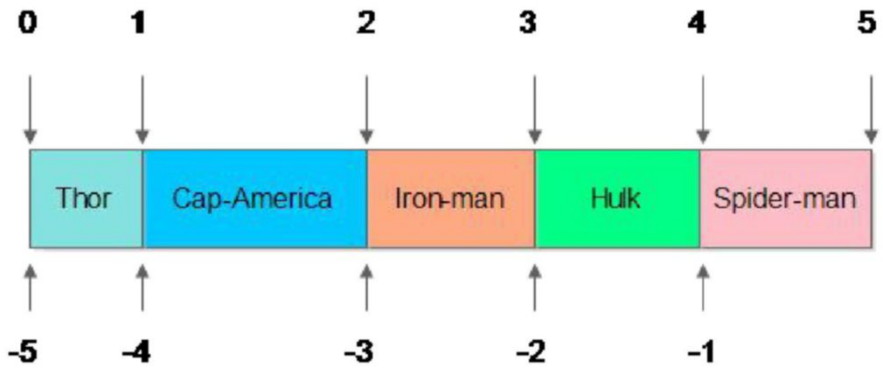

Figure 5.3

```
>>> tup1
('Thor', 'Cap-America', 'Iron-man', 'Hulk', 'Spider-man')
>>> tup1[2:5]
('Iron-man', 'Hulk', 'Spider-man')
>>>
>>> tup1[0:5]
('Thor', 'Cap-America', 'Iron-man', 'Hulk', 'Spider-man')
>>>
```

We can give a negative slicing too:

```
>>> tup1[-4:-2]
('Cap-America', 'Iron-man')
>>>
>>> tup1[-4:4]
```

We can give a combination of positive and negative index too:

```
('Cap-America', 'Iron-man', 'Hulk')
>>>
>>> tup1[1:]
('Cap-America', 'Iron-man', 'Hulk', 'Spider-man')
```

If we want to obtain items from a specific index to the last index, we can leave the last index blank:

```
>>> tup1[:3]
('Thor', 'Cap-America', 'Iron-man')
>>>
>>> tup1[1:10]
('Cap-America', 'Iron-man', 'Hulk', 'Spider-man')
>>>
```

In slicing, the interpreter does not throw any error if the index gets out of range:

```
>>> tup1[20:10]
()
>>>
>>> tup1[0:5:2]
```

```
('Thor', 'Iron-man', 'Spider-man')
>>>
```

The third integer represents the steps or difference between two consecutive items.

A tuple is the immutable data structure; we cannot delete the element from the tuple, although but we can delete the entire tuple:

```
>>> del tup1
>>> tup1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tup1' is not defined
>>>
```

Note: Indexing and slicing of tuple and list are the same.

Tuple methods

The following methods are associated with the tuple class:

`count()`

Please see the following syntax:

```
tuple.count(given item)
```

The count method is used to count the total occurrence of a given item:

```
>>> tup1 = (90,56,34,23,1,2,3,4,1,3,4,1)
>>> tup1.count(1)
3
>>> tup1.count(2)
1
>>> tup1.count(11)
0
>>>
```

If the given item is not present, then the frequency is 0.

If you want to obtain the index of a particular item, then use a method called index.

index()

Please see the following syntax:

```
Tuple.index(given item)
```

The index method returns the index of the given item:

```
>>> tup1
(90, 56, 34, 23, 1, 2, 3, 4, 1, 3, 4, 1)
>>> tup1.index(90)
0
>>> tup1.index(1)
4
```

The method returns the index of the first occurrence:

```
>>> tup1.index(91)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>>
```

If the item is not present, then the `index()` method gives an error.

Tuple functions

There are some functions that can be applied to the tuple.

len()

The `len()` function gives the length of the tuple, which means that it returns the total number of items present in a tuple:

```
>>> tup1 = ("iron-man", "vision","Thor", "hulk")
>>> len(tup1)
4
```

max()

The `max(tuple)` function gives the element of tuple with maximum value:

```
>>> tup1 = (1,2,3,4)
```

```
>>> len(tup1)
4
>>>
```

Min()

The min(tuple) function gives the element of tuple with minimum value:

```
>>> min(tup1)
1
>>>
```

Note: In Python 3, we cannot perform max and min operations if the tuple contains the items of different data-types.

Operations of Tuples

In this section, you will see the use of the operators in tuple.

Addition of tuples

With the help of the + operator, we can add two tuples:

```
>>> tup = (1,2)
>>> tup2 = (3,4)
>>> tup + tup2
(1, 2, 3, 4)
```

Multiplication of tuple

We can multiply a tuple with an integer:

```
>>> tup*2
(1, 2, 1, 2)
```

The multiplication with an integer does not create new items; it just creates new references. See it in more detail, in the following screenshot:

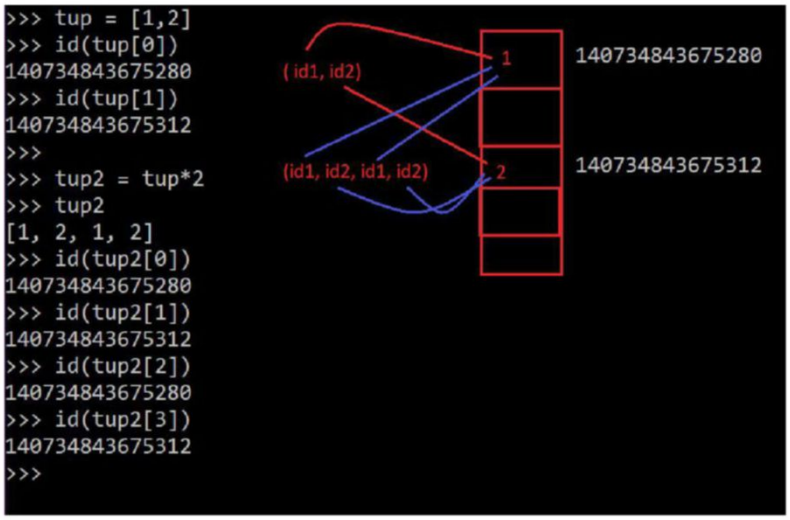

Figure 5.4

In the preceding screenshot, the tuple `tup2` contains the reverences of the objects, multiple times (memory address).

In operator

We can use the in operator to check the existence of an item:

```
>>> tup1 = (1,5,3,"a")
>>> 1 in tup1
True
>>> "x" in tup1
False
>>> "x" not in tup1
True
>>>
```

The not in can also be used to check the non-existence of the item.

for loop with tuple

Let us see how to use the "for" loop with tuple:

```
tup1 = ("a", 1,3,5,5.6)
```

```
for each in tup1:
  print (each)
```

The output is showcased in the following screenshot:

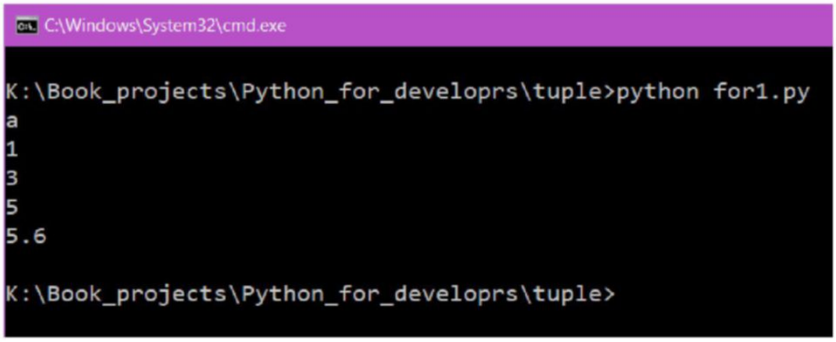

Figure 5.5

To understand the working of the preceding code, see the following figure:

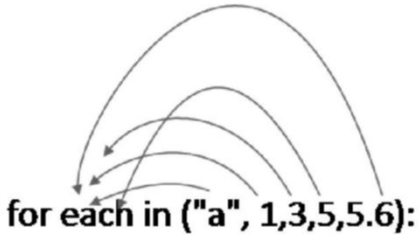

Figure 5.6

For each iteration, the item of the tuple gets assigned to the 'each' variable.

Unpacking of tuple

You can also unpack the tuple items to the corresponding variables.

```
Let us understand this with an example:
>>> tup1 = (9,5,8)
>>> a,b,c = tup1
>>> a
9
>>> b
5
>>> c
8
>>>
```

The tuple `tup1` contains three items. In the second statement, the items of tuple assign to the corresponding variables `a`, `b`, and `c`.

But if you use fewer number of variables than the number of tuple items, then the interpreter raises an error:

```
>>> a,b = tup1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>>
```

If you use a greater number of variables than the number of tuple items, then the interpreter will raise an error again:

```
>>> a,b,c,d = tup1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 4, got 3)
>>>
```

Similarly, the tuple operation can also be applied on the list. In the next section, we will learn about the list.

List

A list is also a container, like a tuple, which can store miscellaneous items. Unlike tuple, the list is a mutable data structure. We have list methods that can change the list.

Creating a list

In this section, we will learn about all the ways to create a list.

Empty list

```
list1 = []
```

The empty list is created by writing two square brackets containing nothing.

List with the elements

To create a list, fill the items in the square brackets, separated by commas.

For example, marvel's heroes:

```
A = ['Tony', 'Steve', 'Thor', 'Bruce']
```

List operations

As the list is a mutable sequence so you can add, delete, access, and update elements from the list.

Accessing item of a list

To obtain a list item, use the list name or identifier with an index in square brackets. The following is a simple example:

```
>>> A = ["Captain", "IRON-Man", "Thor", "HULK"]
>>> A[0]
'Captain'
>>> A[1]
'IRON-Man'
>>> A[2]
'Thor'
>>> A[3]
'HULK'
>>> A[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> A[2:3]
['Thor']
>>>
```

If the index is not there, then the interpreter gives an error. You can also obtain a slice of the list by providing the indices of the list.

Updating list

The list is a mutable data structure; we can update the existing item of the list with the help of the index:

```
>>> A
['Captain', 'IRON-Man', 'Thor', 'HULK']
>>> A[0] = "Captain America"
```

```
>>> A
['Captain America', 'IRON-Man', 'Thor', 'HULK']
>>>
```

The above example shows that Captain is changed to Captain-America.

In the preceding example the item "Captain" is changed to "Captain-America".

Deleting list item

With the help of `del` and list index, we can delete any item by specifying its index.

Let us discuss in the next example:

```
>>> A
['Captain America', 'IRON-Man', 'Thor', 'HULK']
>>> del A[3]
>>> A
['Captain America', 'IRON-Man', 'Thor']
>>> del A
>>> A
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined
>>>
```

To delete more than one item, you can provide indices. See the following example:

```
>>> A = ['Captain America', 'IRON-Man', 'Thor', 'HULK']
>>> del A[2:4]
>>> A
['Captain America', 'IRON-Man']
>>>
```

Addition of the lists

Two lists can be added using the + operator:

```
>>> A = ['Captain', 'IRON-Man', 'Thor', 'HULK']
>>>
>>> A2 = ["C-Marvel", "Vision"]
```

```
>>> A + A2
['Captain', 'IRON-Man', 'Thor', 'HULK', 'C-Marvel', 'Vision']
>>>
```

Multiplication of List

Similarly, you can use the multiplication operator * to multiply the list. See the following example:

```
>>> A2
['C-Marvel', 'Vision']
>>> A2*2
['C-Marvel', 'Vision', 'C-Marvel', 'Vision']
>>>
```

in operator

With the help of the in operator, we can check the membership of an item:

```
>>> A
['Captain', 'IRON-Man', 'Thor', 'HULK']
>>>
>>> "Thor" in A
True
>>> "HULK" not  in A
False
>>>
>>> "Vision"   in A
False
>>>
```

List with for loop

The working of the `for` loop with the tuple and the list is the same. The `for` loop can fetch the list's item one-by-one. See the following example:

```
>>> list1 = [0,4,5,2,56,3,4]
>>> for i in list1:
...     print (i)
...
```

```
0
4
5
2
56
3
4
>>>
```

For each `for` loop iteration, every item of the list is assigned to the variable i. Thus, we iterate over the list.

List functions

In this section, we will discuss the built-in python functions, which can be applied on the list.

len()

The `len()` function returns the length of the list:

```
>>> A
['Captain', 'IRON-Man', 'Thor', 'HULK']
>>> len(A)
4
>>>
```

max()

The `max()` function returns the items from the list with the maximum value. Let us discuss it with an example:

```
>>> list1 = [0,5,3,2,1]
>>> max(list1)
5
>>> list2 = ["aaa", "aa", "aab", "aaz"]
>>> max(list2)
'aaz'
>>>
```

For the string items, the interpreter checks the first character of each string. If the first character is the same then check the second, third and so on, until the character with the maximum ASCII value is found:

```
>>> list3 = ["ab", "b", 1,2]
>>> max(list3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'int' and 'str'
>>>
```

In Python 3, the mixed data type is not allowed, although the mixed type is permitted in Python 2.

In Python 3, the comparison operator does not compare the items of different data types. In Python 2, you can compare the string with an integer, but in Python 3, it does not allow for it. See the following examples:

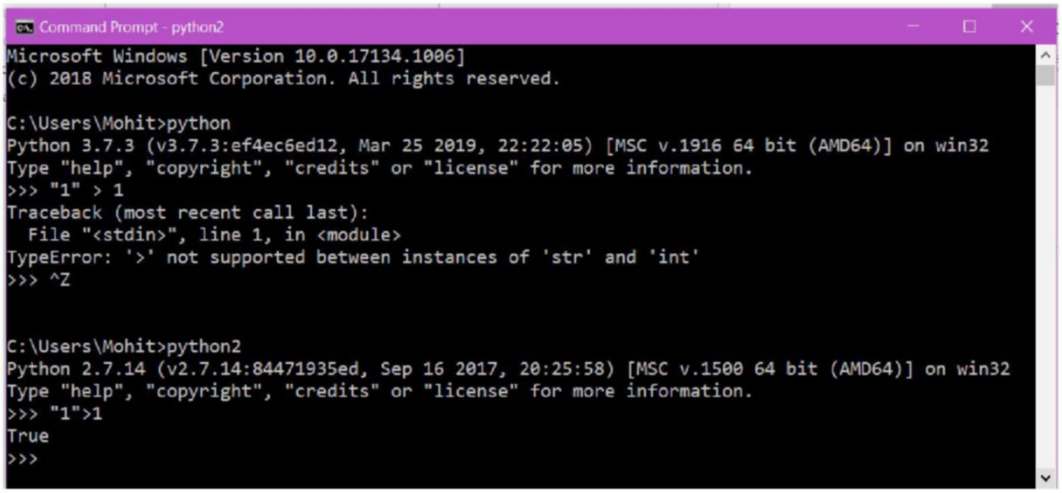

Figure 5.7

Similarly, we can use the `min()` function:

```
>>> list1
[0, 5, 3, 2, 1]
>>> min(list1)
0
>>> list2
['aaa', 'aa', 'aab', 'aaz']
```

```
>>> min(list2)
'aa'
>>>
```

List methods

There are several methods, which are offered by list.

To check all the methods of list, use the dir function:

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__
dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

Insert methods

We will see the method used to insert the items in the list.

If you want to add the items at the end of the list, then use the append method.

append()

The syntax of the append method is `list.append().`

This method adds an item at the end of the existing list:

```
>>> list1 = [7,8,6,3,2]
>>> list1
[7, 8, 6, 3, 2]
>>> list1.append(90)
>>> list1
[7, 8, 6, 3, 2, 90]
>>> list1.append(23)
>>> list1
[7, 8, 6, 3, 2, 90, 23]
>>> Avengers = ["Thor", "Captain"]
```

```
>>> Avengers
['Thor', 'Captain']
>>> Avengers.append("IRON-MAN")
>>> Avengers
['Thor', 'Captain', 'IRON-MAN']
>>>
>>>
```

In this way, an item can be added at the end by using the `append` method.

If you want to add a complete sequence at the end, then use the `extend()` method.

See the following syntax of extend method:

```
list1.extend(sequence)
```

The `list1` is the primary list. The sequence must be an iterable object like tuple, list, string, and so on. The `extend()` method extends the primary list by appending all the items of the sequence.

```
>>> list1 = [1,2,3]
>>> tup1 = (7,8)
>>> list1.extend(tup1)
>>> list1
[1, 2, 3, 7, 8]
>>>
>>> Av = ["captain", 'HULK', "IRON-MAN"]
>>> Av2 = ["IRON-MAN", "Black Widow"]
>>> Av.extend(Av2)
>>> Av
['captain', 'HULK', 'IRON-MAN', 'IRON-MAN', 'Black Widow']
>>>
```

You can see the difference between append and extend. The `append` method always considers the argument as a single item and the `extend` method considers the argument as a sequence. See the following example for more clarity:

```
>>> list1 = ["a", "b"]
>>> list1.extend("Mohit")
>>> list1
['a', 'b', 'M', 'o', 'h', 'i', 't']
>>>
```

```
>>> list1.extend(45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>>
```

If you want to add an item at a desired place, then use the insert method.

insert()

See the following syntax of the insert method:

```
list.insert(index_number, item)
```

The `index_number` is the position where the given item has to be inserted. The second argument is the item, that to be inserted into the list. See the following examples:

```
>>> infinity_gems = ["Space","Mind","Reality", "Soul","Power"]

>>> infinity_gems.insert(3,"Time")
>>> infinity_gems
['Space', 'Mind', 'Reality', 'Time', 'Soul', 'Power']
>>>
```

So in the preceding example, if I forgot to put `Time` stone, and I want to keep `Time` stone after `Reality`, in this situation, I used `insert()`, and pass 3 as the `index_number` and `Time` as the item.

Deletion

In this section, we will see the methods used in the deletion of an item from a list.

If you want to remove the item, but don't know the index of the item to be removed, then you can use the `remove` method.

remove()

See the following syntax of the `remove` method:

```
list.remove(item)
```

The `remove()` method is used to remove the item from the list. In the `remove` method, we need to specify the item name.

For example:

```
>>> A =  ['captain', 'iron-man', 'thor', 'hulk', "Thanos"]
>>> A.remove("Thanos")
>>> A
['captain', 'iron-man', 'thor', 'hulk']
>>>
```

In the preceding example, Thanos is removed by the remove() method:

```
>>> list1 = [1,2,3,4,1,1]
>>> list1.remove(1)
>>> list1
[2, 3, 4, 1, 1]
>>>
```

The `remove()` method only removes the first occurrence from the list.

The `remove` method does not return the removed item, but the pop method does. Let us see the `pop` method.

pop()

See the following syntax of the pop method:

```
list.pop(index)
```

The `pop()` method not only deletes the item, but also returns the deleted item from the list.

We need to specify the index of the item to be removed:

```
>>> infinity_gems
['Space', 'Mind', 'Reality', 'Time', 'Soul', 'Power']
>>> infinity_gems.pop(1)
'Mind'
>>>
```

You can see that the `pop()` method removed the item, which was at the specified index.

If you don't provide the index number, then the `pop()` method removes the last item:

```
>>> infinity_gems
['Space', 'Reality', 'Time', 'Soul', 'Power']
>>> infinity_gems.pop()
'Power'
>>> infinity_gems
['Space', 'Reality', 'Time', 'Soul']
>>> infinity_gems.pop()
'Soul'
>>> infinity_gems
['Space', 'Reality', 'Time']
>>>
```

The next method, count, returns the occurrence of the given item.

count()

See the following syntax of the count method:

```
list.count(item)
```

The `count()` method returns the frequency of the specified item in the list. Let's check the following example:

```
>>> list1 = [1,2,3,6,9,3,2,1,2,4,5]
>>> list1.count(1)
2
>>> list1.count(2)
3
>>> list1.count(7)
0
>>>
```

If the item is not present, then the `count()` method returns `0`.

If you want to know the index of an item in the list, then the `index` method allows you to do so.

index()

See the following syntax of the index method:

```
list.index(item)
```

The `index` method is used to find the index of a given item. If the given item has occurred two times, then the method finds the index of the first occurrence.

Let's check the following example:

```
>>> list1
[1, 2, 3, 6, 9, 3, 2, 1, 2, 4, 5]
>>> list1.index(1)
0
>>> list1.index(3)
2
>>>
```

Copy()

Please see the following syntax:

```
list1.copy()
```

The copy method is used to make a copy of the list:

```
>>> A = ["iron-man", "Thor"]
>>> A1 = A.copy()
>>> A1
['iron-man', 'Thor']
>>> id(A)
2665188319944
>>> id(A1)
2665190345416
>>>
>>> A1.append("Hulk")
>>> A
['iron-man', 'Thor']
>>> A1
['iron-man', 'Thor', 'Hulk']
>>>
```

The different address shows that `A1` was the copy of `A`.

There is one more different technique to create a copy of the list:

```
>>> A1
['iron-man', 'Thor', 'Hulk']
>>> id(A1)
2665190345416
>>> A2 = A1[:]
>>> id(A2)
2665190151240
>>>
```

Note: The above technique would not work on the tuple.

In the next method, we will see how to sort the items of the list. With the help of a few examples, we will see the benefits of the sort method.

Sort()

Please see the following syntax:

```
list.sort(key=None, reverse=False)
```

The `sort()` method sorts the items of the list. The Python sort is stable and in-place. In a stable sort, the order of the elements that compare equal will be preserved and in an in-place sort, the sorting does not take extra memory.

In Python 3, the sort only works on homogeneous items.

Let us see the first case.:

```
>>> list1 = [1,2,0,1,5,2,7,2]
>>> list1.sort()
>>> list1
[0, 1, 1, 2, 2, 2, 5, 7]
>>>
```

For reverse order, use the keyword argument "reverse = `True`" or "reverse = `1`":

```
>>> list2 = ['azz','az','azy']
>>> list2.sort()
>>> list2
['az', 'azy', 'azz']
>>>
```

The strings are compared based on their ASCII value.

Special case:

Example 1:

Consider a list of tuples and sort the list according to the second element of the tuple:

```
list1 = [("a",30),("b",20),("c",10),("d",40),("e",45)]
Let use the “key” keyword
list1 = [(10,"a"),(20,"d"),(5,"b"),(10,"c"),(4,"e")]
def fun1(x):
  return x[1]
list1.sort(key = fun1)
print (list1)
```

The output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\list1>python sort3.py
[(10, 'a'), (5, 'b'), (10, 'c'), (20, 'd'), (4, 'e')]

K:\Book_projects\Python_for_developrs\list1>
```

Figure 5.8

Every item of the list is passed to the function `fun1`. The argument x takes the tuple at each iteration.

The `fun1` function returns the second element of the tuple, and based on the second element, the sort method sorts the list.

Example 2:

Consider a list of tuples, and the tuple contains integers, sort the list based on the sum of the elements of the tuple:

```
list1 = [(10,20),(4,5,8),(7,8),(45,1,4),(2,3)]
def fun1(x):
  c = sum(x)
  return c
list1.sort(key=fun1)
print (list1)
```

The output is showcased in the following screenshot:

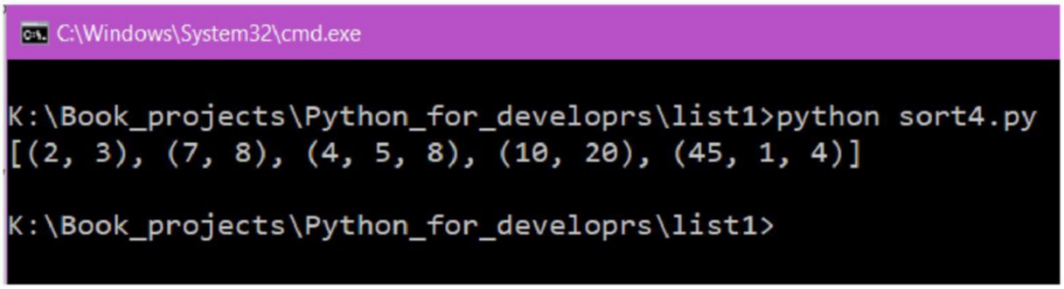

Figure 5.9

Example 3:

The next problem is to sort the IP address based on the last octet.

We generally take an IP address as a string:

```
list_ip = ["192.168.1.100","192.168.1.2","192.168.1.09","192.168.1.99","
192.168.1.10"]
def fun1(ip):
  c=ip.rsplit(".",1)[-1]
  return int(c)
list_ip.sort(key = fun1)
print (list_ip)
```

The output is showcased in the following sreenshot:

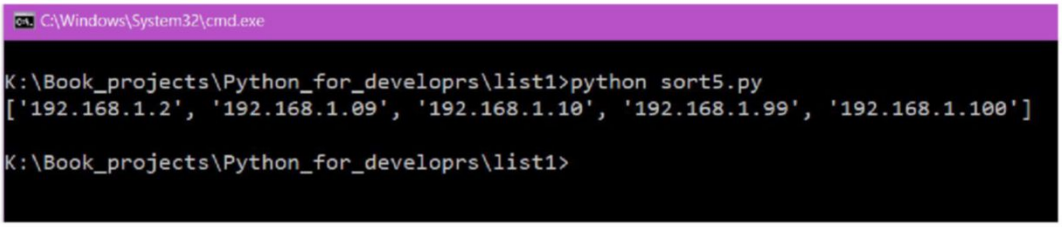

Figure 5.10

The syntax `ip.rsplit(".",1)[-1]` returns the last octet of the IP address. Based on the last octet integer, the sort method sorts the list.

Example 4:

Consider a case, a list that contains all integers, the sort is being performed in the ascending order, but you want to put all the zeros at the right side:

```
list1 = [30,40,0,41,31,50,0,23,0,6,0,12,14,15,19,3,2,1,0,7]
```

```
max1 = max(list1)+1
def fun1(x):
  if x ==0:
    return max1
  else :
    return x
list1.sort(key=fun1)
print (list1)
```

The output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\list1>python sort6.py
[1, 2, 3, 6, 7, 12, 14, 15, 19, 23, 30, 31, 40, 41, 50, 0, 0, 0, 0, 0]

K:\Book_projects\Python_for_developrs\list1>
```

Figure 5.11

In the preceding example, the element zero has been sorted based on the max value of the list.

Sometimes, we need to reverse the list. The reverse method facilitates us to reverse the list.

reverse()

See the following syntax of the reverse method:

```
list.reverse()
```

The `reverse()` method reverses the items of the list:

```
>>> list1 = [1,3,4,"a","b"]
>>> list1.reverse()
>>> list1
['b', 'a', 4, 3, 1]
>>>
```

List comprehensions

The list comprehension is a concise way to create lists. In list comprehension, we write one-line codes to make a list with the `for` loop.

Consider a simple example, that list contains the square of numbers:

```
squ = []
for x in range(1,6):
   squ.append(x**2)
print (squ)
```

You can do the same thing by Python list comprehensions:

```
>>> [x*2 for x in range(1,6)]
[2, 4, 6, 8, 10]
>>>
```

Let us consider one more example, find the even numbers.

Let us see a simple program:

```
list_even = []
for each in range(1,51):
  c = each%2
  if c ==0:
    list_even.append(each)
print (list_even)
```

The same thing can be achieved with the Python list comprehensions:

```
>>> print ( [each for each in range(1,51) if each%2==0  ]  )
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
40, 42, 44, 46, 48, 50]
>>>
```

Exercise:

```
list1 = ["2019-6-20","2019-4-4","2019-6-23","2019-6-21","2019-7-20"]
```

In the preceding list, obtain the day from the dates.

Exercise

1. Merge two sorted lists with following condition.
   (*a*) The final list must be sorted.
   (*b*) Only one loop can be used.

   Solution:

```
# merge two sorted lists
list1 = [1,3,7,10,14,20,25,28,100,104]
list2 = [4,8,11,23,26,90,93]

list3 = []
len_list1 = len(list1)
len_list2 = len(list2)
i = j =0
while i<len_list1 and j<len_list2:     #8<10 ,7<7
  if list1[i] < list2[j] :
    list3.append(list1[i])
    i=i+1
  else :
    list3.append(list2[j])
    j= j+1
if j==len_list2:
  list3.extend(list1[i:])
else :
  list3.extend(list2[j:])
print (list3)
```

Conclusion

You have seen the tuple and the list data structure. Both play an important role in development. The tuple is an immutable data structure, which does not contain a method to update the item.

Once you have defined the tuple, it is the same throughout the program, and you can delete the tuple. The list can be updated. The list can do all the operations that a tuple can do. You can also unpack, slice a list. The indexing is the same for tuple and list. The list contains the method to add, update, and delete. In the next chapter, you will learn the dictionary data structure.

Questions

1. What is the advantage of tuple over a list?

   Answer: A tuple is an immutable object, which cannot be changed, and tuple takes less memory then the list. See the following explanation:

```
>>> list1 = [1,2]
>>> tupl = (1,2)
>>>
>>> import sys
>>> sys.getsizeof(list1)
80
>>> sys.getsizeof(tupl)
64
>>>
```

2. Multiplication of list.

```
>>> list1 = [[]]
>>> list2 = list1*3
>>> list2
[[], [], []]
>>> list2[0].append(2)
>>> list2
?
```

   Answer: The result is [[2], [2], [2]].

   When we multiply a list, the returned list does not make new items in the memory, it just creates more references.

   For example:

```
>>> id(list1[0])
1774366026248
>>>
>>> id(list2[0])
1774366026248
>>> id(list2[1])
1774366026248
```

```
>>> id(list2[2])
1774366026248
>>>
```

See the following diagram for more clarity.

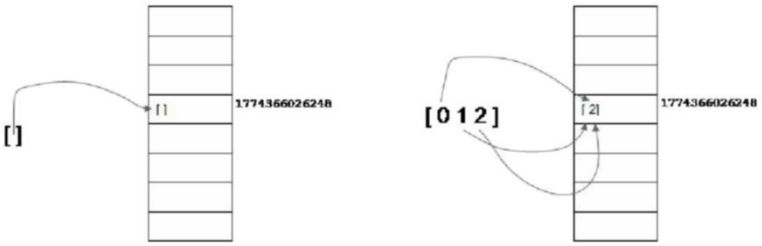

Figure 5.12

The figure shows that the index value 0,1,2 represents the same items in the memory.

3. Consider a `tup= (1,2,[])`, so, can we do that `tup[2].append(80)`?

   Answer: Yes, because immutability is not magic. It exists because we don't have a method to update the tuple. Here we are updating the list inside the tuple.

Chapter 6
Dictionary and Sets

In the last chapter *Tuple and List*, we have seen that we need the index to access any item. The index is always an integer, consecutive and starts with 0. Let's consider that you want to create your index where a string, float, and integer can be used. The dictionary allows you to create such an index. A dictionary allows you to map the different objects. In reality, a dictionary is like a collection of words and its meaning. Similarly, a Python dictionary offers you the collections of key and its value. We will learn about the dictionary methods and functions, and different operations that can be applied to the dictionary.

Structure

- Dictionary
- Operations of dictionary
- Dictionary function
- Dictionary methods
- Set

Objective

In this chapter, we will learn about the dictionary and set. We will learn how to create a dictionary, features of the dictionary, operations, and methods. Like list,

the dictionary is also a mutable data structure. In the end, we will see the set and its uses.

Dictionary

A dictionay is a data structure, which contains key and value pairs. A dictionary is written as a sequence of the key/value or item pairs separated by commas.

Let us see a few examples:

```
port = {20: "FTP", 23: "Telnet”, 53: "DNS", 80: "HTTP" }
```

```
Dict1 = {"AP": "Access Point", "IP": "Internet Protocol"}
```

Creating a dictionary

Let us see the syntax that creates a dictionary:

```
Dictionary_name = { key  : value  }
```

The key and value pair is referred to as an item; `key` and `value` are separated by a colon (:). The Each item is separated by a comma (,), the whole thing is enclosed in curly braces ({ }). An empty Python dictionary can be created by just two curly braces {}.

Features of dictionary

The following are the features of the dictionary:

- The key of the dictionary cannot be changed.
- Keys are unique.
- A `string, int, float,` and `tuple` can act as a key, provided the tuple does not contain any list.
- The value of the key can be changed.
- A value can be anything - for example list, tuple, and so on.
- Ordering is not significant; the order in which you have entered the items in the dictionary, may not get the details in the same order.

The data structure, which is a hashable type (all immutable), can be used as a key. A list is an unhashable type. The dictionary stores the value by taking the hash of the key. The average lookup time for a value using the key, is O(1).

Operations on dictionary

In this section, you will learn about the different types of operations on the dictionary.

Accessing the values of dictionary

The dictionary contains pairs of keys and values. You will learn how to obtain the value using its key from the dictionary.

The following are the examples of accessing the value, by using the key:

```
>>> port1 = {23: "Telnet", 20: "FTP", 80: "http", 53 : "DNS"}
>>> port1[23]
'Telnet'
>>> port1[20]
'FTP'
>>> port1[21]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21
>>>
```

The key works like an index. The preceding example shows that, to access the dictionary elements we need to use the square brackets along with the key. If a key is not present, then the `KeyError` occurs.

Deleting item from dictionary

With the help of a key, we can delete an item. See the following syntax that showcases how to delete an item.

- `del dict[key]:`

  You can delete a single item of the dictionary, as well as the entire dictionary.

  See the following command line code of deleting an item:

  ```
  >>> del port1[20]
  >>>
  >>> port1
  {23: 'Telnet', 80: 'http', 53: 'DNS'}
  >>> del port1
  >>> port1
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'port1' is not defined
  >>>
  ```

You can see the error because the dictionary named `port1` exists after the command `del port1`.

Updating and adding in the dictionary

The updating and adding an item in the dictionary is straight forward. See the following syntax:

```
dict[key] = new_value
```

You can update and add an item in the dictionary. To update, we need to specify the value's key in the square bracket and assign a new value. If the key is not present, then the key-value pair would be added.

Let us see an example of updating the value:

```
>>> port = {22: "SSH", 23: "SMTP" , 53: "DNS", 80: "HTTP" }
>>> port
{22: 'SSH', 23: 'SMTP', 53: 'DNS', 80: 'HTTP'}
>>> port[23]= "Telnet"
>>> port
{22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP'}
>>>
```

You cannot set more than one value to one key. If you do, this last assigned value is considered as the real value.

Adding item to dictionary

Let us see the addition of an item in the dictionary. The syntax would remain the same:

```
dict[new_key] = value
```

In the dictionary, adding an item is very easy. Specify the key in [] as showcased in the following examples:

```
>>> port1 = {23: "Telnet", 20: "FTP", 80: "http", 53 : "DNS"}
>>> port1[22] = "SSH"
>>> port1
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS', 22: 'SSH'}
>>> port1[23]= "Apache"
>>> port1
{23: 'Apache', 20: 'FTP', 80: 'http', 53: 'DNS', 22: 'SSH'}
>>>
```

One rule that applies here, is that the key cannot be duplicated.

Dictionary functions

In this section, you will learn about some dictionary functions. Let us see the functions one-by-one.

len(dict)

The `len()` function gives the total number of item(s) in the dictionary.

See the following example:

```
>>> dict1 = {22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
>>> len(dict1)
5
>>>
```

Max(dict)

The `max` function works on the keys of dictionary. The `max()` function returns the key with the maximum value.

See the following example:

```
>>> dict1 = {22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
>>> max(dict1)
443
```

Similary, the `min()` function returns the key with the minimum value.

Dictionary methods

Here we will discuss the dictionary methods. If you want to know the methods associated with the `dictionary` class, then use the dir function as showcased below:

```
>>> dir({})
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__
getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__
iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get',
'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
>>>
```

Let us see the methods one-by-one.

copy()

See the following syntax of the `copy()` method:

```
Dict.copy()
```

The `copy()` method allows you to make a copy of the existing dictionary:

```
>>> port1 = {23: "Telnet", 20: "FTP", 80: "http", 53 : "DNS"}
>>> port2 = port1.copy()
>>> id(port1)
2354409948288
>>> id(port2)
2354414650568
>>>
>>> port2
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS'}
>>>
>>> port1[443] = "HTTPS"
>>> port1
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS', 443: 'HTTPS'}
>>> port2
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS'}
>>>
```

The `port2` is the copy of `port1`.

If you access the value with the help of a subscript operator and if the key is not found, then the interpreter gives an error. To prevent the error, we use the `get()` method.

get()

See the following syntax of the `get()` method:

```
dict.get(key, default=None)
```

The `get()` method is used to get the value of the given key, if the key is not found the default value or message would return:

```
>>> port1
```

```
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS', 443: 'HTTPS'}
>>> port1.get(23)
'Telnet'
>>> port1.get(24)
>>> port1.get(24,"Value not found")
'Value not found'
>>> port1.get(53,"Value not found")
'DNS'
```

When a key is found, it prints its value. But when the key is not found; it prints the set the message "`Value not found`".

setdefault()

This method is similar to the `get()` method, but it adds the default value to the dictionary.

See the following syntax of the `setdefault()` method:

```
Dict.setdefault()
```

Let us understand by looking at examples:

```
>>> port1
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS', 443: 'HTTPS'}
>>>
>>> port1.setdefault(23)
'Telnet'
>>> port1.setdefault(24)
>>> port1
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS', 443: 'HTTPS', 24: None}
>>> port1.setdefault(25,"unknown")
'unknown'
>>> port1
{23: 'Telnet', 20: 'FTP', 80: 'http', 53: 'DNS', 443: 'HTTPS', 24: None,
25: 'unknown'}
>>>
```

You can see that for new keys, the default value has been added to the dictionary.

items()

The `items()` method returns the iterator, which contains the dict's (key, value) tuple pairs.

See the following syntax of the `items()` method:

```
dict.items()
```

Let us understand by looking at examples:

```
>>> dict1
{22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
>>> dict1.items()
dict_items([(22, 'SSH'), (23, 'Telnet'), (53, 'DNS'), (80, 'HTTP'), (443, 'HTTPS')])
```

Let us see how to use the "`for`" loop with dictionary:

```
dict1 = {22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
for each in dict1.items():
  print (each)
```

The output is showcased in the following screenshot:

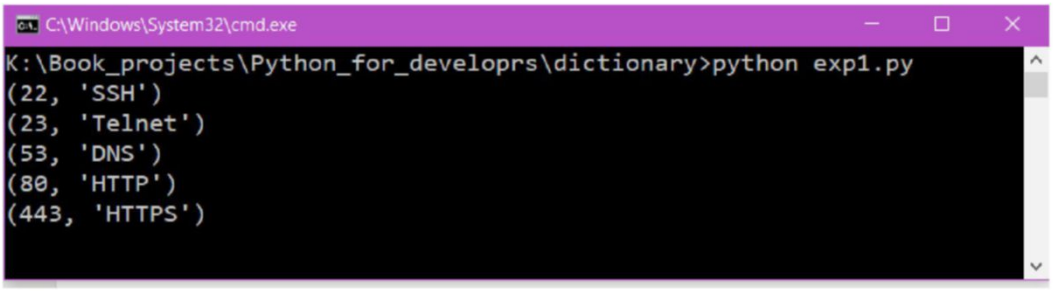

Figure 6.1

As you can see that we are getting the tuples. Let's unpack the tuple in the code.

Now, let's see the new code:

```
dict1 = {22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
for k,v in dict1.items():
  print (k,v)
```

The output is showcased in the following screenshot:

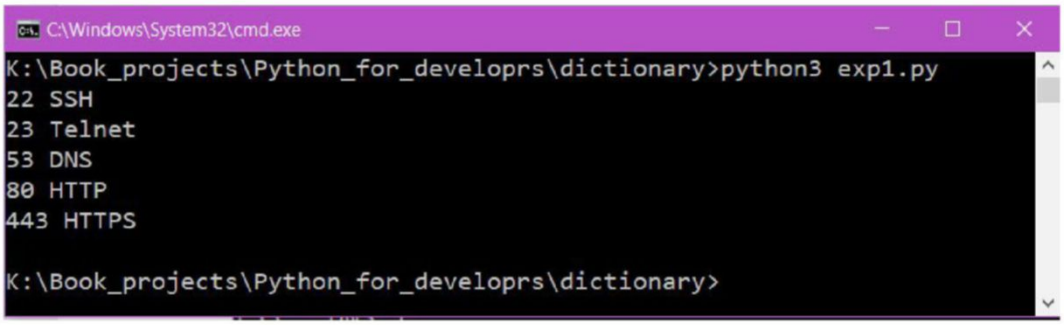

Figure 6.2

In this way, we can iterate over a dictionary.

keys()

The `keys()` method returns all keys of the dictionary.

See the following syntax of the `keys()` method:

```
dict.keys()
```

Let us see a couple of examples to understand the `keys()` method:

```
>>> dict1
{22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
>>>
>>> dict1.keys()
dict_keys([22, 23, 53, 80, 443])
>>>
```

values()

The `values()` method returns all values of the dictionary.

See the following syntax of the values() method.

```
dict.value()
```

Let us see some examples to understand the `values()` method:

```
>>> dict1
{22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
>>> dict1.values()
```

```
dict_values(['SSH', 'Telnet', 'DNS', 'HTTP', 'HTTPS'])
>>>
```

The `items(), keys()` and `values()` methods do not generate a list. It looks like the methods return a list. They return an iterator, which does not consume memory.

Let us see an example:

```
>>> dict1 = {20:"FTP"}
>>> import sys
>>> sys.getsizeof(dict1.items())
48
>>> dict1 = {20:"FTP",22 : "SSH"}
>>> sys.getsizeof(dict1.items())
48
>>> dict1 = {20:"FTP",22 : "SSH", 23: "Telent", 80: "HTTP"}
>>> sys.getsizeof(dict1.items())
48
>>>
```

The `sys.getsizeof()` method is used to obtain the size of an object in bytes. You can see that we are increasing the items in the dictionary, but the size remains the same.

You want to generate the list use the following syntax:

```
>>> dict1 = {20:"FTP"}
>>> import sys
>>> list_item = list(dict1.items())
>>> sys.getsizeof(list_item)
96
>>> dict1 = {20:"FTP",22 : "SSH"}
>>> list_item = list(dict1.items())
>>> sys.getsizeof(list_item)
104
>>> dict1 = {20:"FTP",22 : "SSH", 23: "Telent", 80: "HTTP"}
>>> list_item = list(dict1.items())
>>> sys.getsizeof(list_item)
120
```

As you can see that the size of variable `list_item` is increasing, as the dictionary's size increases.

The question is when to use `list(dict1.items())` and `dict1.items()`.

Let us run a program:

```
dict1 = {22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
for k,v in dict1.items():
  if k==23:
    del dict1[23]
  print (k,v)
```

The output is showcase in the following screenshot:

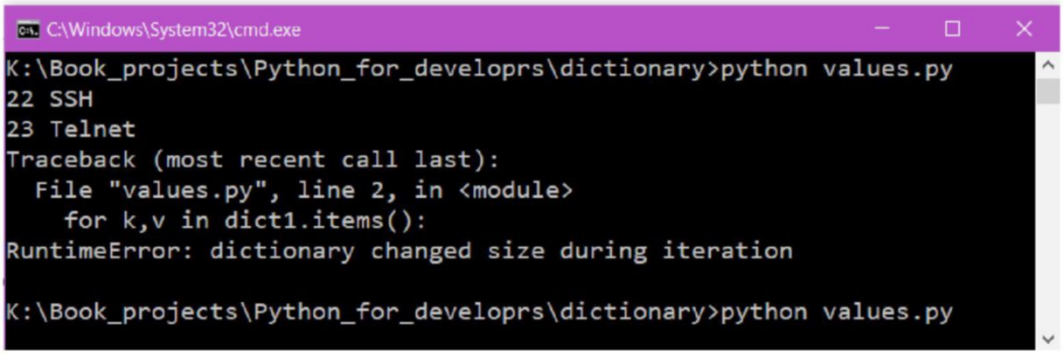

Figure 6.3

You can see the error. The `dict1.items()` does not consume memory and returns the next value on the fly. This means that `dict1.items()` calculates the next value and presents it to the '`for`' loop. If you change the dictionary in the middle of the iteration, then `dict1.items()` gives error. If you want to perform the operation while iterating, then use `list(dict1.items())`.

update()

Let's consider that you want to add one dictionary to another, you can take advantage of the `update()` method.

See the following syntax of the `update()` method:

```
dict.update(dict2)


dict2: This is the dictionary to be added.
```

Let us see some examples to understand the update method:

```
>>> dict1
{22: 'SSH', 23: 'Telnet', 53: 'DNS', 80: 'HTTP', 443: 'HTTPS'}
>>> dict2 = {3306: "DNS", 110 :"POP3"}
>>>
```

In the preceding example, port is updated with the new `port1`. In this way, you can add one dictionary to another dictionary.

If both the dictionaries contain the same key, then the corresponding of the key in `dict1` will be replaced by the value of the identical key of `dict2`.

Exercise

In this section, we will complete the exercise for more a better understanding of the dictionary.

Create one dictionary by using two lists:

```
list1 = [1,2,3,4,5]
list2 = ['a','b','c','d','e']
len1 = max(len(list1),len(list2))
dict1 ={}
for i in range(len1):
  dict1[list1[i]]= list2[i]
print (dict1)
```

Let us check the output in the following screenshot:

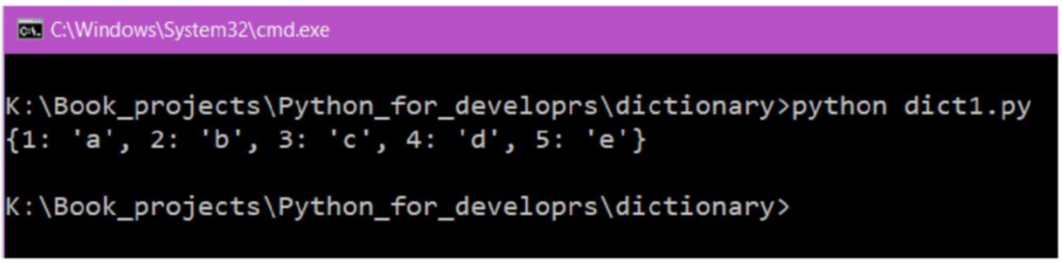

Figure 6.4

Let us accomplish the same thing in one line. First with the zip function:

```
>>> dict(zip(list1,list2))
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

Then without the zip function.

```
>>> dict([(list1[i],list2[i]) for i in range(len(list1))])
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
>>>
```

Set

In this section, we will learn about the sets. The set is like a dictionary, which contains only keys, not values. A set is an unordered and unindexed collection.

Let's see the declaration of a set with items or without items.

With item

To declare a set, we use curly braces.

For example:

```
>>> s1 = {1,2,"m"}
>>> s1
{1, 2, 'm'}
>>> type(s1)
<class 'set'>
>>>
```

Without items

Here we don't use curly braces. If we do, then the interpreter takes it as a dictionary.

For example:

```
>>> S2= set()
>>> type(S2)
<class 'set'>
>>>
```

Let us see the methods of sets.

```
>>> dir(s1)
['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__
doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__iand__', '__init__', '__init_subclass__', '__ior__', '__
isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__', '__ne__',
'__new__',  '__or__',  '__rand__',  '__reduce__',  '__reduce_ex__',  '__
```

```
repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear',
'copy', 'difference', 'difference_update', 'discard', 'intersection',
'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop',
'remove', 'symmetric_difference', 'symmetric_difference_update', 'union',
'update']
```

Now, let's explore the methods of the set.

add()

```
To add an item to a set:
>>> s1
{1, 2, 'm'}
>>> s1.add("k")
>>> s1
{'k', 1, 2, 'm'}
>>>
```

remove()

To remove the item from the set:

```
>>> s1.remove(2)
>>> s1
{'k', 1, 'm'}
>>>
```

You can try the rest of the methods of the set. The primary use of the set is to remove duplicating items. Let us see an example:

```
>>> list1 = [1,2,3,4,2,1,2,4]
>>> s3 = set(list1)
>>> s3
{1, 2, 3, 4}
>>>
```

In the preceding example, each element of the list behaves like the key of the set. The keys are always unique and that is why duplicate items are removed.

Conclusion

In this chapter you have learned about the dictionary, and that the dictionary contains key, value pairs. The key works like an index.

The dictionary is a mutable data structure. The key of a dictionary cannot be changed, values can be changed. With the help of the dictionary method, we can change add, update and delete the items from the dictionary. To remove duplicate items we use set. The set is like a dictionary, which contains only keys and not values. In the next chapter, will learn about functions.

Questions

1. Can we use tuple as a key of a dictionary?
2. What is the return value of `dict1.items()` in Python 3.

CHAPTER 7
Function

You might have seen an architecture of an organization such as a school or a college. The organization is divided into small departments like finance, account, and so on. Each department is responsible for certain a input and output. For example, the finance department deals with the input and output of finance. If a person has a query related to finance, then the person calls the finance department and not the others. Let's consider that there is no department and the organization handles all the queries, then the situation may be chaotic. People with a query will be confused about where to inquire. Similarly, to manage a big program, we divide the program into a small function, which performs a specific task.

Structure

- What is a function
- Defining a Python function
- Function with positional arguments
- Function with the argument and return value
- Function with default argument
- Function with variable-length arguments
- Function with keyworded arguments
- Argument pass by reference or value

- Scope
- Memory management
- Scope of variables

Objective

In this chapter, you will learn about the user-defined function, what is a function, the significance of making a function, different types of passing arguments of a function. After the function, different types of scope will be discussed. The memory management section presented the concept of storing the variables and value in the memory. In the end, you will learn about the scope of a variable and a global variable.

What is function?

So far, we have seen the Python built-in functions like `range()`, `input`, and `len()`, and many more. In this chapter, we will learn about the user-defined function. The function is used to reduce redundancy. Like a big organization is divided into small departments, we will divide a big code into small functions, which will do a specific task. We always try to make a generalized function so that the function can be used multiple times. As we go further, you will get to know the advantages of function.

Defining a Python function

The rules to define a function are as follows:

- Use the `def` keyword followed by the function name with parentheses ()
- Any arguments to the function must be placed within these parentheses
- The code block must start with a :
- The code with the function must be indented

See the following syntax to define a function:

```
def function_name(arguments):
  code
  return value
```

With the help of following syntax, we can call the above function:

```
function_name()
```

Calling a Python function is like calling the built-in function. To call the function, use the name of the function followed by a set of parentheses. A function can be called with the same or different arguments. You can also execute a function by calling it from another function. So, it can be said that function reduces the size of the code, as well as reduce redundancy. Let us see a function that just prints a statement:

```
def hello():
  print ("Hello World")
hello()
```

The *Figure 7.1* showcases the output:

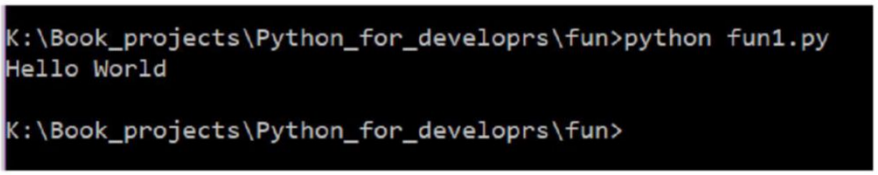

Figure 7.1

Function with positional arguments

As you have seen, a built-in function, like the `len()` function, you pass the sequence, and the function returns its length. You can also provide arguments to the function as showcased below:

```
def fun1(a,b):
  c = a+b
  print (a)
  print ("Result is ",c )

fun1(10,20)
```

Figure 7.2 is showcases the output of the program:

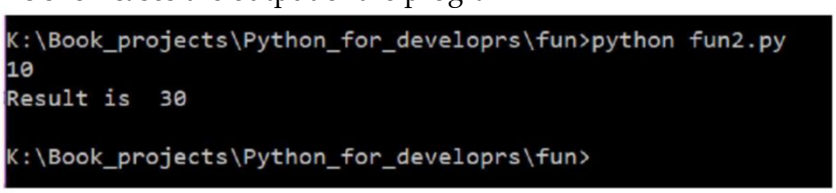

Figure 7.2

In the preceding program, two arguments have been passed by calling the Python function. The result c has been calculated. You can also provide an argument like a `key & word` form, which allows you to place them out of the order because the Python interpreter can use the keywords provided to match the values with the parameters:

```
def fun1(a,b):
  c = a+b
```

```
  print (a)
  print ("Result is ",c )


fun1(b=10,a=20)
```

The output of preceding program is showcased in *Figure 7.3*:

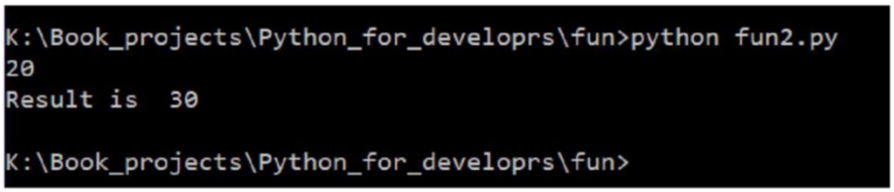

Figure 7.3

Let us consider a case where an argument is not given, then what happens? The next example, showcased in *Figure 7.4*, will provide clarity:

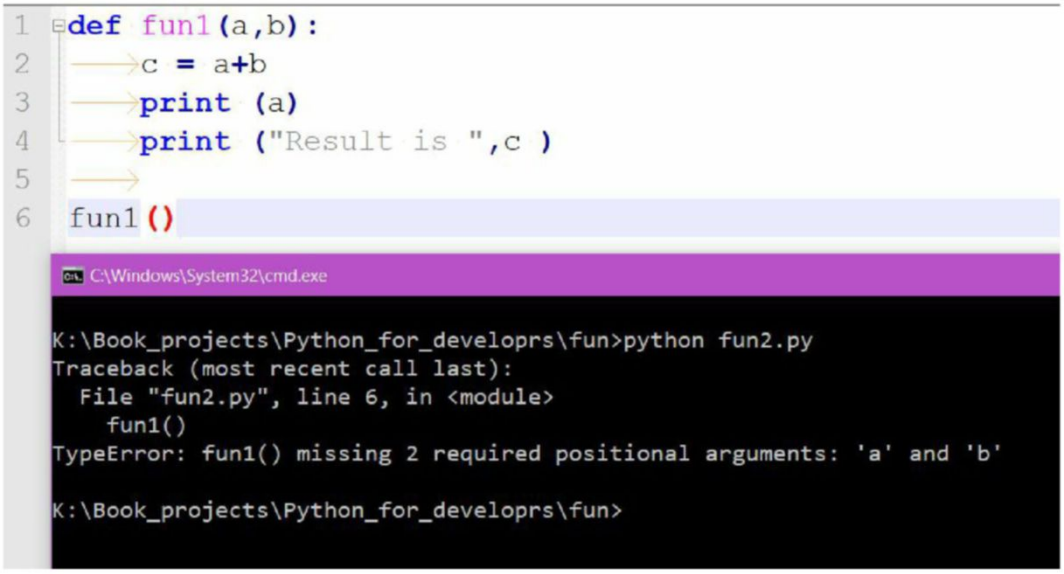

Figure 7.4

The error says that while calling, two arguments have been missed.

Function with the arguments and return value

A function with a return value is like a small department, where some material goes in, and newly processed material comes out.

Let us discuss this in the next example:

```
def fun1(a,b):
  c = a+b
  return c

result = 20 +fun1(1,12)
print (result)
```

The output is showcased in the following screenshot:

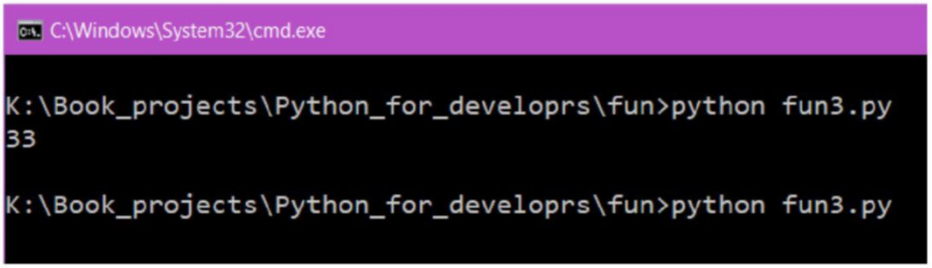

Figure 7.5

As the function returns c, which is an integer, that is the value of the function.

Function with default argument

We can also define the default argument. We can set the default value in the default argument. If we don't specify the argument while calling, then the function takes the default value. See the following example:

```
def fun1(name,skill, age=30):
  print (name,"\t", skill, age)
print ("Name  \t skill age")
fun1("Mohit","python", 32)
fun1("Bhaskar", "java")
```

See the *Figure 7.6* for the output:

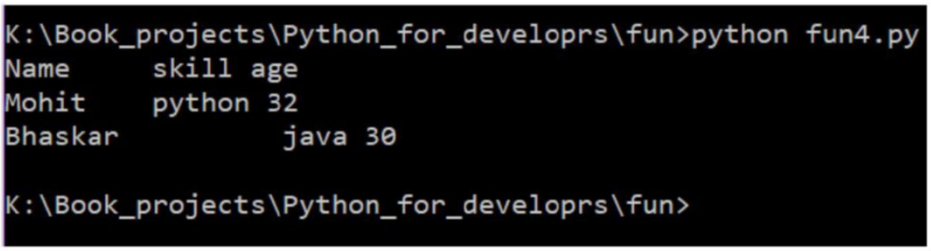

Figure 7.6

Make sure that the default argument always comes after the positional arguments.

The value of the default argument is set when we define the function.

If two default arguments are present, and you want to pass a value to the second default argument, then you can especially pass the value to the desired argument as showcased in the following the program.

```
def fun1(name,mark=60, age=30):
  print (name,"\t", mark, age)
print ("Name  \t skill age")
fun1("Bhaskar", age=32)
```

In the preceding program, we are passing a value to the age argument, not to the mark argument. The mark argument takes the default argument.

Function with variable-length arguments

With the help of a variable-length argument, we can give a different number of arguments.

See the following example:

```
def addition(a,*args):
  print (type(args))
  c = a+sum(args)
  return c


print (addition(10))
print (addition(10,50,12,20,40))
print (addition(10,20))
```

The output is showcased in the following screenshot:

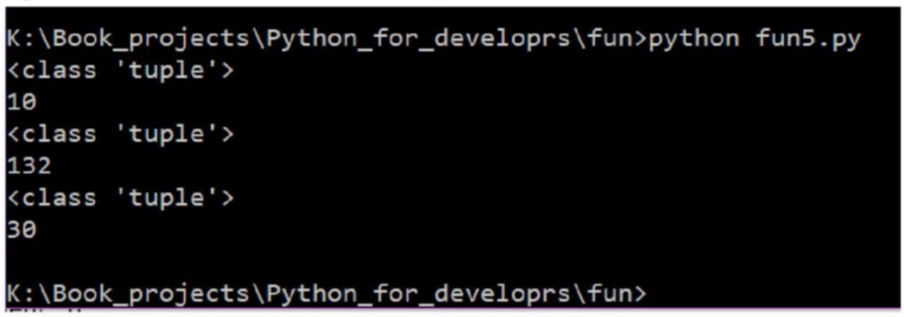

Figure 7.7

From the output we can conclude that the first argument is handled by argument a, and `*args` handles the rest of the arguments. The `args` takes the arguments as tuple.

Function with keyworded arguments

Let us consider a situation, where you want to pass the argument as the `key=value` format. See the following example:

```
def fun1(**kwargs):
  print (type(kwargs))
  print (kwargs)


fun1(name= "XYZ", age = 30, skill= "Quantum")
```

See the following *Figure 7.8* for the output:

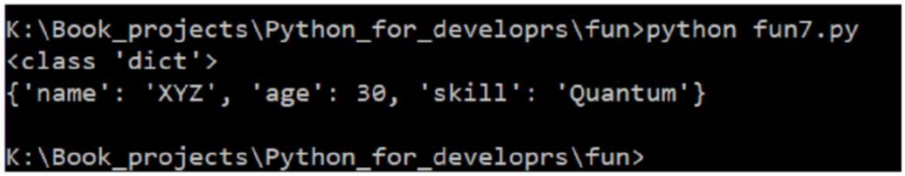

Figure 7.8

The `kwargs` uses the dictionary to take the keyworded arguments.

Let us look at one program, which contains all types of arguments:

```
def fun1(arg, age=30, *args, **kwargs):
  print (arg, age)
  print (sum(args))
  print (kwargs)
  print ("**************************")


fun1("Mohit", 31,1,2,3,4,5, skill= "Python", emp= 1234)
fun1("ravender",skill= "Python", emp= 1234)
```

The output is showcased in the following screenshot:

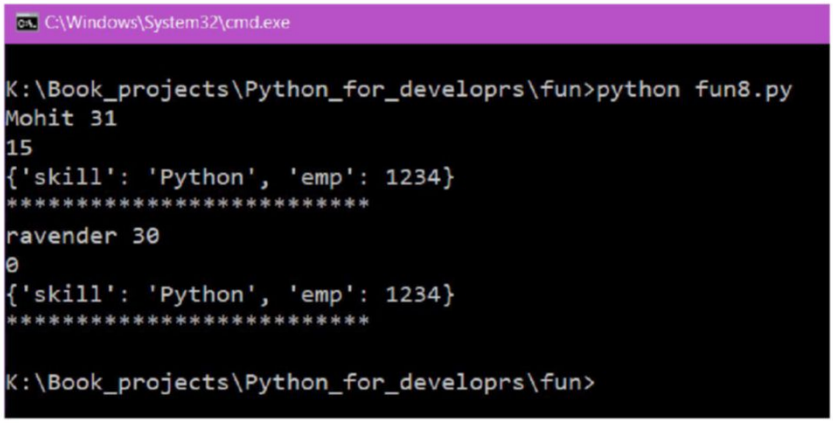

Figure 7.9

The preceding example shows the order of the arguments.

The order of the arguments: Positional ⌛ default⌛ Variable length ⌛ Keyworded.

If we change the order of arguments, it can cause an error.

Argument pass by reference or value

Call by reference means passing the address of a variable, where the actual value is stored. All the arguments in the Python language are passed by reference. The called function uses the value stored in the given address. Any changes to it, will affect the source variable.

For example:

```
def fun1(list1):
  print ("Inside list", list1)
  print ("Inside address", id(list1))
  list1.append(45)
  print ("inside after appending ", list1)

list2 = [10,80,90]
fun1(list2)
print ("Address outside", id(list2))
print ("After calling ", list2)
```

See *Figure 7.10* for the output of the preceding program:

```
K:\Book_projects\Python_for_developrs\fun>python fun9.py
Inside list [10, 80, 90]
Inside address 2115918324360
inside after appending  [10, 80, 90, 45]
Address outside 2115918324360
After calling  [10, 80, 90, 45]

K:\Book_projects\Python_for_developrs\fun>
```

Figure 7.10

The list extends inside the function, but the change is also reflected in the calling function.

Scope

Scope refers to a *part of the program* where a collection of identifiers are visible. The scope is just the range in which a variable can be modified or change.

The identifier must be defined using `id = ...`

In the previous chapters, we have examined variables, assignment operators. But now we have to explore a little more.

For instance,

```
k = 9
```

You would say k equals to 9, but it would be careless to say that. The meaning of the preceding expression is that k is a reference that points to an object that contains 9 inside it. In simple words, references have names, and objects are indicated to by references.

Types of scope

There are four of types of scope:

- Local scope
- Enclosing scope
- Global scope
- Built-in scope

Let us understand by looking at an example. See the following *Figure 7.11*:

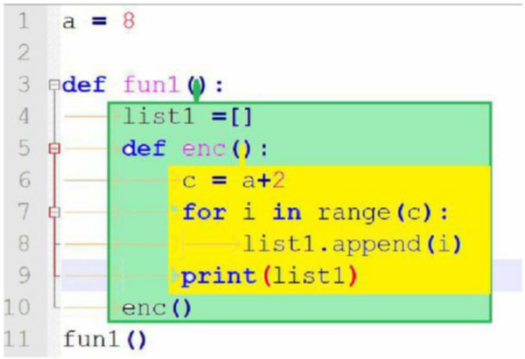

Figure 7.11

In the preceding figure, you can see three colored regions. They are white, green, and yellow. Let us learn all the scopes.

Local scope

Let us consider that the interpreter is executing the `enc()` function at line number 5. At that moment, the yellow region is a local scope for the function, `enc()`. The variable c is defined in the local scope. When the interpreter was at line number 3, the green region was the local scope for the `fun1()`.

Enclosing scope

When the interpreter is at line number 5, at that moment the green region is the enclosing scope for the `enc()` function. If any variable is not found in the local scope, then the interpreter checks the enclosing scope. The list1 variable is at the enclosing scope for the `enc()` function.

Global scope

For the `fun1()` and `enc()` functions, the white region acts as a global scope when the interpreter is executing the `enc()`function, then it examines the value of the variable a. The interpreter first checks the local scope, then the enclosing `scope`. If the variable is not found, then the interpreter checks the global scope. The variable a is defined in the global scope.

Built-in scope

If anything is not found in the local, enclosing, and global scope, then the interpreter checks the built-in scope. The global scope is limited to the program of the file. The `range()` function is defined in the built-in scope of the program.

Memory management

The interpreter stores the variable and data inside the RAM of the computer. The interpreter divides the RAM into two parts - **runtime stack and heap**. The runtime stack is the stack of the Activation Records. The Activation Record is a "portion of the memory", which contains all the information that is mandatory to keep a track of a "function call". When a function is called, the interpreter pushes the activation record of the function onto the run-time stack. When a function returns something, its corresponding activation record is popped from the run-time stack. The runtime stack stores the variables and the heap stores the values of the variables. The run-time stack stores only (variables) references pointed to corresponding values (objects) in the heap, as showcased in the following figure:

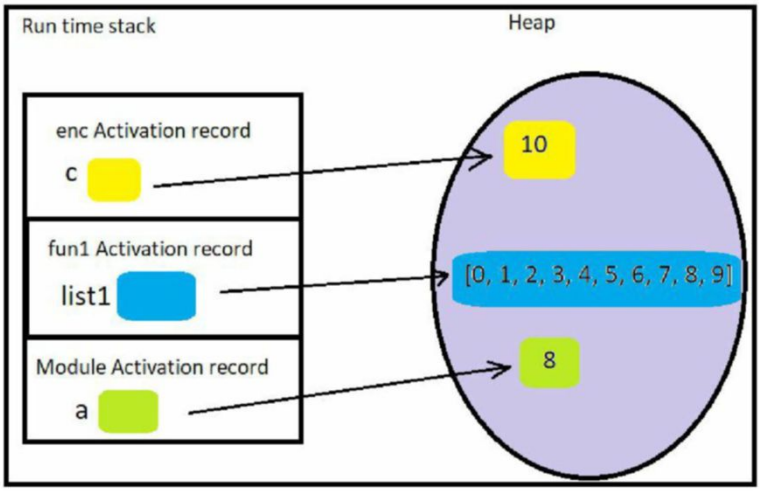

Figure 7.12

The *Figure 7.12* showcases the execution of the line number 5 of the program illustrated in the *Figure 7.11*. When the interpreter executes the line number 3, the activation record of the fun1 function is pushed to the top of the run-time stack.

When we use *c=10*, the assignment operator binds the variable c to the memory address, which contains 10. When we use the statement `del c`, the interpreter deletes only the binding and not the value 10. If any variable is not referring the value, then the garbage collector removes the value 10 from the heap memory.

Scope of variables

You have seen the scope, and the variables defined in the different scopes. In this section, we will learn the scope of variables in more detail. According to scope, there are two types of variables:

- Local variables
- Global variables

The local variable is defined inside the function. Local variables are only accessible within their local scope.

The global variable is defined outside the Python function. Global variables are accessible throughout the program. See the following program:

```
def fun():
   print ("Inside the function", K )
K= 20
fun()
print ("Outside K",K)
```

See the following screenshot for the output:

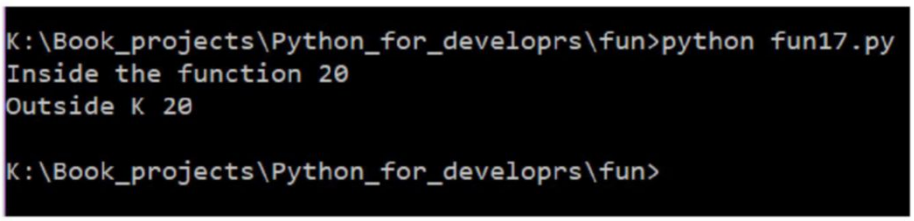

Figure 7.13

Variable `K` is a global variable. Its value remains the same outside the function and inside the function. But what if you reassign it inside the function? Let us discuss this with an example:

```
def fun():
   K = 10
   print ("Inside the function", K )
K= 20
fun()
print ("Outside K",K)
```

See the following screenshot for the output:

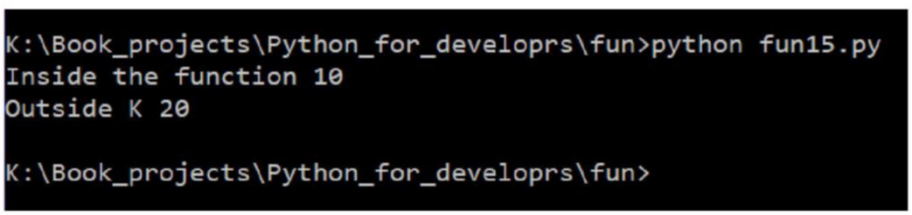

Figure 7.14

If you reassign a global variable inside the function, it does not reflect outside the Python function.

If a local variable is accessed outside the function let us discuss this in the next example:

```
def fun():
   K = 10
   print ("Inside the function", K )
fun()
print ("Outside K",K)
```

See the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\fun>python fun21.py
Inside the function 10
Traceback (most recent call last):
  File "fun21.py", line 5, in <module>
    print ("Outside K",K)
NameError: name 'K' is not defined

K:\Book_projects\Python_for_developrs\fun>
```

Figure 7.15

If you access a local variable from outside the function, it gives an error. Let us consider a situation where you feel that the change inside the function should reflect outside the Python function.

Let us try to change the global variable inside a function:

```
def fun():
   K = K+30
   print ("Inside the function", K )
fun()
K =20
print ("Outside K",K)
```

A local variable, K, is being created using the global variable K. Let us see the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\fun>python fun20.py
Traceback (most recent call last):
  File "fun20.py", line 4, in <module>
    fun()
  File "fun20.py", line 2, in fun
    K = K+30
UnboundLocalError: local variable 'K' referenced before assignment

K:\Book_projects\Python_for_developrs\fun>
```

Figure 7.16

If you are using an assignment operator, then the variable K is taken as a local variable, that is why the error occurred. Let us consider a tough situation, where you want to change the global variable inside the function.

In this situation, you would explicitly define the `global` keyword, as showcased in the following example:

```
def fun():
   global K
   K = K+30
   print ("Inside the function", K )
K =20
fun()
print ("Outside K",K)
```

The output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\fun>python fun22.py
Inside the function 50
Outside K 50

K:\Book_projects\Python_for_developrs\fun>
```

Figure 7.17

In the preceding example, after calling, the function value of K changed because inside the function, K is defined as a global variable. The statement global K tells Python that K is a global variable. Python stops searching the local namespace for the variable.

Conclusion

In this chapter, we have learned about the function, to organize a big program and reduce the redundancy we use functions. In this chapter, we have learned about the definition of function, passing arguments, return value, and calling the function. Python passes the argument by using the references. You have learned the different types of scopes. In the end, you have seen the local and global variables. In the next chapter, we will learn about modules and packages.

Questions

1. What would be the output of the following program?

```
def fun1(val, list_new=[]):
    list_new.append(val)
    return list_new
list1 = fun1(10)
list2 = fun1(123)
list3 = fun1('a')
print ("list1 = %s" % list1)
print ("list2 = %s" % list2)
print ("list3 = %s" % list3)
```

2. What would be the output of the following program?

```
def fun1(val, list_new=[]):
    list_new.append(val)
    return list_new
list1 = fun1(10)
list2 = fun1(123,[])
list3 = fun1('a')
print ("list1 = %s" % list1)
print ("list2 = %s" % list2)
print ("list3 = %s" % list3)
```

3. What is the return type of a variable-length argument?
4. If a function does not contain anything in the return statement, then what is the return value of the function?

Chapter 8
Module

In the previous chapter, you have seen the functions, which are used to remove to the redundancy of code. But if the function is present on the different file, then we can import the file as a module and use the function.

Let us consider a situation where we develop a project. It is not easy to write everything in one file. The project is divided into a team of developers. After getting the task, each developer writes the code. For example, a developer named "Alice" writes the database related stuff. The second developer, named Mohit, writes the data extraction related stuff. If developer Mohit needs the database code written by Alice, then Mohit will use Alice's code by importing it as a module. Of course, a module must be written in such a way that other people can use it. When we have a lot of files in one folder, we can make that folder as a package. A package is the collection of modules and the `__init__.py` file.

Structure

- Module
- The import statement
- Locating Python modules
- Compiled Python files
- Python package

Objective

In this chapter, you will learn about the modules and packages. Modules help in organizing the project. You will learn how to import a module and the different types of statements to import modules, how a module is located by the interpreter. You will study how to make packages using the `__init__.py` file.

Module

A module consists of a Python source file. A Python module can comprise of functions, classes, and statements. Generally, any python program can act as a module. But when we actually make modules to organize the project, we certainly follow some rules. We will learn the rules as we proceed in the chapter.

Let us take a look at an example:

```
def sum1(x,y):
  z = x+y
  return z

def mul1(n,m):
  p = n*m
  return p
```

Save the preceding code as `module1.py`, consider it as a module.

The import statement

The import has the following syntax:

```
import module1, module2, module
```

The import statement should be at the beginning of the code. The `import` keyword is followed by one or more Python module, separated by commas. When the interpreter sees the import statement, it imports the module, if available. For example, let's see how to import the module "`module1.py`":

```
import module1
a =10
b =90
print ( module1.sum1(a,b) )
```

The interpreter executes module body immediately. A module is loaded only once. To access the attribute of the module, use the module object as a prefix. In the preceding example, module1 is the module name, and `sum1()` is a function defined in the module. See the output in *Figure 8.1:*

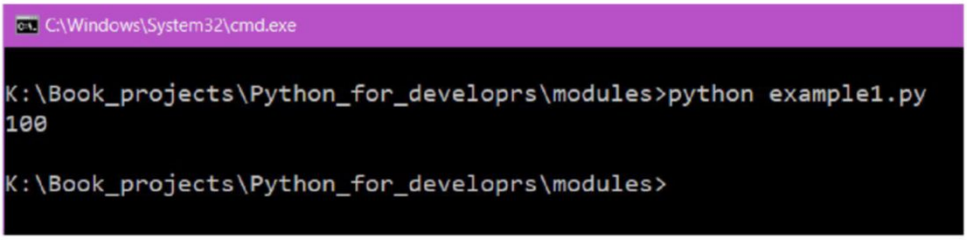

Figure 8.1

The `module1` name seems lengthy; you can shorten the name using the "`as`" keyword, as showcased in the following example:

```
import module1 as m1
a =10
b =90
print ( m1.sum1(a,b) )
```

See the output in the following screenshot:

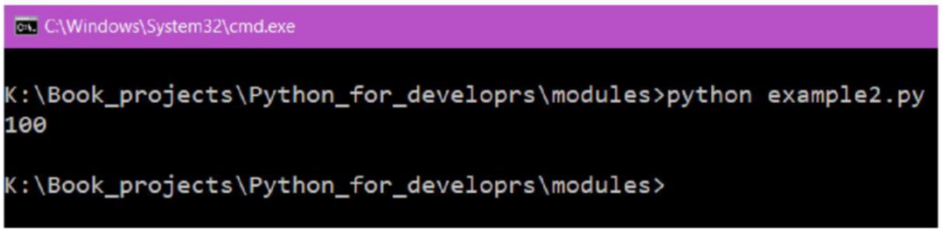

Figure 8.2

The from statement

The `from` statement allows you to import specific attributes from a module into the current program. The syntax of from import statement is written below.

```
from module_name import name1,  name2
```

Let us understand this with the help of an example:

```
from module1 import sum1
a =10
b =90
print ( sum1(a,b) )
```

In the preceding code, we did not use the module name with the `sum1()` function.

Let us consider that if two modules contain a function with the same name, but the functionality is different.

Let's see the `module2.py` module:

```
def sum1(k,l):
  t = k-l
  return t
```

Now, we are importing both the modules with the from statement. The example code is showcased below:

```
from module1 import sum1
from module2 import sum1
a =10
b =90
print ( sum1(a,b) )
```

See the output in the following screenshot:

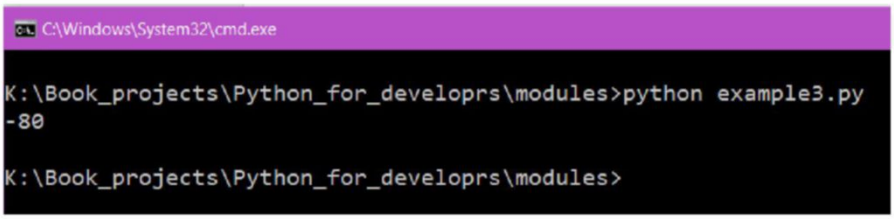

Figure 8.3

The `sum1` of module1 is overwritten by the sum1 of `module2`. We are getting the output from the `sum1()` function defined in the module2. Let us consider that we change the sequence of the statements. Let us see the following code:

```
from module2 import sum1
from module1 import sum1
a =10
b =90
print ( sum1(a,b) )
```

See the following screenshot for the output:

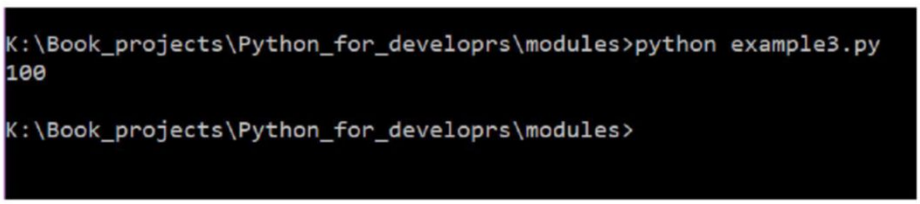

Figure 8.4

Now we get a different output. We are getting the output from the `sum1()` function defined in the `module1`. So always pay attention to the order of importing the modules.

You can import all names from a module, using the following statement.

```
from module_name import *
```

But you can restrict the access to specific functions or variables. See the following example, consider the `module2.py` as showcased below.

```
__all__ = ["sum1",'a']
a = 10
b = 20
def sum1(a,b):
  c = a+b
  return c

def mul1(a,b):
  c = a*b
  return c

def sub(a,b):
  c = a-b
  return c
```

The "`__all__` " is a magic variable, which refers to a list that contains the items to be available for the others. By specifying `__all__` we can put the restriction on module's function or variables. Let us see the following testcases.

```
>>> from module2 import *
>>> sum1(10,20)
30
>>> a
10
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> mul1(10,20)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mul1' is not defined
>>>
```

You can see that only the items present in the "`__all__`" list are allowed to be used.

Locating Python modules

You now understand the idea of the module. There are some built-in modules like JSON, pickle, and some are user-made for their project. How does the Python interpreter finds the built-in module, as well as user-made modules? After encountering the import statement, the interpreter searches the module in the following sequences:

- The current directory, which contains the running script
- PYTHONPATH
- The installation-dependent default

The interpreter first sees the current directory of the running script. If the module is not found in the current directory, then interpreter checks the `PYTHONPATH`.

Let us see the `PYTHONPATH` example.

In Windows, `PYTHONPATH` can be assigned permanently or temporarily. You can permanently add in the system variable or the user variable. You can temporarily add in the command prompt. Let us suppose that we made a program `mod1.py`, it contains the following statement:

```
import module1 as m1
print (m1.sum1(10,9))
```

The module1 is located at `K:\Book_projects\Python_for_developrs\modules\m`.

When we try to execute the code, we see the error showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\modules>python mod1.py
Traceback (most recent call last):
  File "mod1.py", line 1, in <module>
    import module1
ModuleNotFoundError: No module named 'module1'
```

Figure 8.5

This is because the Python interpreter does not know the path of `module1`. Let us set the path of the module.

Use the following command to set the `PYTHONPATH`:

```
set PYTHONPATH=K:\Book_projects\Python_for_developrs\modules\m
```

See the following screenshot:

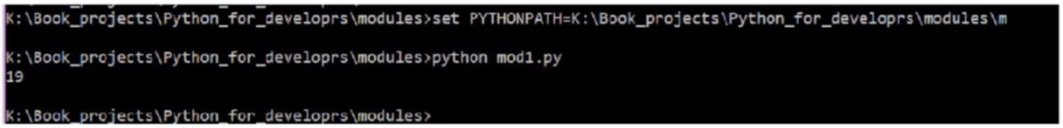

Figure 8.6

The `PYTHONPATH` is working.

The installation-dependent is default.

If you want to know the installation-dependent path, you can check the `sys.path` variable in the `sys` module. See the following screenshot:

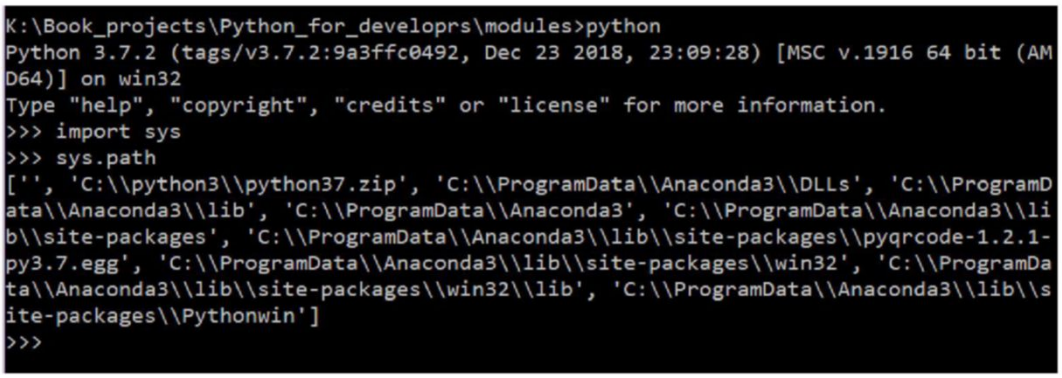

Figure 8.7

The `sys.path` returns a list of paths.

Let us suppose that you are importing a built-in module and you don't know the path, with the help of " `__path__`" you can find the module path. In the following code, we are obtaining the path of the `json` module:

```
>>> import json
>>> json.__path__
['C:\\ProgramData\\Anaconda3\\lib\\json']
>>>
```

Compiled Python files

If a file called `module1.pyc` exists in the directory where `module1.py` is found (as shown in the following screenshot), then it is assumed to contain an already-"byte-compiled" version of the `module1` module. Usually, you do not need to do anything to create the `module1.pyc` file. Whenever `module1.py` is successfully compiled, an attempt is made to write the compiled version to `module1.pyc`. It is not an error if this attempt fails. If, for any reason, the file is not composed entirely, then the resulting `module1.pyc` file will be recognized as invalid and thus will be ignored later. The content of the `module1.pyc` file is platform-independent so that machines of different architectures can share a Python module directory.

In Python 3, compiled files are created under the directory `__pycache__`:

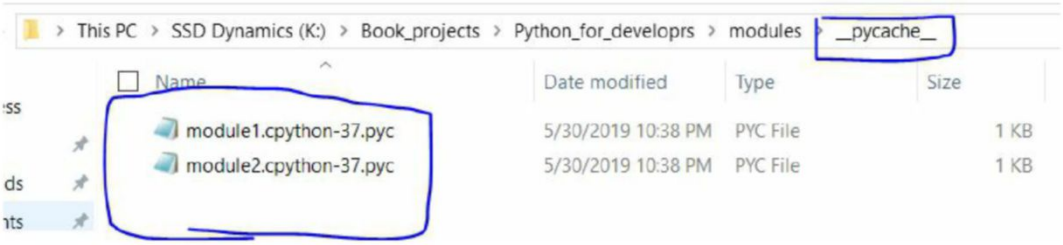

Figure 8.8

dir()

The built-in function `dir()` is used to find out the content of modules. It returns a list, which contains functions, variables, class, and so on; whatever defined in the module. An example of `dir()` is showcased below:

```
import module1
print (dir(module1))
```

The output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\modules>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] o
n win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import module1
>>> dir(module1)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package
__', '__spec__', 'mul1', 'sum1']
>>>
>>>
```

Figure 8.9

You can see the output returns a list, which contains the `mul1` and `sum1` function.

The __name__ statement

Consider a Let us suppose that we make a program for our use. The program contains two functions, and you call the function within file. Let us see the program:

```
def sum1(a,b):
  c = a+b
  return c

def mul1(a,b):
  c = a*b
  return c
print (sum1(10,30))
```

Let us see the output in the following screenshot:

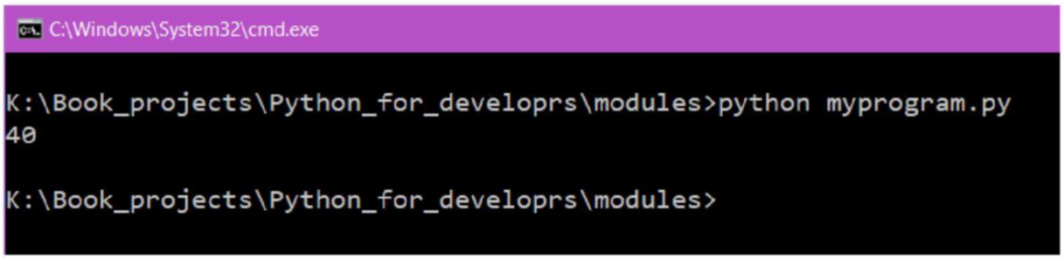

Figure 8.10

We are getting the desired output. Now let us suppose that another person uses our program as module. Let us see the following example:

```
import myprogram as mg
print (mg.sum1(100,200))
```

Let us see the output in the following screenshot:

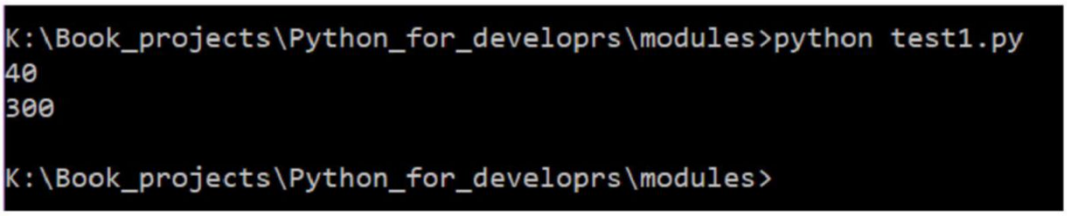

Figure 8.11

We are getting our expected result, but we are also getting the result of the function call that is defined in the module. If we remove the function call from the module, then the owner of the module will not get the result. To tackle this situation, we use the `__name__` keyword. Let's see the significance of the `__name__` keyword. Let us add the `__name__` keyword in the module, as showcased below:

```
def sum1(a,b):
  c = a+b
  return c
def mul1(a,b):
  c = a*b
  return c
print (__name__)
print (sum1(10,30))
```

Let us see the output:

```
K:\Book_projects\Python_for_developrs\modules>python myprogram.py
__main__
40
```

Figure 8.12

We are getting the value of `__name__`, that is `__main__`. The `__main__` is a string here. Let us run the `test1.py` code again, and check the output. See the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\modules>python test1.py
myprogram
40
300

K:\Book_projects\Python_for_developrs\modules>
```

Figure 8.13

We are getting the value of `__name__` as myprogram, which is the name of the importing module. So when we run the module, the `__name__` keyword returns the `__main__` string. If the module is imported by the other program, then the `__name__` keyword returns the name of the module. With the help of the following line, we can avoid the undesireable result.

```
If __name__ == "__main__".
```

Let us see the full code of the module:

```
def sum1(a,b):
  c = a+b
  return c

def mul1(a,b):
  c = a*b
  return c

if __name__ == "__main__":
  print (sum1(10,30))
```

Let us check the output of the module run and the `test1.py` code. See the following screenshot:

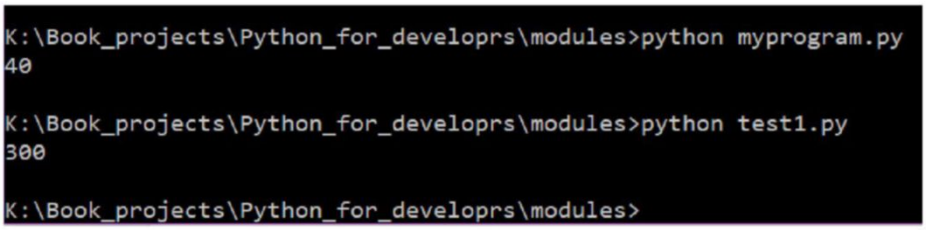

Figure 8.14

We do not get an undesirable result now.

Python package

Let us consider that, in a project, there is a need for database interaction. The developer "A" writes the code, `db_mysql.py`, to handle the MySQL with Python. After a day, the project needs MongoDB database interaction. The second developer “B”, writes his/her own file, `db_mongo.py`, instead of writing `db_mysql.py`. After a few days, the project requires to use a Redis database. The developer "C" writes a new file, `db_redis.py`, to interact with Redis.

Now the project has three database files. Instead of combining all the files, we can make a package, which acts as a module.

A package is a Python module that contains other Python modules. Generally, a Python package is a directory, which includes modules and a file `__init__.py`. Due to `__init__.py` file, the directory behaves like a module. The `__init__.py` gets executed when the package is imported.

Let us discuss this with an example

Let us consider the two python files available in the `K:\Book_projects\Python_for_developrs\modules\My_pack` directory. The files have the following lines of source code:

```
def IBM():
   print ("We make IT Happen")
```

Similarly, we have another file that has a different function:

```
def INTEL():
   print ("Sponsors of Tomorrow" )
```

Now, make an empty `__init__.py` file in the same directory.

Outside the directory, create a new program, to import the package, as showcased below:

```
from My_pack import ibm
from My_pack.intel import INTEL
ibm.IBM()
INTEL()
```

We used two different approaches to import the modules of packages; you can use any one.

To make all of your functions available when you've imported `My_pack`, you need to put explicit import statements in the `__init__.py`, as follows:

```
from .ibm import IBM
```

Now make `file2.py`, which will use these packages:

```
from My_pack import IBM
IBM()
```

Run the above program. The following screenshot showcases the output:

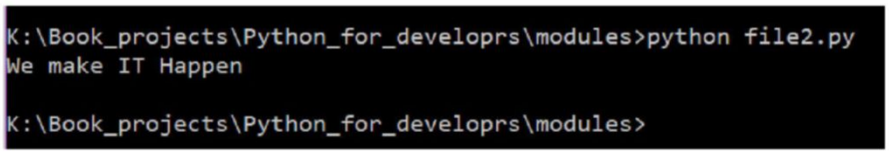

Figure 8.15

You can see that there is no need to import the module. It seems like the functions have directly linked with the package.

Importing the modules from different path

Let us consider your modules are not present in the current directory, the built-in path, and the `PYTHONPATH`. To import the module from a different path, we have two approaches:

- Adding the path to `sys.path`
- Using the `importlib` library

Let us understand the scenario.

The module1.py module is present at the following address:

`K:\Book_projects\Python_for_developrs\modules\m2`

A program, `mod1.py`, present at `K:\Book_projects\Python_for_developrs\modules`, tries to use the `module1.py` module. As the interpreter could not find the module1, it gives the error "`No module named`", as showcased in the following screenshot:

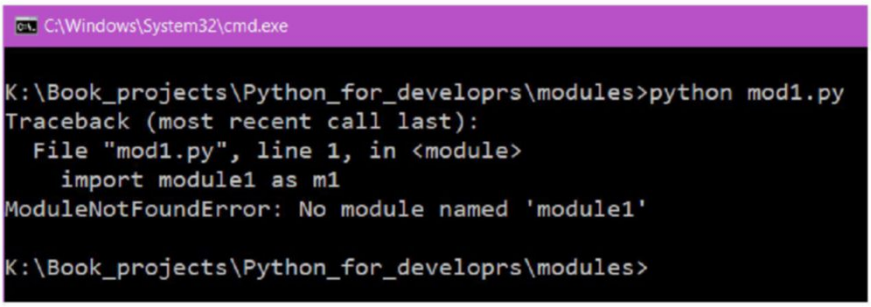

Figure 8.16

As expected, we are getting an error. We know that sys.path returns a list of the installation paths.

Use the following code to add your new path to the sys.path:

```
import sys
sys.path.append("K:\\Book_projects\\Python_for_developrs\\modules\\m2")
import module1 as m1
print (m1.sum1(10,9))
```

The output is showcased in the following screenshot:

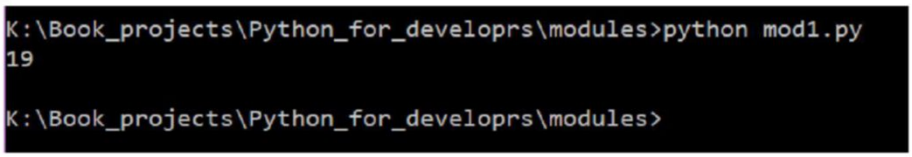

Figure 8.17

Let us see how to use "importlib" to load the module.

See the following code. The code is only for Python 3.5 or a higher version:

```
from importlib.machinery import SourceFileLoader
m1 = SourceFileLoader("mohit", 'K:\\Book_projects\\Python_for_developrs\\
modules\\m2\\module1.py').load_module()
print (m1)
print (type(m1))
print (m1.sum1(1,2))
```

See the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\modules>python mod2.py
<module 'mohit' from 'K:\\Book_projects\\Python_for_developrs\\modules\\m2\\module1.py'>
<class 'module'>
3

K:\Book_projects\Python_for_developrs\modules>
```

Figure 8.18

The `SourceFileLoader` function loaded the module and assigned to the m1 variable. Now, the m1 variable acts as the module1 module. The syntax of `SourceFileLoader` is `machinery.SourceFileLoader(name, pathname)`. The name can be any string, and the pathname is the full path of the module, as showcased in the preceding example.

Conclusion

In this chapter, we have learned about organizing the big program. In the module section, we learned that the module is a file. It may be self-executed or executed by another file, by importing as a module. The module removes redundancy. After modules, we have learned about packages. A package is a directory, which contains modules and the `__init__.py` file. Sometimes, different programmers want to work on different files instead of a single file. In that case, we make packages. In the next chapter, we will learn about exception handling.

Questions

1. How does a Python interpreter locate a module?
2. Why do we make packages?
3. What is the return value of __name__ when we execute the module?
4. How to find the path of a module?

Chapter 9
Exception Handling

You may have seen that, sometimes, just adding one line in the existing code causes execution fails. Due to that, the whole program suffers. In order to avoid errors, it is always advisable to handle the exception in your code.

Structure

- Exception
- Try statement with an except clause
- Multiple exception block
- Else in exception
- finally statement
- Program find its exception type
- Raising an exception
- Advance section

Objective

In this chapter, we will learn about how to handle the exception, what are the different statements associated with exceptional handling. We will also learn about

how errors can be avoided, by using exception handling. At the end of the chapter, we will learn how to make our own exception.

Exception

When the interpreter encounters an error, it stops the running program and shows an error message, detailing the exception.

Let us see an elementary example:

```
>>> n = int(input("Enter a number "))
Enter a number ok
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ok'
>>>
```

You can intercept and manage exceptions by using the exceptional handling features of Python. Exception handling prevents the program from ending abruptly. You can have your program exit gracefully, instead of crashing awkwardly.

Try statement with an except clause

To handle the exception, the `try` and `except` blocks are used. A single try statement can have multiple except statements. Due to multiple exceptions, the program can throw an exception in the appropriate section. See the following example:

```
try:
  num = int(input("Enter the number "))
  res = 50/num
  print (res)
except:
  print ("Something is wrong")
```

See the output in the following screenshot:

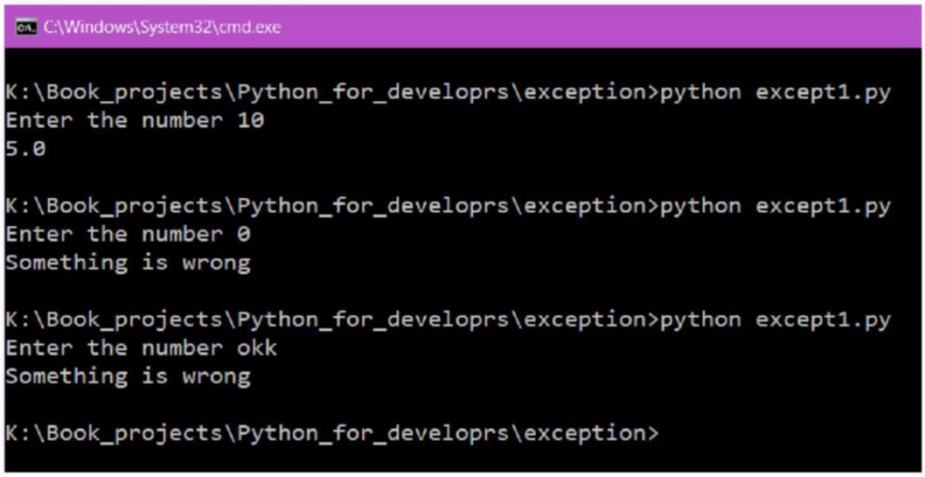

Figure 9.1

No error has been exhibited this time. When we pass a string, the text, "`Something is wrong`", will be exhibited. When we provided `0`, which is `int`, we still got the error. When we gave the `int`, except `0`, we did not get the error.

Multiple exception block

In order to handle different type of exceptions, we can use the multiple exception block. If any exception occurs, then the corresponding exception block handles the exception.

Let us discuss our first example:

```
try:
  num1 = int(input("Enter the number "))
  c = 50/num1
  print (c)
except ValueError :
  print ("Use integer only ")
except ZeroDivisionError:
  print ("Don't use Zero")
except :
  print ("Unknown Error")
```

We executed the code three times, as showcased in the following screenshot:

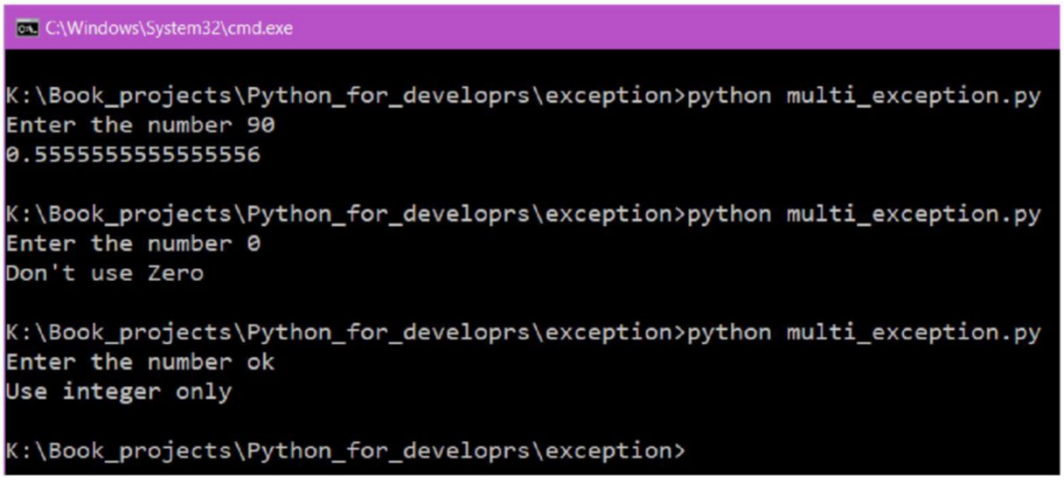

Figure 9.2

In the preceding example, you can understand that the interpreter throws an error in the corresponding exception block. If the corresponding exception block is not found, then the default exception block is used, if present.

else in exception

Let us consider that we want to run a piece of code. If the code in the try block gets executed successfully, then you can use the else block. The interpreter executes the else block if there is no error in the try block. Let's see the following example:

```
try:
  num1 = int(input("Enter the number "))
  c = 50/num1
  print (c)
except ValueError :
  print ("Use integer only ")
except ZeroDivisionError:
  print ("Don't use Zero")
except :
  print ("Unknown Error")
else :
  c = c +60
  print (c)
```

We have run the code three times, as showcased in the following figure. In the first execution, the `try` block got executed successfully. Consequently, the interpreter executed the `else` block. In the second and the third run, the interpreter throws the exception in the corresponding block, hence the `else` block was not executed:

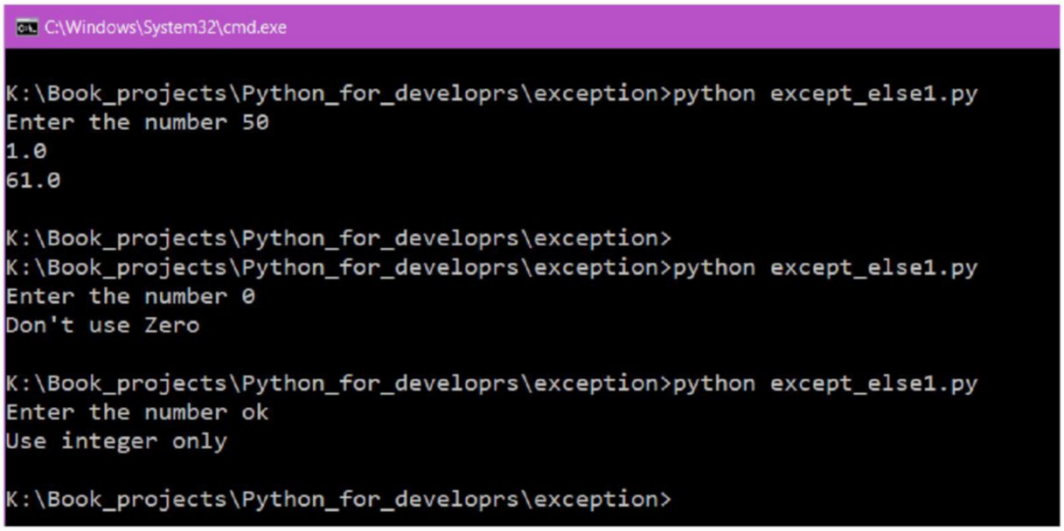

Figure 9.3

finally statement

Whatever code you write in the finally block, the interpreter always executes the code of the finally block. The finally block does not depend on the execution of the try or except block.

Let us discuss our first example:

```
try:
  a = int(input("Enter the value :"))
  c = 30/a
except:
  print ("Unknown error")
else:
  c=  c +a
  print (c )
finally:
  print ("Program exit")
```

See the output in the following screenshot:

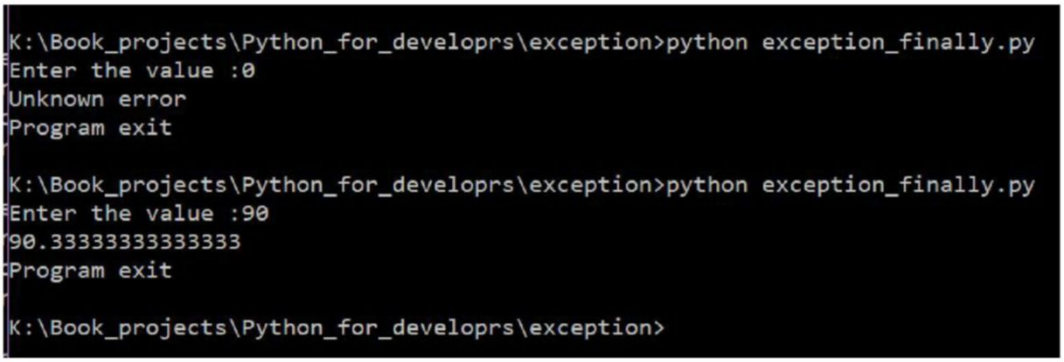

Figure 9.4

From the output, you can understand that the `finally` block will be run, whether an error is handled or not. Let us consider a scenario, where we have opened the database connection in the `try` block and closed the connection at the end of the try block. If any error occurs in the "try" block, then the database connection will not be closed and will remain open. In this situation, we place the closing syntax in the "finally" block. If the "try" statement reaches a break, continue or return statement, the "finally" clause will execute just prior to the break, continue or return statement's execution.

Program find its exception type

Writing all types of exceptions in a program is not possible. There are a lot of exception types, and it is a very tedious and mundane task to write all types of exception. See the following example:

```
try:
  num1 = int(input("Enter a number "))
  c = 50/num1
  print (c)
except Exception as  e :
  print (e, type(e))
```

The preceding code presented an excellent technique to find its exception type. It catches the exception as "e" and type(e) displays its exception type.

See the output in the following screenshot:

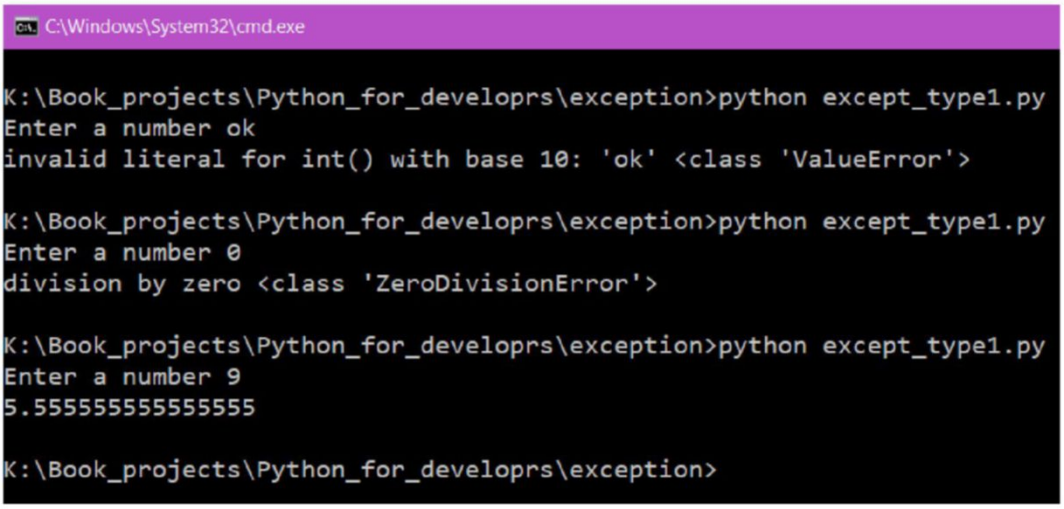

Figure 9.5

From the output, you can see that the code itself is telling the exception type.

Raising an exception

You can use the raise statement to explicitly trigger exceptions. Their general form is simple.

The following syntax could be used with raise:

```
raise <instance> # Raise instance of class
raise <class> # Make and raise instance of class
raise # Reraise the most recent exception
```

In the preceding examples, you have seen that entering a zero or a string causes errors. Let us consider a situation, where a user enters the number 2, logically there is no error, but the business logic says that if somebody enters 2, it should be considered an exception. See the following code, illustrating the business logic:

```
Val = int(input("Enter the number "))
try:
   if val ==2:
      raise IOError
   c = 50/val
except IOError:
```

```
   print (“an exception is raised”)
except Exception as e :
   print (e)
else:
   print (“value is not 2”,c)
```

If you enter a value equal to 2, then the if block will raise the exception by using the raise keyword. The `except` block will handle that exception.

The code is executed three times, as showcased in the following screenshot:

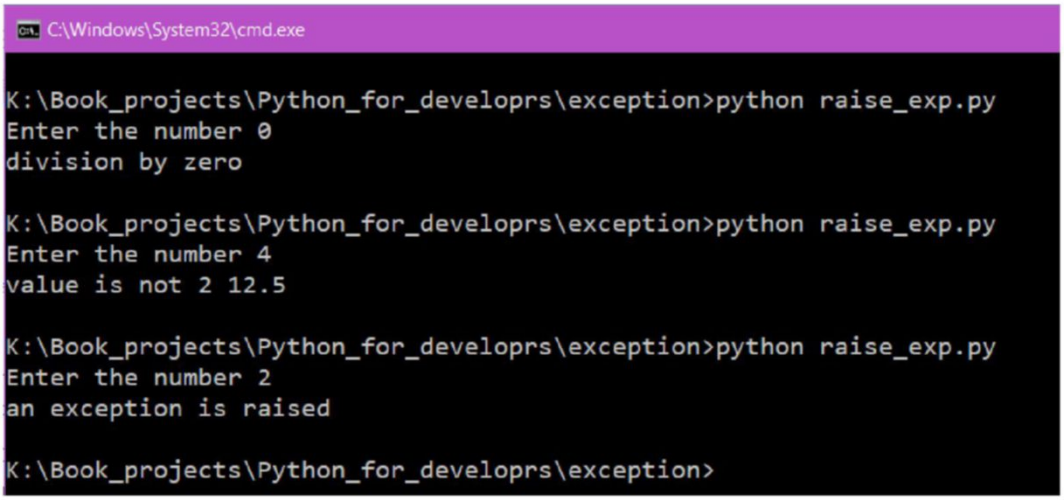

Figure 9.6

In the preceding example, if `val == 2`, then the exception, IOError, is raised. (2 may be an unlucky number for a user). And this exception is caught by the `except` block.

If you type `raise` in the Python shell, then:

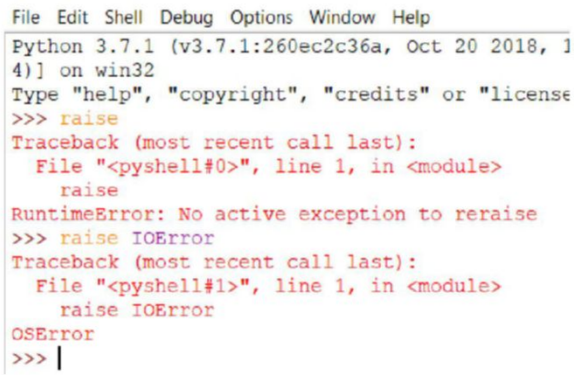

Figure 9.7

Advance section

In this section, we will cover the user-defined exceptions.

User-defined exceptions

Python also enables you to generate exceptions by establishing a new exception class. Exceptions should typically be obtained either directly, or indirectly, from the `Exception` class. See the following code:

```
class MyException (Exception):
  def __init__(self):
    pass
  def __str__(self):
    return "2 is not allowed"
try :
  a = int(input("Enter the value "))
  if a ==2:
    raise MyException
  c = 10/a
  print (c )
except MyException as e :
  print (e)
except Exception as e:
  print (e )
```

In the preceding example, a subclass, `MyException`, is made by deriving the `Exception` class. In this example, the default `__init__()` of Exception has been overridden. In the `__str__` method, a customized fixed message has been returned. In the `try` block, an exception, `MyException`, has been raised.

The *except* block caught the exception and displayed the message returned by the `__str__` method. Let us see the output in the following screenshot:

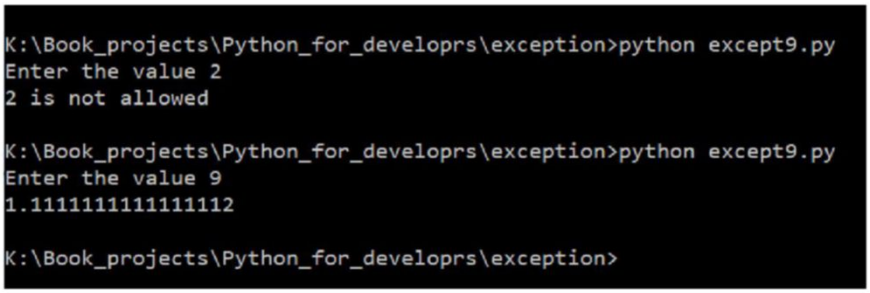

Figure 9.8

In the preceding example, the message is fixed. If you want to change the customized message with every new and try block, see the following example:

```
class MyException (Exception):
  def __init__(self,h):
    self.h = h
  def __str__(self):
    return "Not valid "+self.h


try :
  a = int(input("Enter the value "))
  if a ==2:
    raise MyException("Don't use 2")
  elif a == 13:
    raise MyException("Don't use 13")
  c = 10/a
  print (c )

except Exception as e:
  print (e )
```

In the preceding example, integer 2 and 13 are included in the exception list. In the `try` block, the messages, "`Don't use 13`" and "`Don't use 2`", have been passed to the `MyException` class. The `MyException` class returns the modified message, as showcased in the output.

See the execution in the following screenshot:

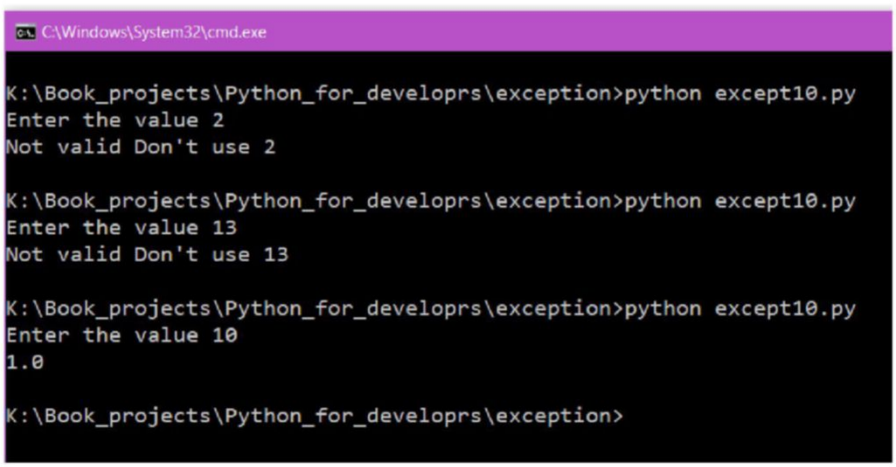

Figure 9.9

Now add following line at the end of the code:

```
print (MyException.__mro__)
```

See the output, after adding the preceding line:

```
K:\Book_projects\Python_for_developrs\exception>python except10.py
Enter the value 90
0.1111111111111111
(<class '__main__.MyException'>, <class 'Exception'>, <class 'BaseException'>, <class 'object'>)

K:\Book_projects\Python_for_developrs\exception>
```

Figure 9.10

In the preceding example, you can see the `class` order. The base class of the exception class is `BaseException`. To check all the types of exception and hierarchy, check the following link:

https://docs.python.org/3/library/exceptions.html# Exception

So far, we have learned about the `try` and `exception` statements. Let us suppose that we run a code of 100 lines, with exception handling. After getting an error, it may be challenging to find out, which line causing the error. Let us see the following example:

```
num1 = int(input("Enter the number "))
c = 10/num1
a = c+10
print (a)
```

Let us run the code and try to produce the error. See the output in the following screenshot:

```
Select C:\Windows\System32\cmd.exe

K:\Book_projects\Python_for_developrs\exception>python without_except.py
Enter the number 0
Traceback (most recent call last):
  File "without_except.py", line 2, in <module>
    c = 10/num1
ZeroDivisionError: division by zero

K:\Book_projects\Python_for_developrs\exception>
```

Figure 9.11

The program stops abruptly. Let us use the exception handling:

```
try:
```

```
  num1 = int(input("Enter the number "))
  c = 10/num1
  a = c+10
  print (a)
except Exception as e :
  print (e)
```

See the following output:

```
K:\Book_projects\Python_for_developrs\exception>python without_except.py
Enter the number 0
division by zero

K:\Book_projects\Python_for_developrs\exception>
```

Figure 9.12

Now the program is handling the exception. However, we are still not getting the line that is causing the error. The previous example was showing the error in line 2. See the following example, which handles the exception, as well as, shows the error line:

```
import traceback
try:
  num1 = int(input("Enter the number "))
  c = 10/num1
  a = c+10
  print (a)
except Exception as e :
  print (e, type(e))
  print (traceback.print_tb(e.__traceback__))
  #print (traceback.format_tb(e.__traceback__))
print ("finish")
```

The traceback returns the error details with the line number that is causing the error.

See the following output:

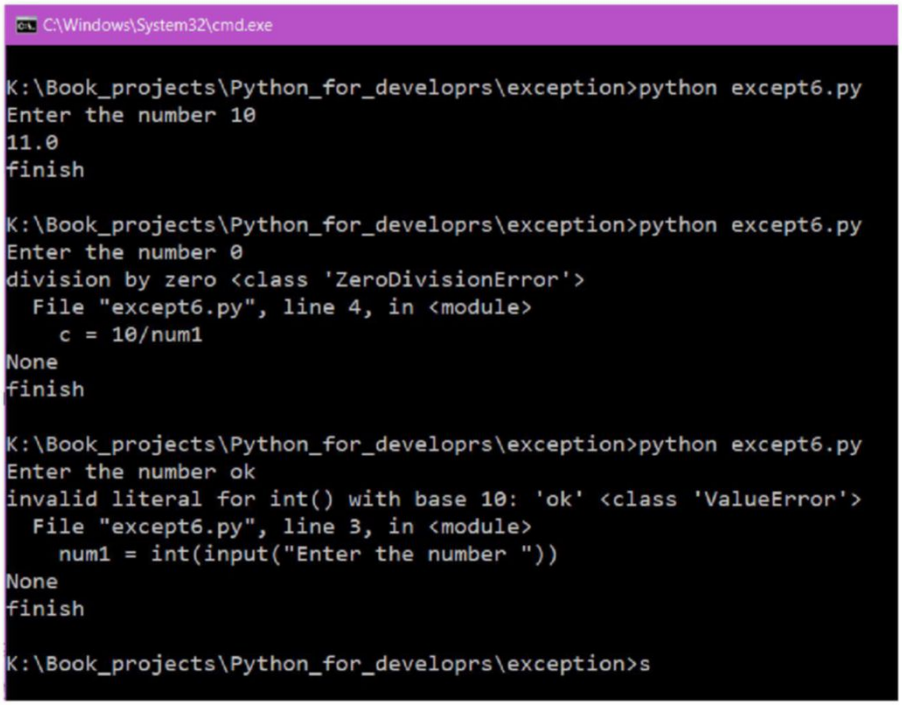

Figure 9.13

The figure clearly shows that the code is catching the exception, as well as, showing the real error.

Exercise

Use exception handling in the following program:

```
list1 = [1,2,0,3,4,9,0]
for each in list1:
  result = 90/each
  print (result)
```

Answer:

```
list1 = [1,2,0,3,4,9,0]
for each in list1:
  try:
    result = 90/each
    print (result)
```

```
except Exception as e:
  print (e)
```

Conclusion

In this chapter, you have learned about exception handling. In exception handling, we handle the error. Sometimes a small error halts the execution. To control the error, we use the "`try`" and "`except`" blocks. In the except block, we can handle the exceptions based on its type. The "`except` `Exception` as "`e` "can handle any exception. The try statement can be used with "else" and "finally". The else is used when the try block gets executed successfully. The interpreter always executes the `finally` block, whether the error has occurred or not. In the advance part, we learned how to make our own exceptions. In the next chapter, you will learn about file handling.

Questions

1. Write the output of following program:

```
list1 = [0]
for each in list1:
  try:
    if each == 0:
      continue
  finally:
    print ("I am inevitable ")
```

2. Write any five types of exceptions.

Chapter 10
File Handling

We usually store data in a text file. The text files are convenient to use as they are platform-independent; almost every operating system recognizes text files. The text files are straightforward to read and write. A text file can act as both, an input and output, for a program. There is a particular format, which is used to store the complex data structures like list and dictionary. In this chapter, we will use pickle and JSON to store complex data structures. The stored data can be considered as a local database and retrieved again.

Structure

- Text files
- Reading text from a file
- Writing text to a file
- With statement
- Pickle
- JSON with Python

Objective

In this chapter, you will learn how to read and write a text file. You will learn about reading and writing the data, line-by-line. After the text file, the pickle and JSON files will be discussed. We will see the advantages of these files over text files.

Text files

We have seen examples of programs that have taken the input data from the users at the keyboard. With Python, it is easy to read strings from plain text files. You can store data in a text file. As you already know, text files are platform-independent.

Reading text from a file

The Python built-in function `open()`, is used to open the file. The `open()` function creates a file object, which is used to read and write a file.

See the following syntax of the `open()` function:

```
file object = open(file_name, access_mode)
```

The first argument, `file_name`, represents the file name that you want to access.

The second argument, `access_mode`, determines in which mode the file has to be opened - that is, `read, write, append`, and so on.

The `Access_mode` for reading is 'r'.

We created a file on our system using a text editor. Showcased below are the contents of the file:

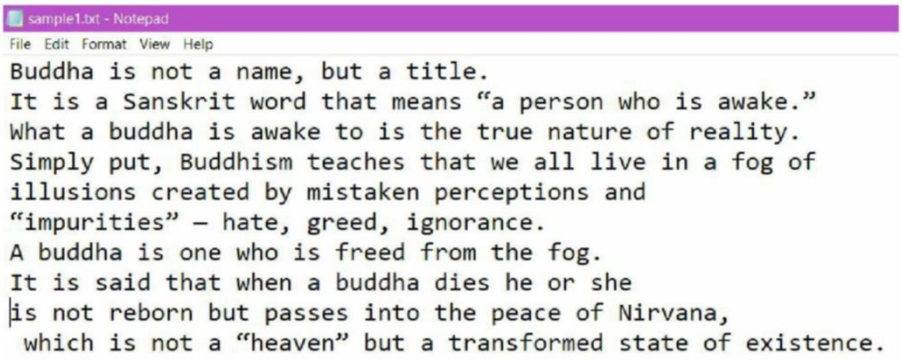

Figure 10.1

We saved the file with the name `simple1.txt`, and put it in the same directory for easy access.

See the following program:

```
file_read = open("sample1.txt", "r")
print (file_read.read())
file_read.close()
```

Let us see the output first, then we will discuss the code:

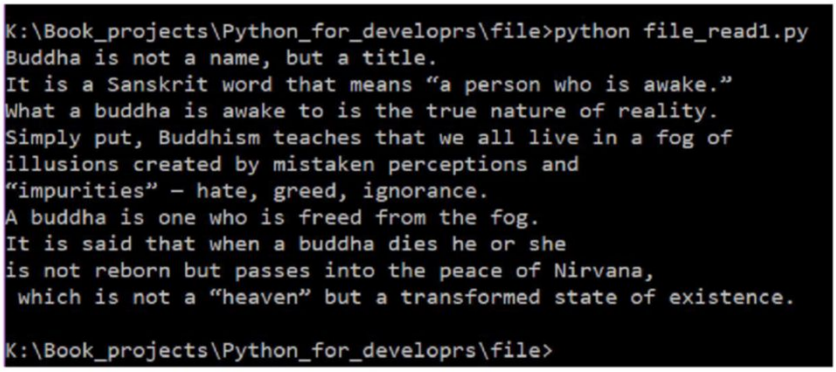

Figure 10.2

The `file_read` is an object, and the `read()` method has been used to read the entire content of the file. The last `line, file_read.close()`, closed the file.

You can read the characters from the file by supplying an integer argument to the `read()` method.

Let us discuss this in next example:

```
file_read = open("sample1.txt", "r")
print(file_read.read(9))
print (file_read.read(10))
file_read.close()
```

See the following screenshot for the output:

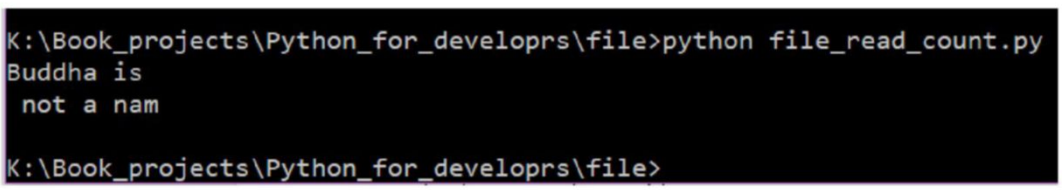

Figure 10.3

The second-line, `file_input.read(9)`, reads the first nine characters. The third line reads the next ten characters.

To read line-by-line, use the `readline()` method. Let us discuss this in the next example.

```
file_r = open("sample1.txt", "r")
print (file_r.readline())
print (file_r.readline(15))
print (file_r.readline(7))
file_r.close()
```

Let us see the output in the following screenshot:

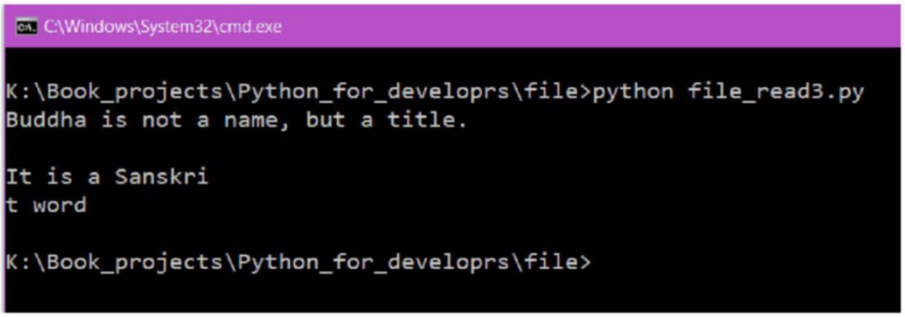

Figure 10.4

If you specify the count integer in the `readline()` method, then it reads character by character, similar to the `read(count)` method.

What happens if we use the `readlines()` function.

Let us discuss this in the next example:

```
file_r = open("sample1.txt", "r")
print (file_r.readlines())
file_r.close()
```

See the following screenshot:

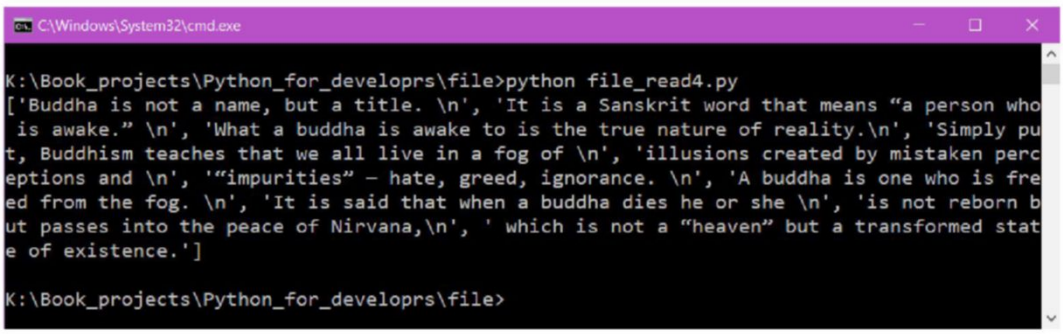

Figure 10.5

It returns a list of lines.

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to a simple code:

```
file_r = open("sample1.txt", "r")
for line in file_r:
  print (line )
file_r.close()
```

Let us see the output in the following screenshot:

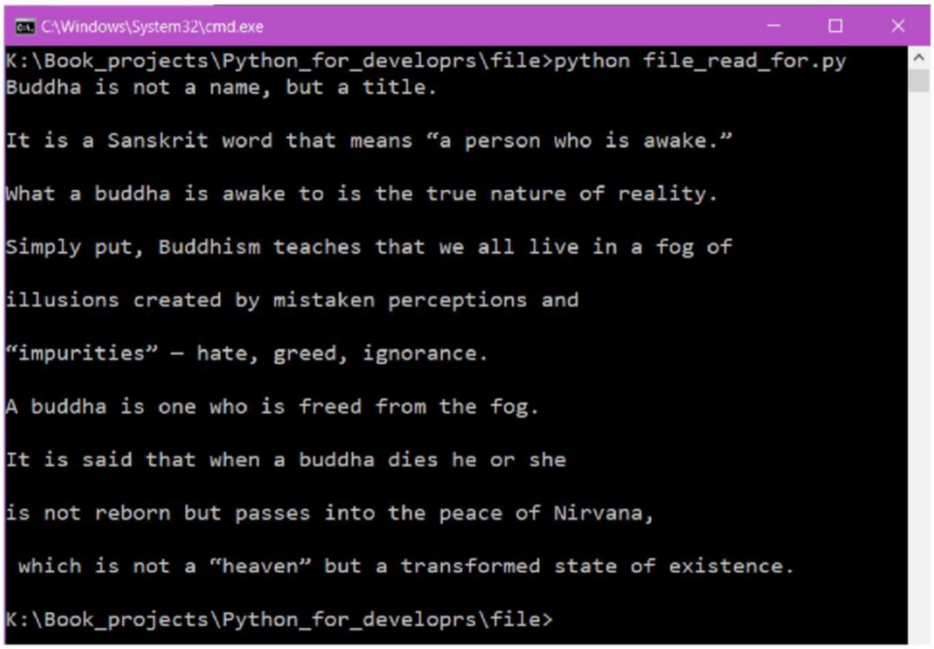

Figure 10.6

If file is small, you can use the `read()` method. If file is very large, then you can use the *loop over file object* methodology to read. If you want to know the current position of the pointer, then you can use the `tell()` method. The `tell()` method returns the integer that points to the current position.

Writing text to a file

In this section, we will discuss writing a file with Python. This time we will use the write mode "`w`", in the `open()` function.

We will use the `write()` function.

Let us discuss our example:

```
file_w = open("pfd.txt", "w")
file_w.write("Welcome everyone\n")
file_w.write("Python Programming \n")
file_w.write("Nice to use ")
file_w.close()
```

The output is showcased in the following screenshot:

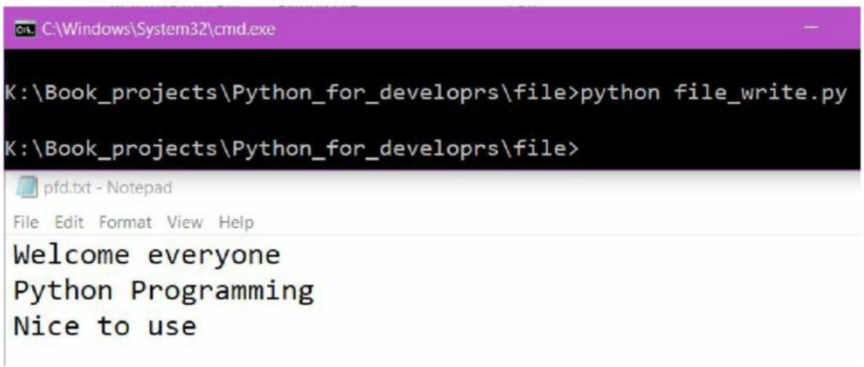

Figure 10.7

The `file_write.py` program has created a new file, `pdf.txt`. It has also written three lines to the file, as showcased in the preceding screenshot.

You can write all three lines in a single function like this:

```
file_w.write("Welcome everyone\n Python Programming \n Nice to use ").
```

Next, we create the same file using the `writelines()` function. This method writes a list of strings to a file. In this, we create a list of strings to be written:

```
list1 = [ 'Welcome everyone\n', 'Python Programming \n',
'Nice to use',]
file_w = open("pfd2.txt", "w")
file_w.writelines(list1 )
file_w.close()
```

The output is showcased in the following screenshot:

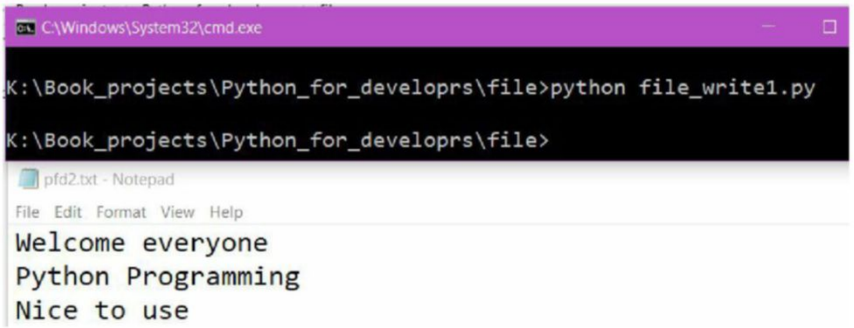

Figure 10.8

Here is a list of the different modes for opening a file:

Modes	Description
`r`	Opens a file for reading only. This is the default mode. This mode places the pointer in the beginning of the file.
`rb`	Opens a file for reading only in a binary format. This mode places the pointer in the beginning of the file.
`r+`	Opens a file for reading and writing. This mode places the pointer in the beginning of the file.
`rb+`	Opens a file for reading and writing only in the binary format. This mode places the pointer in the beginning of the file.
`w`	Opens a file for writing only. Creates the file, if it doesn't exist. If the file exists, then overwrites the file.
`wb`	Opens a file for writing only in the binary mode. Creates the file, if it doesn't exist. If the file exists, overwrites the file.
`w+`	Opens a file for reading and writing. Creates the file, if doesn't exist. If the file doesn't exist, overwrites the file.
`wb+`	Opens a file for reading and writing in binary mode. Creates the file if it doesn't exist. If the file exists, overwrites the file.
`a`	Opens a file for the appending mode. If the file doesn't exist, it creates a new file. If the file already exists, the pointer is placed at the end of the file.
`ab`	Opens a file for appending in binary mode. If the file doesn't exist, it creates a new file. If the file already exists, the pointer is placed at the end of the file.
`a+`	Opens a file for appending and reading. If the file doesn't exist, it creates a new file. If the file already exists, the pointer is placed at the end of the file.
`ab+`	Opens a file for appending and reading in binary mode. If the file doesn't exist, it creates a new file. If the file already exists, the pointer is placed at the end of the file.

Table 10.1

The w mode creates a new file and writes again, but we want to update the file. Let us discuss the a+ mode:

```
file_w = open("pfd.txt", "a+")
file_w.write("Good for beginners \n")
file_w.write("Got number rank 1 in IEEE ranking\n")
file_w.close()
```

See the following screenshot:

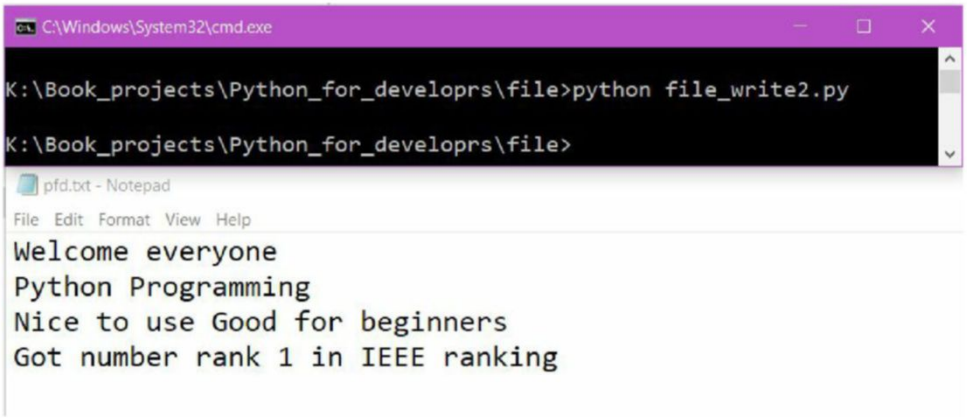

Figure 10.9

You have seen that in the a+ mode, the file got appended. You have seen the examples of reading and writing. Actually, `file_object` is like a pointer. When we use the `read()` method, the pointer starts from the first position, reads the entire file and goes to end of the file. Let us understand this through an example:

```
file_read = open("sample1.txt", "r")
print (file_read.read())
print (file_read.read())
file_read.close()
```

In the preceding code, the read method has been used two times. In the first glance, we can say that the text should be written two times. Let us see the output in the following screenshot:

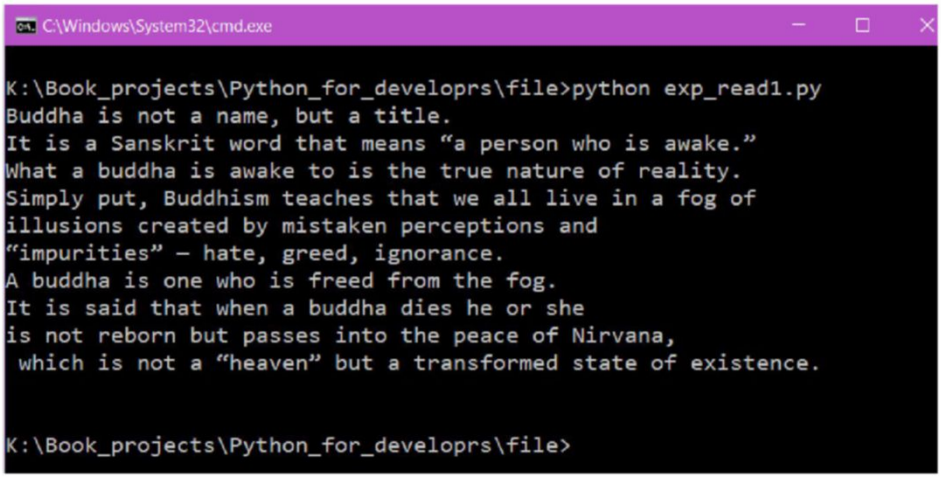

Figure 10.10

The preceding screenshot shows that we are getting the desired text only once. The second time, using the `read()` method gave nothing, because when the interpreter called the `read()` method the second time, the pointer was at the end position of the file. With the help of the `seek()` method, you can place the pointer at the beginning of the file. See the following code:

```
file_read = open("sample1.txt", "r")
print (file_read.read())
file_read.seek(0,0)
print (file_read.read())
file_read.close()
```

Let us see the output in the following screenshot:

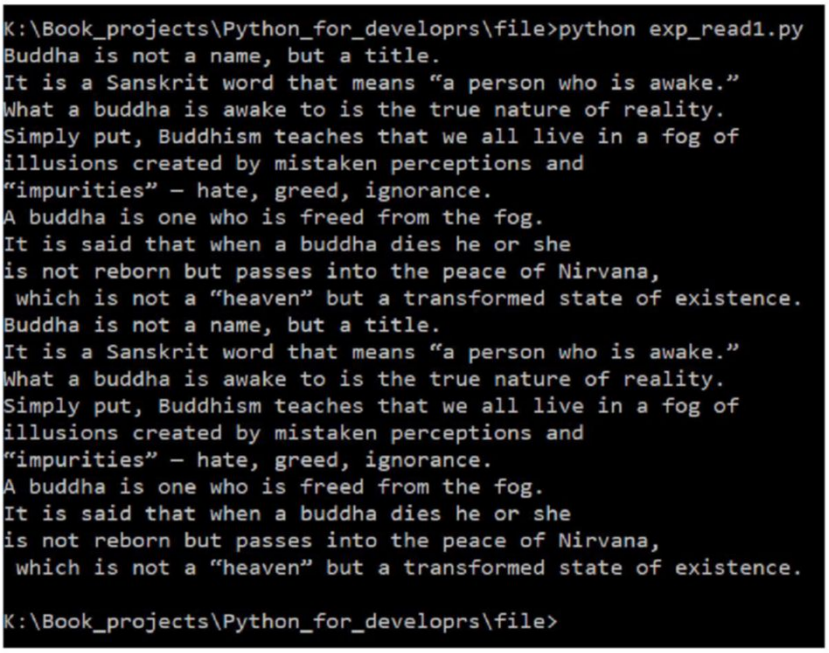

Figure 10.11

To place the pointer at the beginning of the file, we use `seek(0,0`, and to place the pointer at the end of the file, we use the `seek(0,1)` method.

The with statement

With the help of the with statement, you can read and write the files. The with statement also takes care of exception handling. Let us see the program:

```
with open("sample1.txt", "r") as f_r:
  print (f_r.read())
```

See the code's output in the following screenshot:

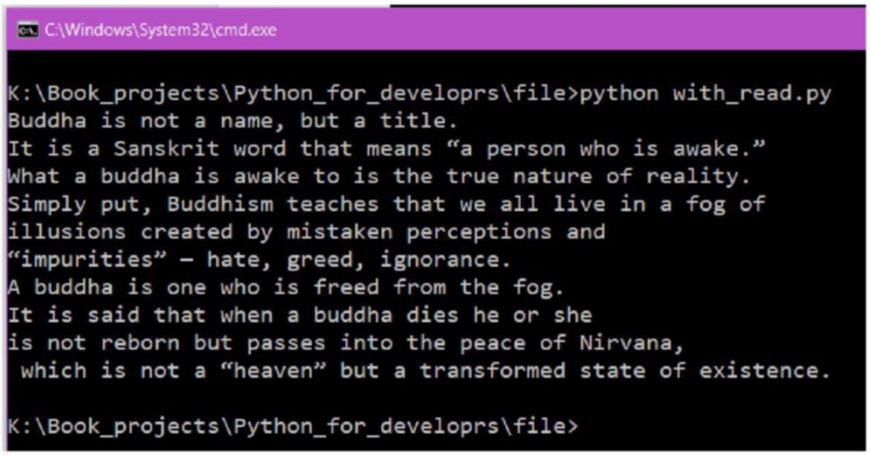

Figure 10.12

You can see that we have not used the close statement. with also takes care of closing the file. Similarly, let us see the code for writing a file:

```
with open("sample3.txt", "w") as f_w:
  f_w.write("what we think we become\n")
  f_w.write("Hackers are here where are you ")
```

The following screenshot is showcasing the output:

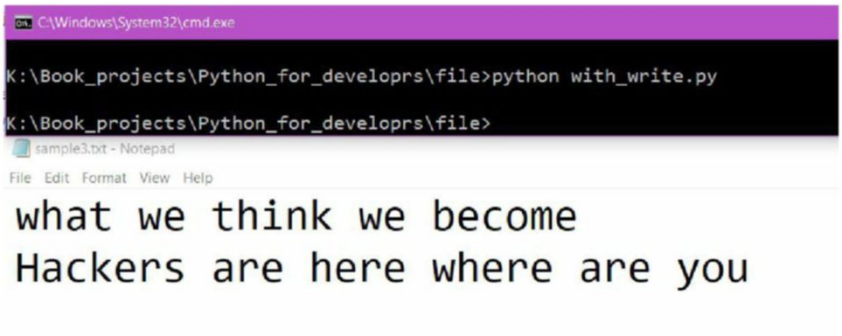

Figure 10.13

You can see how easy it is to read and write the file using the with statement. In the next section, we will learn about pickle.

Pickle

Text files are convenient because you can read, write, and append them with any text editor. However, they are limited to storing a series of characters. Sometimes, you may want to save more complex information, like the list, the dictionary.

Here we will use Python pickle. Python pickle is used for storing more complex data like a list or a dictionary. Let us discuss this with the help of an example:

```
import pickle
name = ["Mohit", "Sahil", "Ravender"]
skill = ["Python", "Face recog", "Data Science"]
file_w = open("sample.raj", 'wb')
pickle.dump(name, file_w)
pickle.dump(skill, file_w)
file_w.close()
```

The program seems a little complex. Let us discuss it line-by-line:

```
import pickle
```

The `pickle` module allows you to pickle and store more complex data in the file:

```
name = ["Mohit", "Sahil", "Ravender"]
skill = ["Python", "Face recog", "Data Science"]
```

These are the two lists that have to be stored:

```
file_w = open("sample.raj", 'wb')
```

Create a file object in the "write" mode, as we have learned in the previous File chapter:

```
pickle.dump(name, file_w)
pickle.dump(skill, file_w)
```

We want to store two lists, name and skill, in the `sample.raj` file using the `pickle.dump()` function. The function requires two arguments – first, the data to pickle, and second, the file to store it.

```
file_w.close()
```

Finally, the program closes the file.

So, this code pickles the list referred to by name, and writes the whole thing as one object to the `sample.raj` file. Next, the program pickles the list referred to by skill and writes the entire thing as one object to the file.

Reading data from file and unpickling

In this section, we will see how to retrieve and unpickle the two lists with the `pickle.load()` function. The function takes one argument as a file, from which it has to load the next pickled object:

```
import pickle
file_r = open("sample.raj", 'rb')
list1 = pickle.load(file_r)
print (list1)
list1 = pickle.load(file_r)
print (list1)
file_r.close()
```

See the following screenshot:

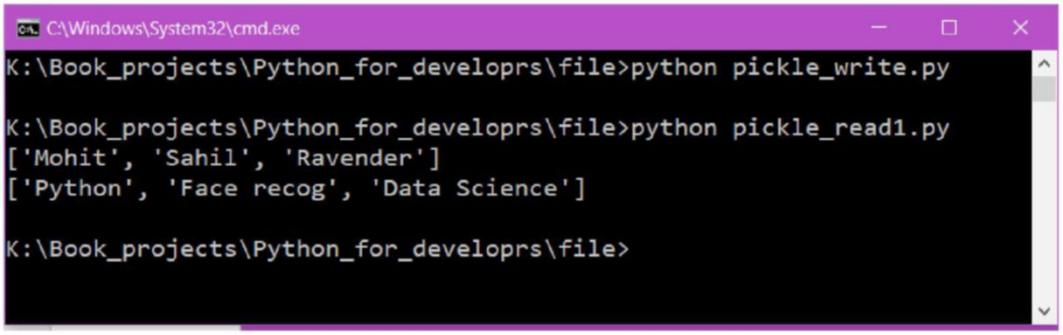

Figure 10.14

The program reads the first pickled object in the file, unpickles it to produce the list `["Mohit", "Sahil", "Ravender"]` and assigns the list to name. Next, the program reads the next pickled object from the file and unpickles it to produce the list `["Python", "Face recog", "Data Science"]` and assigns the list to a variable skill. You can pickle number, string, list, tuple, and dictionary.

Now we can say that pickle stores and retrieves the list sequentially. You cannot access keys randomly. But with the help of the dictionary, we can access the lists randomly. Let us see the next program:

```
import pickle
name = ["Mohit", "Sahil", "Ravender"]
skill = ["Python", "Face recog", "Data Science"]
dict1 = {}
```

```
dict1['name'] = name
dict1['skill'] = skill
file_w = open("sample1.raj", 'wb')
pickle.dump(dict1, file_w)
file_w.close()
```

After running the program, a file named `sample1.raj` will be created. Let us read the file, with the help of the following program:

```
import pickle
file_r = open("sample1.raj", 'rb')
data = pickle.load(file_r)
print (type(data))
print (data.get('skill'))
print (data.get('name'))
```

Let us run and see the output in the following screenshot:

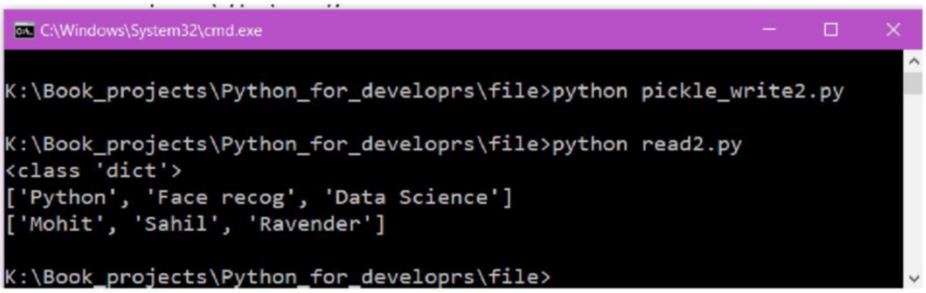

Figure 10.15

In the preceding program, the `pickle.load(file_r)` statement returns a dictionary, which contains two keys: name and skill. Now we can access the lists, which act as the values of the dictionary data, with the help of keys. You have seen that the pickle files are beneficial for storing the data.

Let's consider a situation where you stored a data in the hard disk, using the pickle format and that data has to be used by a different programming language. In that case, the pickle file would not work, because only Python can understand the pickle syntax. For a case like this, we need a format that can be recognized by another programming language. To solve that problem, we use the JSON format.

JSON with Python

JSON (JavaScript Object Notation) is a very common format for creating web APIs.

Let us see how to store a dictionary in a JSON file:

```
import json
name = ["Mohit", "Sahil", "Ravender"]
skill = ["Python", "Face recog", "Data Science"]
dict1 = {}
dict1['name'] = name
dict1['skill'] = skill
fw =  open("mydata.json", "w")
json.dump(dict1,fw, indent=2)
fw.close()
```

The syntax is very similar to the previous pickle programs. JSON format can be read easily.

See the following screenshot:

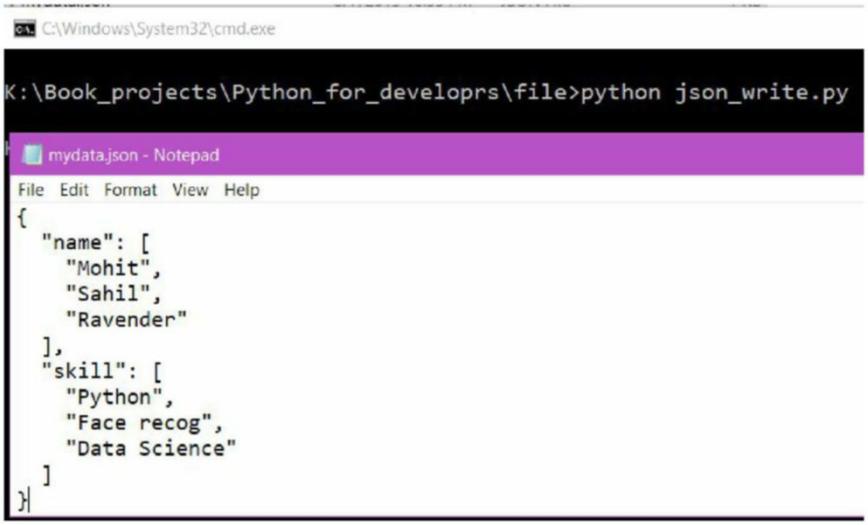

Figure 10.16

Reading the JSON file using Python program:

```
import json
f =  open("mydata.json", "r")
data = f.read()
jsondata = json.loads(data)
print (jsondata)
print (jsondata.get('skill'))
f.close()
```

Output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\file>python json_read.py
{'name': ['Mohit', 'Sahil', 'Ravender'], 'skill': ['Python', 'Face recog
', 'Data Science']}
['Python', 'Face recog', 'Data Science']

K:\Book_projects\Python_for_developrs\file>
```

Figure 10.17

To read the JSON file, we open the file, read the data, and put the data in the `json.loads()` method. The method returns the dictionary. On the returned dictionary, you can apply the dictionary operations.

Exercise

1. Find the frequency of a particular word, irrespective of the case.
2. Find the duplicate line in the file.

Answer 1:

Solution 1.

```
If file size is small.
file_r = open("sample1.txt", 'r')
text = file_r.read().lower()
str1 = input("Enter the string to be checked ")
occ= text.count(str1)
print ("The %s occured %d times"%(str1, occ))
```

See the following screenshot for the output:

```
C:\Windows\System32\cmd.exe

K:\Book_projects\Python_for_developrs\file>python file_read_excer.py
Enter the string to be checked is
The is occured 13 times

K:\Book_projects\Python_for_developrs\file>python file_read_excer.py
Enter the string to be checked buddha
The buddha occured 4 times

K:\Book_projects\Python_for_developrs\file>
```

Figure 10.18

If file size is huge:

```
file_r = open("sample1.txt", 'r')
i =0
str1 = input("Enter the string to be checked ")
for line in file_r:
  n=line.lower().count(str1)
  i = i+n
print ("The %s occurred %d times"%(str1, i))
```

The output is showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\file>python file_read_excer2.py
Enter the string to be checked is
The is occurred 13 times

K:\Book_projects\Python_for_developrs\file>python file_read_excer2.py
Enter the string to be checked buddha
The buddha occurred 4 times

K:\Book_projects\Python_for_developrs\file>
```

Figure 10.19

Answer 2:

We purposely created two duplicate records, as showcased in the following figure:

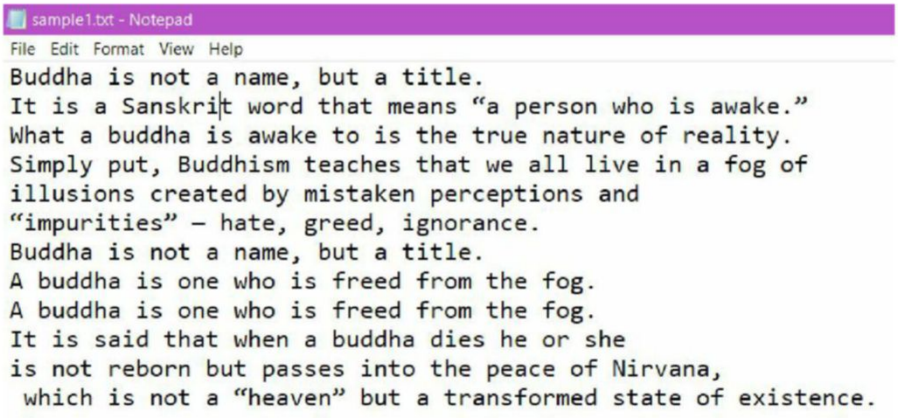

Figure 10.20

See the following program:

```
file_r = open("sample1.txt", "r")
list1 = []
i = 1
for line in file_r:
  line = line.strip()
  line = line.strip("\n")
  if line in list1:
    print ("Duplicate Record--> ", line, "\nat ",i )
  else :
    list1.append(line)
  i = i+1
file_r.close()
```

See the output in following screenshot:

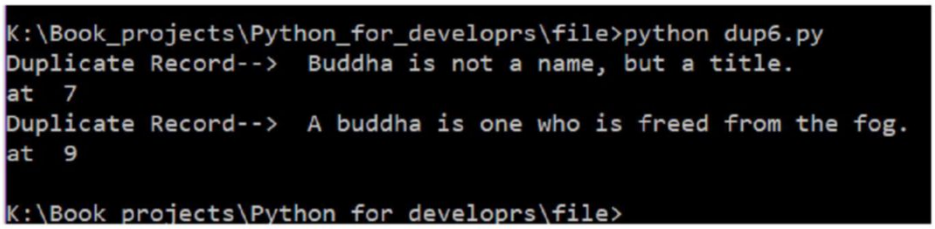

Figure 10.21

Conclusion

In this chapter, you have learned about storing and reading data from a text file. The text file stores the data in the string format. The read() method is mainly used to read the text file and the write() method is used to write the text file. You learned that pickle files are used to store the complex data structures like list, dictionary, and tuple. The JSON file can be read and written in different programming languages. In the next chapter, you will learn about the collections module, which contains a special type of containers.

Questions

1. For writing purpose, what is the difference between the "w" and "a" mode?
2. What is the optimized way to print a text file the line-by-line?
3. What are the advantages of a pickle file over a text file?
4. What are the benefits of a JSON file over a pickle file?

Chapter 11
Collections

So far, you have learnt the built-in data structure of Python such as list, tuple, and dictionary. The collections module includes the implementations of several data structures and offers the particular type of data structures such as `Counter`, `Namedtuple`, the `Setdefault` dictionary, deque, and the ordered dictionary. We know a dictionary is an unordered collection of key-value pairs. If we want to retain the order of items, then we can use the `orderedDict` dictionary of collections module. Similarly, Deque offers a double-ended list, on which we can perform addition and deletion at both ends. Gradually, you will learn the advantages of the collection module.

Structure

- Counter
- Deque
- Namedtuple
- The default dictionary
- The ordered dictionary

Objective

In this chapter, you will learn the collections modules. We will learn the different classes offered by the collections module. The collections module offers several data structures; however, in this chapter, we will learn the five data structure and they are `Counter`, `Namedtuple`, the `Setdefault` dictionary, deque, and the ordered dictionary.

Counter

A Counter is a container, and it tracks the frequency of the items. The counter takes a sequence as an argument and returns the frequency of each item of the sequence. Let's learn by examples.

See the following example:

```
>>> from collections import Counter
>>> co = Counter("what we think we become")
>>> co
Counter({' ': 4, 'e': 4, 'w': 3, 'h': 2, 't': 2, 'a': 1, 'i': 1, 'n': 1,
'k': 1, 'b': 1, 'c': 1, 'o': 1, 'm': 1})
>>>
```

In the preceding example, `Counter()` is a class and co is the object of the `Counter()` class. When we pass a string argument to the `Counter()` class, it returns a dictionary like data structure (key and value); the key represents the letter of string, and value represents the frequency.

Let's take an example of a list:

```
>>> co1 = Counter([1,2,3,4,54,5,6,1,2,2,3,4,5])
>>> co1
Counter({2: 3, 1: 2, 3: 2, 4: 2, 5: 2, 54: 1, 6: 1})
>>>
```

You saw the Counter `co1` object returned the dictionary-like data structure which contains items of the list with their frequency.

Let's see another example:

```
>>> Counter("INDIA")
Counter({'I': 2, 'N': 1, 'D': 1, 'A': 1})
>>> Counter(["INDIA"])
Counter({'INDIA': 1})
>>>
```

The preceding example shows two different examples of Counter. When you only pass one string, the occurrences of the string's letter will be returned. If the list contains a string, it will be treated as a single word by Counter.

Counter methods

In this section, we will see some important methods offered by the `Counter` class. If we want to pass arguments like tuple, list, and so on, to the Counter object after creation, then we can use the `update()` method.

update()

An empty or non-empty Counter can be updated by the `update()` method.

The following is the syntax of the `update()` method:

```
Co.update(sequence)
```

In the preceding syntax, the Co is the object of the `Counter()` class and sequence refers to a tuple, list, string, and so on.

Let's explore some examples of the `update()` method:

```
>>> from collections import Counter
>>> co1 = Counter("Intel")
>>> co1
Counter({'I': 1, 'n': 1, 't': 1, 'e': 1, 'l': 1})
>>> co1.update('dell')
>>> co1
Counter({'l': 3, 'e': 2, 'I': 1, 'n': 1, 't': 1, 'd': 1})
```

You can see the frequency letters of the string "`Intel`" have been updated by the new string '`dell`'.

In the following example, the existing `Counter` object is updated by the dictionary:

```
>>> co1.update({'e':2,'d':3})
>>> co1
Counter({'e': 4, 'd': 4, 'l': 3, 'I': 1, 'n': 1, 't': 1})
>>> co
Counter({'p': 3, 'z': 2, 'y': 1, 't': 1, 'h': 1, 'o': 1, 'n': 1})
```

With the help of the subscript operator, you can obtain the frequency of a particular key as shown in the following example:

```
>>> co1['e']
```

```
4
>>> co1['x']
0
>>>
```

The Counter does not raise `KeyError` for unknown items. If the item is not present, the counter returns `0`.

Consider a problem of the real world to calculate the frequency of letters from a text file. A text example is shown in the following figure:

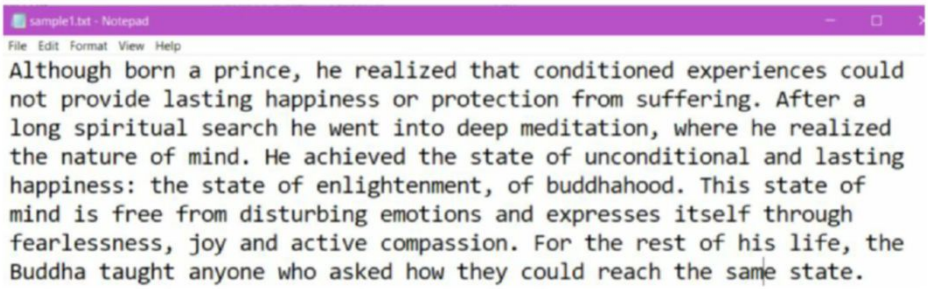

Figure 11.1

The following code calculates the frequency of each letter:

```
from collections import Counter
co = Counter()
file_r = open("sample1.txt")
text = file_r.read()
co.update(text)
file_r.close()
print (co)
print ("**************")
print (co.most_common(5))
```

The output is shown in the following screenshot:

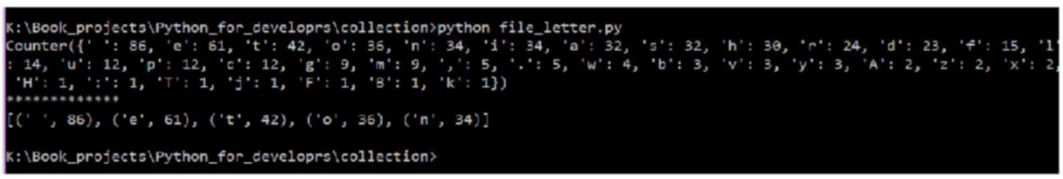

Figure 11.2

The preceding program displays the frequency of all letters. However, you can choose the first five letters of higher frequency. To select the top frequency letter, you can use `most_common()`.

The `counter` object contains several methods; we can check all the method as shown in the following screenshot:

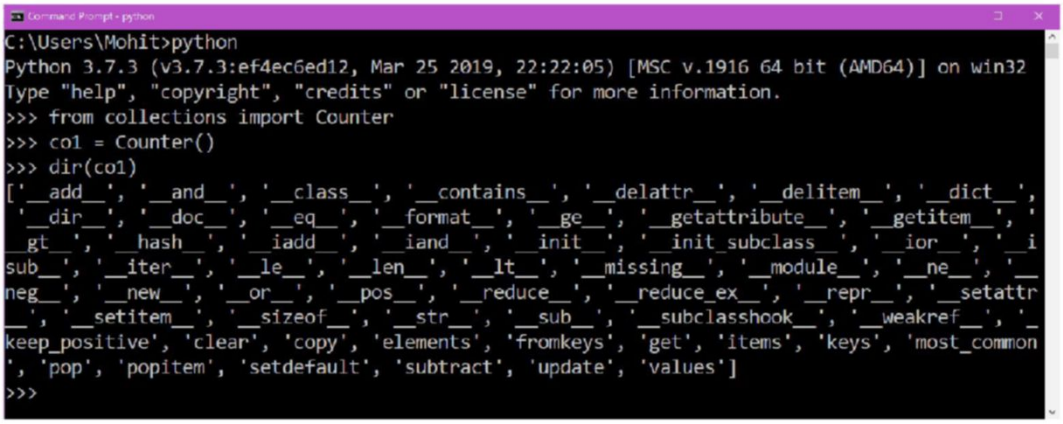

Figure 11.3

Almost all the methods are similar to dictionary methods.

Counter operations

In the Counter, you can apply sets operation such as addition, subtraction, union, and intersection.

Consider we have two different Counter objects, as shown in the following examples:

```
>>> co1 = Counter("hacker")
>>> co1
Counter({'h': 1, 'a': 1, 'c': 1, 'k': 1, 'e': 1, 'r': 1})
>>> co2 = Counter("developer")
>>> co2
Counter({'e': 3, 'd': 1, 'v': 1, 'l': 1, 'o': 1, 'p': 1, 'r': 1})
>>>
```

Addition

Let's add two counters. See the following example of addition of two counters:

```
>>> co1 + co2
Counter({'e': 4, 'r': 2, 'h': 1, 'a': 1, 'c': 1, 'k': 1, 'd': 1, 'v': 1,
```

```
'l': 1, 'o': 1, 'p': 1})
>>>
```

The result shows that the frequency of common member has been added. The following Venn diagram illustrates the preceding result. The blue region is our answer:

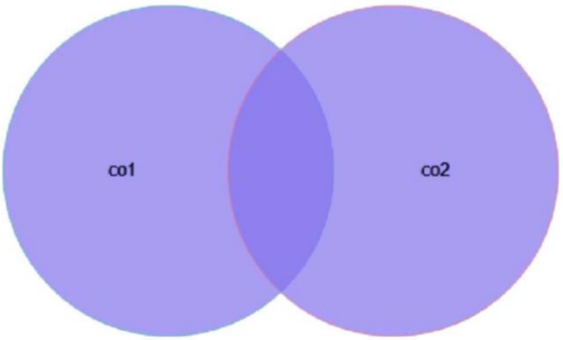

Figure 11.4

Subtraction

The frequency of common members has been subtracted from `Counter co1`. The following example shows the result of subtraction:

```
>>> co1 - co2
Counter({'h': 1, 'a': 1, 'c': 1, 'k': 1})
```

The following Venn diagram illustrates the preceding result. The blue region is our answer:

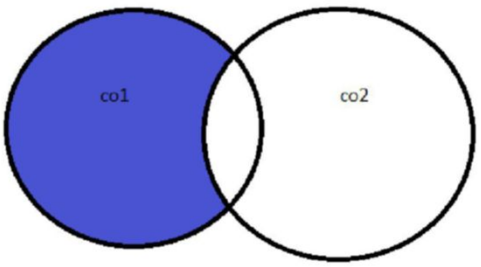

Figure 11.5

Union

In the union, we add the members of Counter `co1` and `co2`. The common members have been not added. See the following example:

```
>>> co1 | co2
```

```
Counter({'e': 3, 'h': 1, 'a': 1, 'c': 1, 'k': 1, 'r': 1, 'd': 1, 'v': 1,
'l': 1, 'o': 1, 'p': 1})
>>>
```

The following Venn diagram illustrates the preceding result. The blue region is our answer:

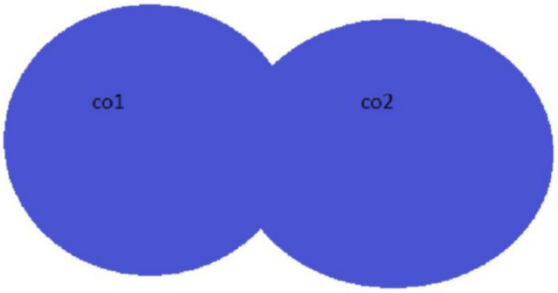

Figure 11.6

Intersection

The intersection returns the common member of co1 and co2. See the following example:

```
>>> co1 & co2
Counter({'e': 1, 'r': 1})
>>>
```

The following Venn diagram illustrates the preceding result. The blue region is our answer:

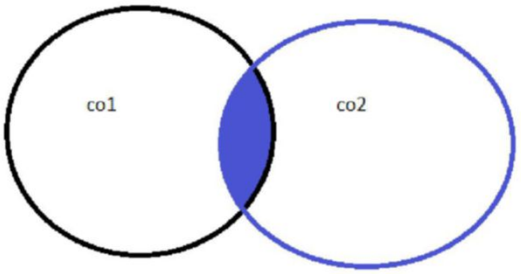

Figure 11.7

Deque

A double ended queue allows a use to add and remove an item at the both ends. In instances where we need faster result, append and pop activities from both sides of

the container. Deque is preferred over list because it takes O(1) operation for appends and pops on either side. Deque is the enhanced version of the list.

Let's discuss Deque operations and methods one by one. Deque supports some list operations.

See the following example:

```
from collections import deque
de = deque("MOHIT")
print ("The number of items are :",len(de))
print ("Deque is ", de)
print ("First item ",de[0])
print ("Last item",de[-1])
```

As Deque is also a sequence, we can use the subscript operator with that. See the output in the following screenshot:

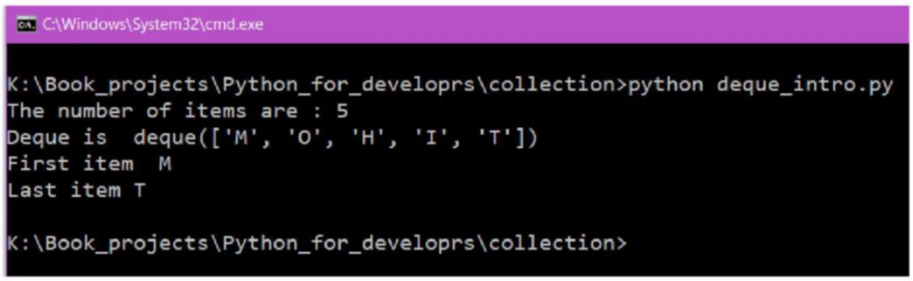

Figure 11.8

The `len()` function returns the length of the deque. You can see the right and left end of deque in the preceding screenshot.

Deque populating

Let's discuss more operations of Deque. As we said earlier, double-ended queue means you can add an item from either side.

Let's add more items to the `deque`:

```
from collections import deque
de = deque("MOHIT")
de.append("python")
print (de)
de.appendleft("Intel")
print (de)
```

```
de.extend("DELL")
print (de)
de.extendleft("DELL")
print (de)
```

The output in the following screenshot clearly shows the code effect. The `extendleft()` method iterates over its input and performs the equivalent of an `appendleft()` method for each item. The end result is that the deque contains the input sequence in the reverse order:

```
K:\Book_projects\Python_for_developrs\collection>python deque_add.py
deque(['M', 'O', 'H', 'I', 'T', 'python'])
deque(['Intel', 'M', 'O', 'H', 'I', 'T', 'python'])
deque(['Intel', 'M', 'O', 'H', 'I', 'T', 'python', 'D', 'E', 'L', 'L'])
deque(['L', 'L', 'E', 'D', 'Intel', 'M', 'O', 'H', 'I', 'T', 'python', 'D', 'E', 'L', 'L'])

K:\Book_projects\Python_for_developrs\collection>
```

Figure 11.9

deque consuming

Like adding, the deque can be consumed from both ends or either end. With the help of pop and `popleft()`, the deque can be consumed.

Let's discuss our next example:

```
from collections import deque
de = deque("Python")
print (de)
print ("poped from right ",de.pop())
print (de)
print ("Poped from left ", de.popleft())
print (de)
```

See the output in the following screenshot:

```
C:\Windows\System32\cmd.exe

K:\Book_projects\Python_for_developrs\collection>python deque_remove.py
deque(['P', 'y', 't', 'h', 'o', 'n'])
poped from right  n
deque(['P', 'y', 't', 'h', 'o'])
Poped from left  P
deque(['y', 't', 'h', 'o'])

K:\Book_projects\Python_for_developrs\collection>
```

Figure 11.10

deque rotating

The deque allows you to rotate items on either side. Right rotation or clockwise is taken as positive rotation. Use `rotate(n)` for right rotation up to n number. For left rotation, use `rotate(-n)`:

```
from collections import deque
de = deque([1,2,3,4,5])
print (de)
de.rotate(2)
print (de)
de.rotate(-2)
print (de)
```

See the following screenshot for the output:

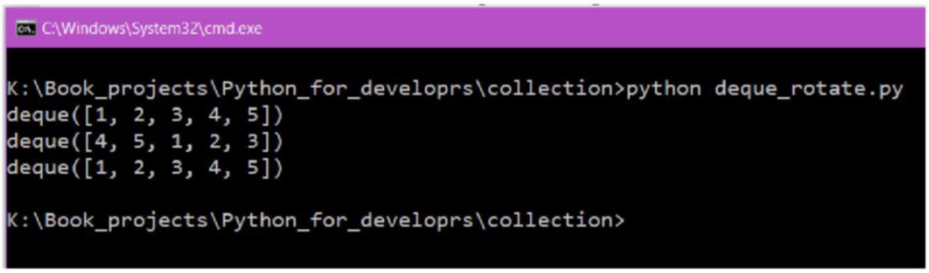

Figure 11.11

From the output, you can deduce that in the right rotation, items are shifted to the right direction. In the left rotation, things are shifted to the left direction.

There are some more methods that you can try:

```
>>> from collections import deque
>>> dir(deque)
['__add__', '__bool__', '__class__', '__contains__', '__copy__', '__
delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__
iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__
le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append',
'appendleft', 'clear', 'copy', 'count', 'extend', 'extendleft', 'index',
'insert', 'maxlen', 'pop', 'popleft', 'remove', 'reverse', 'rotate']
>>>
```

We did most of the methods in the list chapter.

Namedtuple

Now, we have seen several data types. But if you want to produce your datatype, the `namedtuple` of collection module allows you to create your data type. If you want to create a new type of data, you might want to ask some questions: what is the name of the new data type, what are the new data type fields, and so on. Let's discuss the `namedtuple` syntax:

`collections.namedtuple(typename, field_names verbose=False, rename=False])`

`typename` defines the name of the new datatype.

The `field_names` parameter can be a sequence of strings like `['x', 'y']` or a string in which values are white space or "`,`" separated.

If `verbose=False`, then the class definition would not be printed. It's a good idea to let it remain false.

If `rename=False`, then invalid `field_names` automatically get replaced with positional names; for example 'for, age, `empid`' is converted to '`_0`, age, `empid`' because '`for`' is a keyword.

Let's discuss our first example:

```
from collections import namedtuple
employee = namedtuple("IBM", "Name age emp", )
record1  = employee("MOHIT", 32, 121322)
print (employee)
print (record1)
print (record1.Name)
print (record1.age)
print (type(record1))
```

See the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\named>python namedtuple1.py
<class '__main__.IBM'>
IBM(Name='MOHIT', age=32, emp=121322)
MOHIT
32
<class '__main__.IBM'>

K:\Book_projects\Python_for_developrs\named>
```

Figure 11.12

Now you know how to access `namedtuple` values. Namedtuple is a beautiful thing; we are getting properties of tuple and dictionary in the `namedtuple`. Namedtuple are the immutable like tuples. You can access each element from its field name.

In the next example, you will see how to add list values into `namedtuple` and how to make a dictionary from `namedtuple`:

```
from collections import namedtuple

employee = namedtuple("Intel", "Name age emp", )

list1 = ["Mohit", 32, 121322]

record1  = employee._make(list1)

print (employee)

print (record1)

print (record1.Name)

print (record1.age)

print (record1._asdict())
```

With the help of the `_make()` method, a list can be added to `namedtuple` provided the list contains the same number of items as defined in `namedtuple`. The `_asdict()` method returns the `OrderedDict` form of `namedtuple`. In the forthcoming section, we will see the `OrderedDict`.

See the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\named>python namedtuple2.py
<class '__main__.Intel'>
Intel(Name='Mohit', age=32, emp=121322)
Mohit
32
OrderedDict([('Name', 'Mohit'), ('age', 32), ('emp', 121322)])

K:\Book_projects\Python_for_developrs\named>
```

Figure 11.13

You have seen that by using the `_make()` method you can add the list into a `namedtuple` and by using `_asdict`, you can create a dictionary of `namedtuple`.

Like tuple, the namedtuple are also immutable. But you can use the `_replace` method with reassignment to replace the value from `namedtuple`. See the following example:

```
from collections import namedtuple

employee = namedtuple("Intel", "Name age emp", )

record1  = employee("Mohit", 28, 121322)
```

```
print (record1)
print (record1._replace(age=32))
print (record1)
record1 = record1._replace(age=32)
print (record1)
```

The following screenshot shows the output:

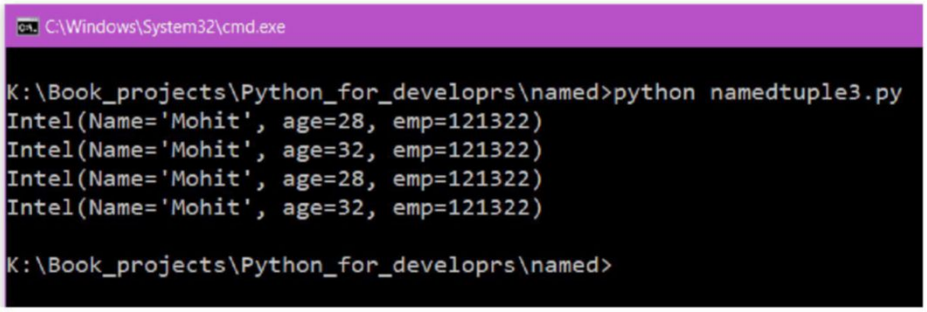

Figure 11.14

The preceding results show that the reassignment worked well.

The default dictionary

So far, we have seen the regular dictionary. Now, we will learn the default dictionary. The working of the default dictionary is the same as a regular dictionary. The default dictionary uses a callable function called `default_factory`. The functionality of `default_factory` will be seen in the following examples.

Function as default_factory

Let's take a function as `default_factory`. See the following code:

```
from collections import defaultdict
def fun1():
  return "cricket"
d1 = defaultdict(fun1)
d1["Bhaskar"] = "Football"
d1["Ankur"] = "Badminton"
print (d1)
print (d1["Bhaskar"])
print (d1["Ankur"])
```

```
print (d1["Mohit"])
print (d1)
```

See the output in the following screenshot:

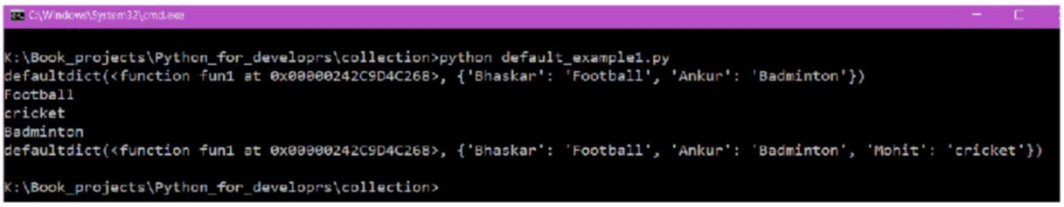

Figure 11.15

The `default_factory` is a function that is defined earlier. So what is the default value of the dictionary `d1`? By using `d1["Bhaskar"]="Football"`, we have defined a value "`Football`" on the "`bhaskar`" key. If a key is new (not found in dictionary `d1`), then `defaultdict` does not give an error, and it will return the default value that is returned by the `default_factory` function `fun1`. So, for the new key "`Mohit`", the default value is "`Cricket`".

int as default factory

Let's use int as `default_factory`. The default value of `int` is `0`. In the following example, we will calculate the frequency of a letter of a string:

```
from collections import defaultdict
d1 = defaultdict(int)
print (d1)
print (d1["X"])
print (d1['Y'])
str1 = "mohitpythonprogrammer"
for i in str1:
  d1[i]=d1[i]+1
print (d1)
```

See the following screenshot for output:

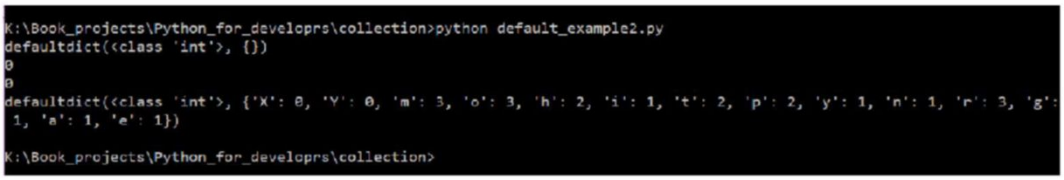

Figure 11.16

So `d1["X"]` and `d1["Y"]` are `0`. The preceding program calculated the frequency of characters of the string. You know the default value is `0`. We incremented it by `1` and in this way, the frequency gets calculated.

list as default_factory

Consider a real problem; you have a list of tuples pair shown as follows:

```
list_state = [('A', "a1"), ('A', 'a2' ),('B','b' ),('B','b2'),('C', 'c') ]
```

The preceding list is the pair of (state, city). So, our aim is to make the state as key and `district` a list of values. See the following code:

```
from collections import defaultdict
list_state = [('A', "a1"), ('A', 'a2' ),('B','b' ),('B','b2'),('C', 'c') ]
d1 = defaultdict(list)
print (d1["Mohit"])
print (d1)
for k,v in list_state:
  print (k,v)
print (d1)
```

Let's check the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\collection>python default_example3.py
[]
defaultdict(<class 'list'>, {'Mohit': []})
A a1
A a2
B b
B b2
C c
defaultdict(<class 'list'>, {'Mohit': []})

K:\Book_projects\Python_for_developrs\collection>
```

Figure 11.17

The default value is a Python list here. The first value of tuple is fixed as key and the second value is being appended.

The ordered dictionary

The regular dictionary does not remember the order in which the items have been entered. The order does not matter in the regular dictionary because the values are accessed using the keys. `OrderedDict` is a dictionary that remembers the order in which its contents are added.

See the following syntax of the ordered dictionary:

```
d1 = collections.OrderedDict()
```

```
d1 is ordered dictionary here.
```

Let us see a simple example and for loop iteration on the `OrderedDict`:

```
from collections import OrderedDict
d1 = OrderedDict()
print (d1)
d1["D"] = "Dell"
d1["I"] = "IBM"
d1['S'] = 'Sapient'
d1["M"] = "MOHIT"
d1["T"] = "INTEL"
print (d1)
for k,v in d1.items():
  print (k,v)
```

The result is shown in the following screenshot:

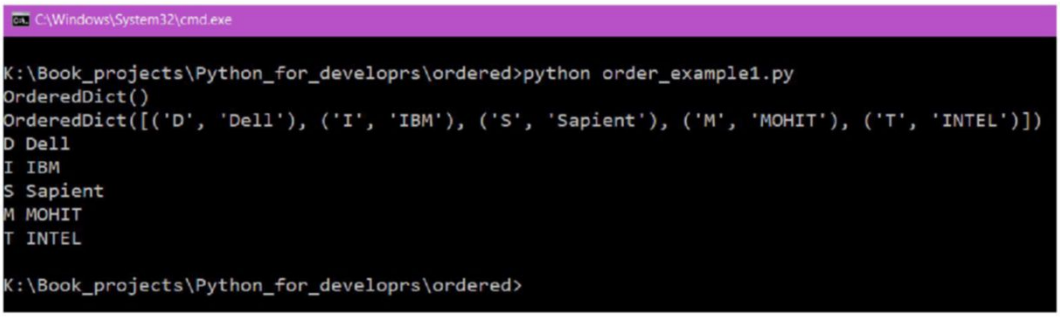

Figure 11.18

Consider you have a regular dictionary, and you want to sort the items based on key or value and want to retain the sorted order, then we will have to use the ordered dictionary.

Sorted of dictionary based upon key

As we know, the regular dictionary may not maintain the order as they have entered. The sorted function only works on the list or tuple. First, we will convert the given regular dictionary to a list, then we will apply the `sorted` function; the sorted function then sorts the list. The sorted list then will be saved to `OrderedDict` again.

See the following code:

```
from collections import OrderedDict
d1 = {}
print (d1)
d1[1] = "k"
d1[3] = "b"
d1[2] = "d"
d1[10] = "a"
d1[10] = "c"
list1 = sorted(d1.items())
d2 = OrderedDict(list1)
print (d2)
```

See the following output:

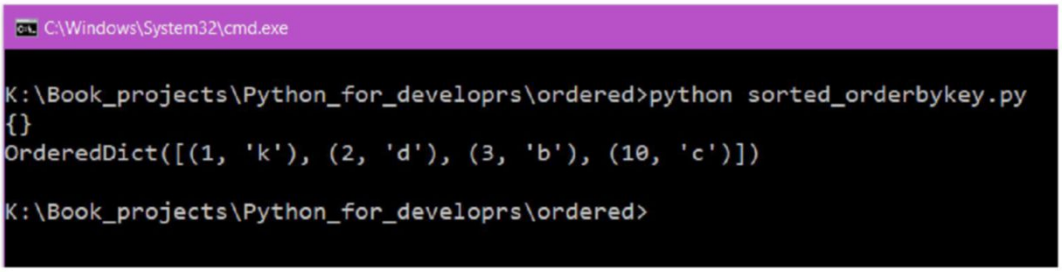

Figure 11.19

By default, the dictionary is sorted according to the key.

Sort the dictionary based upon values

Let's write code to sort the dictionary according to the values. See the following code:

```
from collections import OrderedDict
d1 = {}
print (d1)
d1[1] = "k"
d1[3] = "b"
d1[2] = "d"
d1[5] = "a"
```

```
d1[10] = "c"
list1 = sorted(d1.items(),key= lambda tup : tup[1])
d2 = OrderedDict(list1)
print (d2)
```

The preceding lambda function changes the key to its value. Because `d1.items()` returns the (`key, value`) pair. The `lambda` function makes `key = value`; thus, the dictionary will be sorted by its value and stored in the ordered dictionary. See the output in the following screenshot:

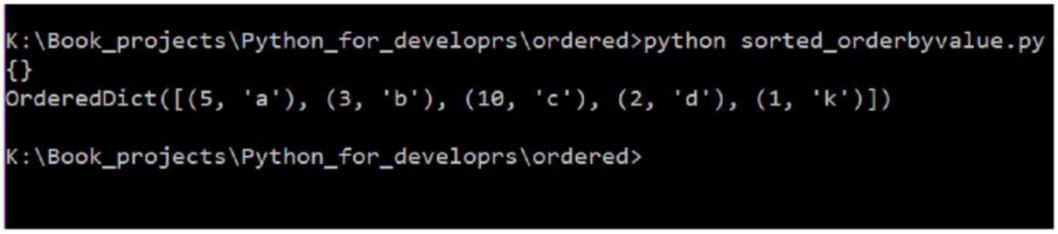

Figure 11.20

The `lambda` function will be explained in the *Chapter 12, Random Modules and Built-in Function* chapter.

Conclusion

In this chapter, you have learnt about collections modules that offered the special types of data structure. These data structures are like an enhanced version of a built-in data structure such as deque, which has the ability to append and extend from both the ends. The `OrderedDict` retains the order in which the items have been inserted. You have learnt that `namedtuple` facilitates you to create your own data type. The data type offers the power of tuple and dictionary simultaneously. The default dictionary offers the way to keep duplicate keys. You have learnt that the Counter is used to calculate the frequency of items of the sequence. This would be helpful in the field of text analytics. In the next chapter, we will learn about the random module and some built-in function of Python programming.

Question

1. Calculate the frequency of words of a text file.
2. Calculate the frequency of vowels in a given string.
3. If we don't use the default factory in defaultdict, then what would happen?

CHAPTER 12
Random Modules and Built-in Function

"**The OTP has been sent. Generate the password**," you must have received such kind of messages. Have you ever wondered how the system generates random OTP or passwords? The Python offers a module called random, which facilitates you to create random numbers. The Radom module allows you to create random integers, float, and random items from the list. Python also offers some built-in function which gives you a greater control on programming. For example, the `filter` function filters the sequence based on the condition provided by the programmer. The `map` function creates a new sequence object of mapped values in a given sequence. The `reduce` function reduces the sequence based on the given condition and returns a single value. There are more examples that you will learn in this chapter.

Structure

- The random module
    - o Random functions for integers
    - o Random functions for sequence
    - o Random functions for floats
- Python special functions
    - o Lambda
    - o filter

- map
- reduce
- isinstance
- eval
- repr

Objective

In this chapter, you will learn the significance of the random module which helps a programmer to create a random number. You will learn about some built-in functions such as `map`, `filter`, `lambda`, `eval`, and `isinstance`. These functions provide greater flexibility to a programmer.

The random module

The random module allows you to generate random numbers. The random number helps in many ways. There are many different applications that use random numbers such as password generator and random samples. In this chapter, we will learn the random modules that will help us build the applications.

Random functions for integers

In this section, we will learn the function that deals with integers.

randint()

The function `randint()` generates the random number between the start and end range, including start and end. See the following syntax of the `randint()` function:

```
random.randint(start, end)
```

Let us understand the `randint()` with help of examples:

```
>>> import random
>>> random.randint(5,15)
7
>>> random.randint(5,20)
9
>>> random.randint(5,15)
10
>>>
```

The preceding `randint()` is returning an integer within the range. Let's see what happens when we don't give the range:

```
>>> random.randint(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: randint() missing 1 required positional argument: 'b'
>>>
```

If integer start and end are not defined, then an error will be raised.

randrange()

It returns the random number selected from a range `(Start, stop, step)`. The default value of start is `0` and the default value of step is `1`. If we introduce the step integer, then it returns the value according to the following condition:

```
Start+step*n<= stop
```

The `n` can be any integer.

See the following syntax of `randrange()`:

```
random.randrange(start, stop ,step)
```

The default value of `start` and `step` are `0` and `1`, respectively:

```
>>> random.randrange(10)
7
>>> random.randrange(10)
0
```

We are getting random values between `0` to `10`:

```
>>> random.randrange(10,30)
23
>>> random.randrange(10,30)
15
>>> random.randrange(10,30)
17
```

Now, we are getting values between `10` and `30`.

Let's introduce the third-argument step:

```
>>> random.randrange(10,30,2)
12
>>> random.randrange(10,30,2)
26
>>> random.randrange(10,30,5)
25
>>> random.randrange(11,30,5)
11
>>> random.randrange(11,30,5)
11
>>> random.randrange(10,30,5)
15
>>> random.randrange(10,30,5)
15
>>> random.randrange(10,30,5)
25
>>> random.randrange(11,30,5)
16
>>>
```

You can see, we are getting the results according to the following condition:

```
Start+step*n<= stop
```

Random functions for sequence

In this section, we'll discuss the function related to a sequence.

Choice(rnlist)

The `choice()` function returns the random value from the sequence.

See the following syntax:

```
random.choice(rnlist)
```

Here the random value would be generated from the rnlist list.

```
>>> random.choice((1,2,3))
2
>>> random.choice('abdbc')
```

```
'b'
>>>
>>> random.choice([3,4,5])
4
>>> random.choice({1:"a",2:"b"})
'a'
```

You can also use a tuple and string.

shuffle()

The `shuffle` function shuffles the items in the list.

See the following syntax of the `shuffle()` function:

```
random.shuffle(list1)
list1 is the list provided by the user.
Let's discuss the example.
>>> list1 = [14,12,56,90]
>>> import random
>>> list1 = [14,12,56,90]
>>> random.shuffle(list1)
>>> list1
[56, 90, 14, 12]
```

Note: The shuffle function changes the original list.

Sample()

The `sample()` function is used to obtain a sample from the sequence (string, list, tuple).

See the following syntax of the `sample()` function:

```
random.sample(sequence, length)
```

A sequence can be a string, list, or tuple.

The length specifies the length of the sample to be obtained.

Consider you have a list of the heights of children from different schools and you want to pick a sample that must be unbiased. In this situation, we will use the `sample()` function shown as follows:

```
>>> import random
>>> class_5th = [3,4,3,4,3,4,3.5,5.1,5.3,4,5,4.2,4.9,5,5.5,4,5.4,4,4.5,4
.6,4.7,5.5,4.6,5.6,4,5]
>>> sample1 = random.sample(class_5th, 5)
>>> print (sample1)
[4.9, 5.5, 5.3, 4.6, 5]
>>>
```

Random functions for floats

In this section, we'll see the random functions for floats.

random()

This function returns the float value between 0 and 1. See the following examples:

```
>>> import random
>>> random.random()
0.3532376202318188
>>> random.random()
0.7606196756783162
>>>
```

Uniform(start, end)

This function returns a float value between start and end ranges.

See the following examples for more understanding:

```
>>> random.uniform(3,10)
7.766192509269688
>>> random.uniform(3,10)
6.331910461269013
>>> random.uniform(3,10)
7.629416621046077
>>>
```

Exercise

In this section, we will see some practical use of random modules.

The Tombola game

In this exercise, we will create a random number generator with the following condition:

- The number must be int
- Ranges from 1 to 100
- Non-repeatable

See the following code:

```
import random
list1 = list(range(1,101))
random.shuffle(list1)
for each in list1:
  print (each)
  input("Press enter to get next value ")
```

See the output in the following screenshot:

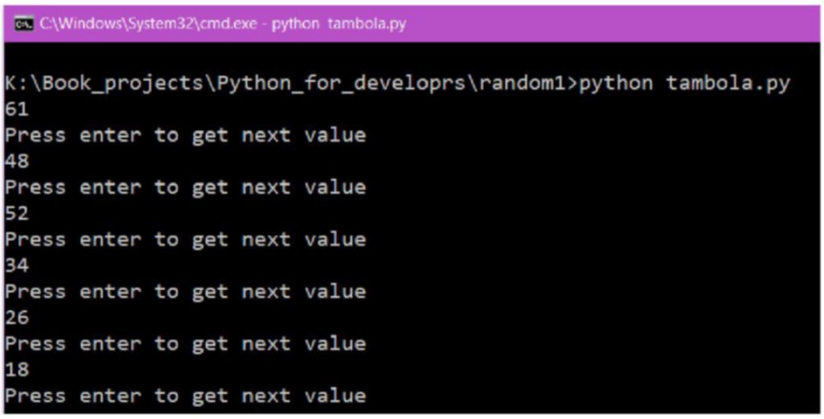

Figure 12.1

The preceding program is working fine.

The OTP generator

In this exercise, we will generate a random OTP number. See the following conditions:

- The OTP must contain the numbers and letters
- The OTP length should be 6

See the following code:

```
import random
def password_gen():
  str1 = "1234567890"
  str2 = "abcdefghijklmonpqrstwxyz"
  list1 =  random.sample(str1,3)
  list2 = random.sample(str2,3)
  list1.extend(list2)
  random.shuffle(list1)
  otp = "".join(list1)
  return otp

print (password_gen())
```

Check the output in the following screenshot:

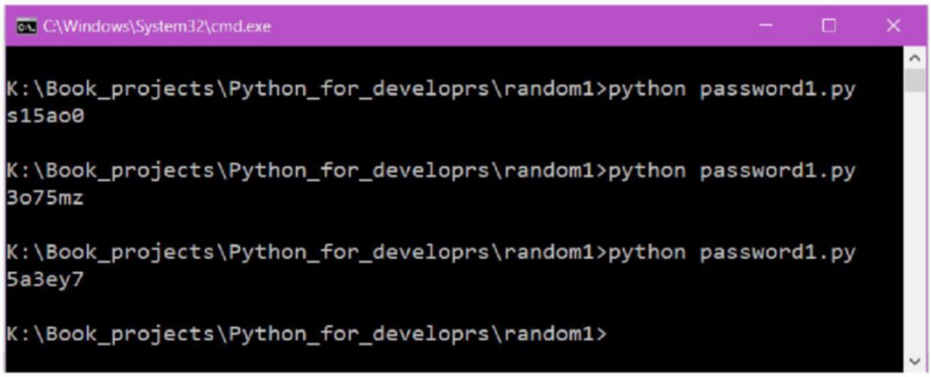

Figure 12.2

Python special functions

In this section, we will discuss some built-in functions of Python, which may be very helpful for making a program.

Lambda

The lambda function does not have the def statement. Its anonymous functions can be used for small tasks. Let's examine the syntax and see an example:

```
lambda arg1 ,arg2,...argn : expression
```

The `arg1`, `arg2` are the arguments and the expression returns value after calculation.

Let's discuss the examples of the `lambda` function:

```
l = lambda x : x**2
print (l(9))
```

In the preceding piece of code, the square of the given number is calculated:

```
l = lambda a,b : a+b
print (l(3,5))
```

In the preceding piece of code, the addition of two numbers has been performed.

See both the output in the following screenshot:

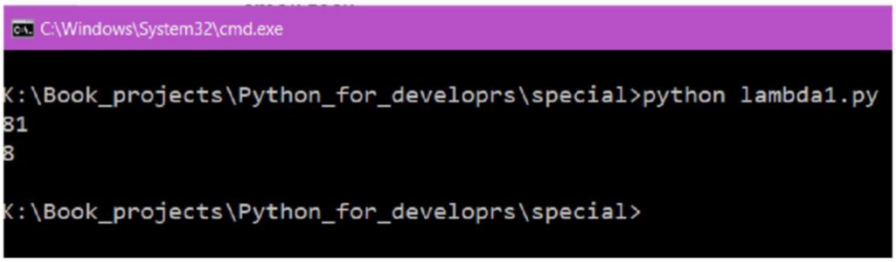

Figure 12.3

The `lambda` can take any number of arguments and returns only one expression.

filter()

The `filter()` function is an easy way to create a filtered list. See the following syntax of the `filter` function.

```
filter(function1, list1)
```

The `list1` is an iterable here; the filter function filters the `list1` based on `true` or `false` return by the `function1`. If `function1` returns true, it means interpreter retains the `list1` items:

```
list1 = [1,2,4,5,6,7,8]
def fun1(x):
  t = x%2
  if t==0:
    return True
  else :
    return False
```

```
res=filter(fun1,list1)
print (list(res))
```

See the output of the following code in the following screenshot:

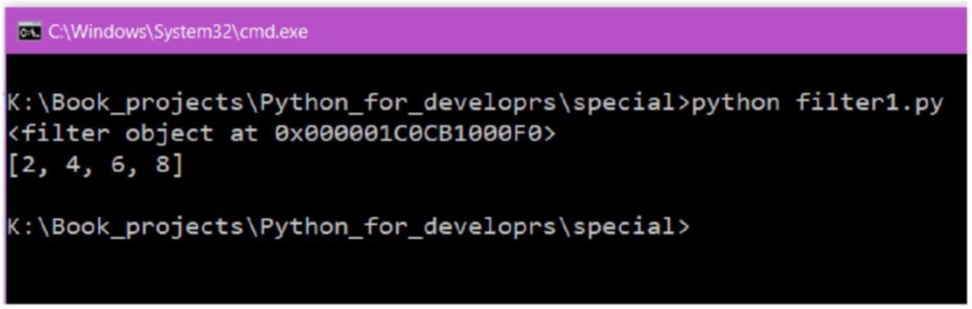

Figure 12.4

The return value of a `filter` function in Python 3 is kind of an iterator type object. You can convert the returned value into list using the `list()` function.

You can also write same code with the help of the `lambda` function. Check out the following code:

```
list1 = [1,2,4,5,6,7,8]
list2 = filter(lambda x: x%2==0, list1)
print (list(list2))
```

map()

The `map()` function is used to map the value. It retuned a new iterator object that contains mapped value after applying the given function to each item of a given sequence (iterable).

See the following syntax of the `map()` function:

```
map(function1, iterable)
```

Each item from the iterable (maybe list, tuple) transfer to the function `function1` and the function `map()` returns an iterator (can be converted to list) with mapped values. Let's see the example to map Celsius to Fahrenheit:

```
list1 = [0,10,37, 38, 45]
def fun1(temp_c):
  f = 9/5*temp_c+32
  f = round(f,2)
```

```
  return f

resp = map(fun1, list1)
print (resp)
print (list(resp))
```

See the following output:

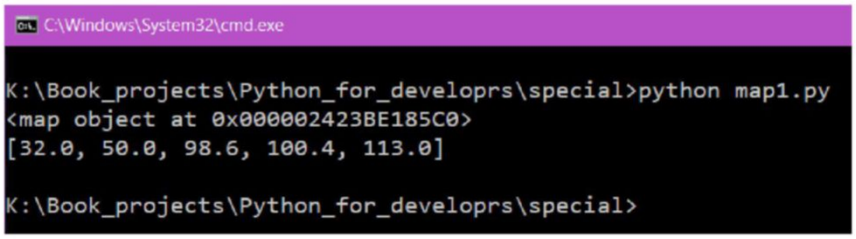

Figure 12.5

Like `filter`, the `map` function also returns an iterator kind of object. Like `filter`, we can use the `lambda` function in the `map()` also. See the following code:

```
list1 = [0, 10, 37, 40 ,45]
a=map(lambda temp1:9.0/5*temp1+32, list1)
print (list(a))
```

reduce()

The reduce() function accepts a sequence and reduces it based on the function specified as an argument. See the following syntax:

`reduce(fun1, list1)`

In the preceding function, `fun1` is the function, and based on the output of the function `fun1`, the `list1` sequence gets reduced:

```
from functools import reduce
list1 = [136,12,37, 8, 45]
def fun1(x,y):
  c = x-y
  return c


resp = reduce(fun1, list1)
print (resp)
```

See the output of the following code:

```
K:\Book_projects\Python_for_developrs\special>python reduce1.py
34

K:\Book_projects\Python_for_developrs\special>
```

Figure 12.6

The reduce function reduces the list `[136,12,37, 8, 45]` as shown in the following figure:

```
[136,12,37, 8, 45]
136-12
[124,37,8,45]
124-37
[87,8,45]
87-8
[79,45]
79-45
34
```

Figure 12.7

I hope you get the sense of the `reduce` function.

isinstance()

Consider a situation where you have a list of heterogeneous values, and you want to perform the calculation on the integer value:

```
list1 = [12,3,4.7,5,6,'2', 'a', 'b', 5.6 ,6, 7]
bw = 20
for each in list1:
  try :
    kpi = each/bw*100
    print (kpi)
  except Exception as e :
    print (e )
```

The preceding program is working fine, but logically, the `try` block also allows the float values. However, we want to calculate only integer values. We need some condition checking system. The `isinstance()` function takes two arguments, first

is an object and second is a class. If an object is the instance of the given class, then `isinstance()` returns `True`, `False` otherwise:

```
list1 = [12,3,4.7,5,6,'2', 'a', 'b', 5.6 ,6, 7]
bw = 20
for each in list1:
  try :
    if isinstance(each, int):
      kpi = each/bw*100
      print (kpi)
  except Exception as e :
    print (e )
```

See the output in the following screenshot:

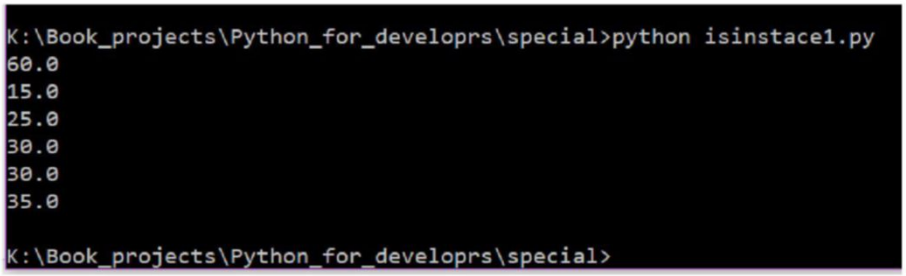

Figure 12.8

It is always advisable to check the type of object before doing any calculation on the object.

eval()

The `eval()` function evaluates the expression passed as an argument. Let's see the syntax, then we will discuss it with the help of examples:

```
eval(expression)
```

In the preceding syntax, the expression must be in the string form.

Let's see some examples:

```
>>> exp = "12+10*2"
>>> exp
'12+10*2'
>>> eval(exp)
32
```

In the preceding example, the exp is a string; the `eval` function removes the quotes and calculates the result. Let's discuss a new case:

```
>>> a = 10
>>> eval("a")
10
>>> eval("b")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'b' is not defined
>>>
```

In the preceding example, a is a variable and `eval()` takes the string a as an argument. The `eval()` function removes the quotes (") and evaluates it. Consider you have a list, but the `list` is in a string form shown as follows:

```
>>> list1 = [1,2,3]
>>> list1 = str(list1)
>>> type(list1)
<class 'str'>
>>> list1
'[1, 2, 3]'
>>>
```

The `list1` looks like a list, but it is a string. To convert that into the list again, we can use the `eval` function:

```
>>> list1 = eval(list1)
>>> list1
[1, 2, 3]
>>> type(list1)
<class 'list'>
>>>
```

Now, you can see, the list1 is a list, not a string.

repr()

The `str()` and `repr()` both return a string object. The goal of the `repr()` function returns the official string representation of a passing object. The goal of the `repr()`

function is to go unambiguous and display the passing object, mainly used for debugging purpose, whereas `str()` displays a representation of the passing object. Let's understand this through the following example:

```
>>> str1 = "Mohit"
>>> str(str1)
'Mohit'
>>> repr(str1)
"'Mohit'"
>>>
```

In the preceding example, the `str()` function is taking a string object as an argument, which returns the same thing, whereas the `repr()` function puts the quotes and displays the object as it is. Let's take a sample datetime module:

```
>>> import datetime
>>> print (str(datetime.datetime.now()))
2019-09-14 22:08:58.605730
>>>
```

The `str()` function returns a readable form of the object. See the following example of `repr()` function.

```
>>> print (repr(datetime.datetime.now()))
datetime.datetime(2019, 9, 14, 22, 9, 5, 941337)
>>>
```

But the `repr()` function returns the object as it is.

I hope you got the sense of the `repr()` function.

Conclusion

In this chapter, you have learned the random module. The random module contains a variety of functions which can be applied on int, float, string, list, and tuple. The purpose of the random module is to generate a random value without any biasing. The random module allows us to build much application such as OTP, password generator, and ludo game. You have also learned the special types of function that gives the programmer great control and helps to speed up the programming. The filter function filters the value based on a given condition; similarly, map function returns the object that contains mapped values. The reduce function reduces the given sequence based on the condition specified in the given function. The eval function evaluates the expression. Sometimes, we got the list in the form of a string. The eval function helps to convert that string into the list again.

Questions

1. What are the return type of filter and map functions?
2. How to do shuffling using a sample function?
3. How to get random numbers between 1 and 100?
4. How to use lambda with filter and map functions?

Chapter 13
Time

We often get scenarios where we deal with time-based events. For example, when we talk about the performance of code, we consider the time elapsed to execute the code. Sometimes, we deal with dates such as current date previous date, for example, how many records have been entered in the past five days. In order to deal with these type of situation, we use the time module in Python. Python offers several modules to deal with time-related things. In this chapter, we'll learn to handle the time-related situations.

Structure

- The time module
    - Current Epoch time
    - Current time
    - Conversion of epoch to human readable format
    - Creating time difference
    - Conversion of Human readable to epoch time
    - time.sleep()
- The Datetime module
- Dealing with Timezone using the Pytz module
- The calendar module

Objective

In this chapter, you'll learn the time modules. With the help of the time module, we'll produce the current time and epoch time. We'll learn different types of human-readable format. We'll see some handy modules such as datetime to produce the current time and time differences. In the end, we will learn how to deal with time zones.

The time module

In this section, we'll see the time module. The time module helps to create the epoch time, conversion of epoch to human-readable, and human-readable to epoch. Most of the functions defined in time modules call the functions of the C library with the same name. Let's see what functions are offered by the time module. See the following command of the interpreter:

```
>>> import time
>>> dir(time)
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'altzone', 'asctime', 'clock', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime', 'mktime', 'monotonic', 'monotonic_ns', 'perf_counter', 'perf_counter_ns', 'process_time', 'process_time_ns', 'sleep', 'strftime', 'strptime', 'struct_time', 'thread_time', 'thread_time_ns', 'time', 'time_ns', 'timezone', 'tzname']
>>>
```

In the preceding functions, we will use most of them.

Current Epoch time

In Python, the time module offers the time method to produce a current epoch. A big question here: what is epoch?

The epoch time, also known as Unix timestamp or POSIX time, is the total number of seconds (int or float) that have elapsed since January 1, 1970 (midnight UTC/GMT). The Epoch time is a number that is continuously increasing. This epoch is useful to compare two time units as almost every language supports the epoch time. In Python, we use the following syntax:

```
import time; time.time()
```

For more details on Epoch time, you can check its official website **https://www.epochconverter.com/.**

Check out the following syntax to find the current epoch:

```
>>> import time
>>> time.time()
1568564149.7536805
>>>
```

When I pressed enter after typing `time.time()`, then it returns the time, in seconds, have elapsed since 1 Jan 1970 (GMT).

Current time

In order to produce the current time, we use `time.ctime()`. See the following syntax:

`time.ctime(secs)`

The `time.ctime()` function converts the seconds or epoch to a 24-character string of the following form: '`Tue May 21 22:53:12 2019`'. If no argument is provided, then `time.ctime()` returns the current time.

Let's understand by the following examples:

```
>>> import time
>>> time.ctime()
'Tue May 21 22:54:00 2019'
>>>
>>> time.ctime(1212121)
'Thu Jan 15 06:12:01 1970'
>>>
```

Conversion of epoch to a human readable format

Let's see how to convert a given epoch time to a human-readable format:

- First convert the given epoch into nine tuple format or `time_struct` using the `localtime()` function shown as follows:

  ```
  >>> t = time.time()
  >>> t1 = time.localtime(t)
  >>> t1
  time.struct_time(tm_year=2019, tm_mon=5, tm_mday=21, tm_hour=22,
  tm_min=46, tm_sec=42, tm_wday=1, tm_yday=141, tm_isdst=0)
  ```

- Convert the nine tuple format to a human-readable form. To convert, we will use `time.strftime()` function. The `time.strftime()` takes two arguments, first is a desired date directive format and second is the time in the `struct_time` format or tuple format shown as follows:

```
>>>
>>> time.strftime("%b %d %Y %H:%M:%S", t1)
'May 21 2019 22:46:42'
```

I used `%b, %y, %Y`. These are the directives. Check out the following section for directives:

- `%a`: Weekday name in an abbreviated form, for example, Mon, Sun, and so on.
- `%A`: Complete weekday name in capital case.
- `%b`: Month name in abbreviation such as Sep and Jan.
- `%B`: Complete month name in capital case.
- `%C`: Century number only, for example, 20, 19.
- `%d`: Day number of the month (01 to 31).
- `%D`: Return the time in the form of /month number/day number/century number.
- `%e`: Day number of the month (1 to 31).
- `%y`: Year number without the century.
- `%H`: Hour number according to a 24-hour clock (00 to 23).
- `%I`: Hour number based on a 12-hour clock (01 to 12).
- `%j`: Day number of the year (001 to 366).
- `%m`: Month in number (01 to 12).
- `%M`: Minute(s).
- `%p`: According to the provided time, it will return AM or PM.
- `%r`: Same `%I:%M:%S %p`, for example, '`10:49:46 PM`'.
- `%R`: Return time in the "`%H:%M`" format.
- `%S`: Second(s).
- `%T`: Current time, same as `%H:%M:%S`.
- `%u`: Weekday in number (1 to 7), Sunday=7.
- `%W`: Week number of the year specified in the given time, beginning with the first Monday of the first week.
- `%w`: Weekday in number (1 to 7), Sunday=0.
- `%x`: Same as `%m/%d/%y` without the time.
- `%X`: Same as `%H:%M:%S`.
- `%y`: Year number without a century (range 00 to 99).
- `%Y`: Year number, including the century (2019).
- `%Z`: Time zone name (India Standard Time).
- %z: Time zone in plus or minus hours (+0530).

If you give abbreviated month names such as Aug and Jan, then the %b directive would be used.

Creating time difference

Producing the time difference is very easy. Let's add one hour; an hour contains 3600 seconds, so add 3600 seconds to the current time:

```
>>> import time
>>> now1 = time.time()
>>> t1 = now1+ 3600
>>> t1
1558463563.593892
>>> time.ctime(t1)
'Wed May 22 00:02:43 2019'
>>>
>>> time.ctime(now1)
'Tue May 21 23:02:43 2019'
>>>
```

Conversion of human-readable date to epoch time

In this section, we will convert the human-readable date to epoch time. We'll use the `time.strptime(date, format)` function.

Let's convert "`6-Jan 1987`" to epoch UNIX time stamp.

Let's first convert the date to struct time. See the following code:

```
>>> import time
>>> t= time.strptime("6-jan 1987",'%d-%b %Y')
>>> t
time.struct_time(tm_year=1987, tm_mon=1, tm_mday=6, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=1, tm_yday=6, tm_isdst=-1)
Now convert the struct time to epoch time.
>>> time.mktime(t)
536869800.0
Now you can use following method too.
```

```
>>> import calendar
>>> calendar.timegm(time.strptime("6-jan 1987",'%d-%b %Y'))
536889600
```

The `mktime` and `calendar.timegm` both are giving different output.

Let us check the difference:

```
>>> time.ctime(536889600)
'Tue Jan  6 05:30:00 1987'
>>>
>>> time.ctime(536869800)
'Tue Jan  6 00:00:00 1987'
>>>
```

The `calendar.timegm` converts the time automatically according to timezone.

If you want to produce some delay in the execution, then you can use `time.sleep()`.

time.sleep(second)

The `time.sleep()` function suspends the execution of the current process for the given number of seconds. See the following code and then check the output:

```
import time
print (time.ctime())
time.sleep(20)
print (time.ctime())
```

The output is shown in the following screenshot:

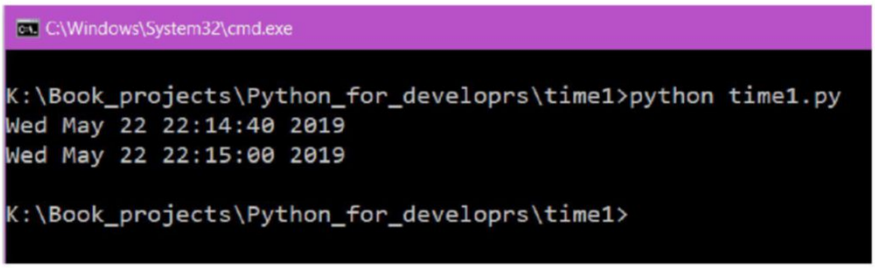

Figure 13.1

There are some other modules that are useful and contain handy methods.

The datetime module

According to Python docs, *"The datetime module supplies classes for manipulating dates and times in both simple and complex ways".*

So, datetime modules contains several classes. Let's discuss one by one:

`datetime.datetime.today()`

The `datetime.datetime.today()` method returns today's date. See the following example:

```
>>> import datetime
>>> print (datetime.datetime.today())
2019-10-15 22:26:45.244975
>>>
```

datetime.datetime.now()

The `datetime.datetime.now()` method returns the current time with date:

```
>>> print (datetime.datetime.now())
2019-10-15 22:26:50.492970
>>>
```

The output of `datetime.datetime.now()` is the same as produced by `datetime.datetime.today()`, however, the `datetime.datetime.now()` method can produce the time of a different time zone. We will see this feature in the pytz module.

datetime.timedelta class

With the help of the `datetime.timedelta` class, we can perform addition and subtraction on datetime to produce a new date.

The class `datetime.timedelta` accepts keyworded arguments. According to pydocs, all arguments are by default kept 0. Arguments may be floats, ints, and longs, in positive or negative form.

Let's see the syntax of timedelta:

```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0,
minutes=0, hours=0, weeks=0)
```

Let's create the time difference:

```
>>> datetime.timedelta(days=2, hours=2)
datetime.timedelta(days=2, seconds=7200)
```

You can see how hours are converted to seconds. From the syntax, it is obvious that all arguments are default arguments with the value `0`. The `timedelta` class only stores the days, seconds, and microseconds. The passed argument gets converted in the following way:

- A millisecond gets converted to 1000 microseconds
- A minute gets converted to 60 seconds
- An hour gets converted to 3600 seconds
- A week gets converted to 7 days

Let's create time differences using `datetime.timedelta` class:

Produce the current time shown as follows:

```
>>> import datetime
>>> t1=datetime.datetime.now()
>>> print (t1)
2019-05-22 22:58:16.614310
>>>
```

Create a timedelta of 2 days and 2 hours later and previous as shown here:

```
>>> d1 = datetime.timedelta(days=2, hours=2)
>>> d2 = datetime.timedelta(days=-2, hours=-2)
>>>
>>> t2 = t1+d1
>>> print (t2)
2019-05-25 00:58:16.614310
>>>
>>> t3 = t1+d2
>>> print (t3)
2019-05-20 20:58:16.614310
>>>
```

You can easily conclude the time difference between current time, `t2`, and `t3`. If you want to convert time delta into number of seconds. You can take the help `d1.total_seconds()`, where `d1` is the `timedelta`.

Dealing with Timezone with the pytz module

The pytz modules is used to create the time of different timezones.

Let's create the current time of India and US. Check out the following series of code:

```
>>> import datetime
>>> import pytz
>>> print (datetime.datetime.now(pytz.timezone('US/Eastern')))
2019-05-22 13:51:35.448140-04:00
>>>
>>> print (datetime.datetime.now(pytz.timezone('Asia/Kolkata')))
2019-05-22 23:21:59.950024+05:30
>>>
```

Now, we are using `pytz` with `datetime`. In order to obtain the time of particular time-zone, you must know a string such as '`Asia/Kolkata`'.

Let's consider you don't know the string of timezone. We'll explore the `pytz` module. See the following screenshot:

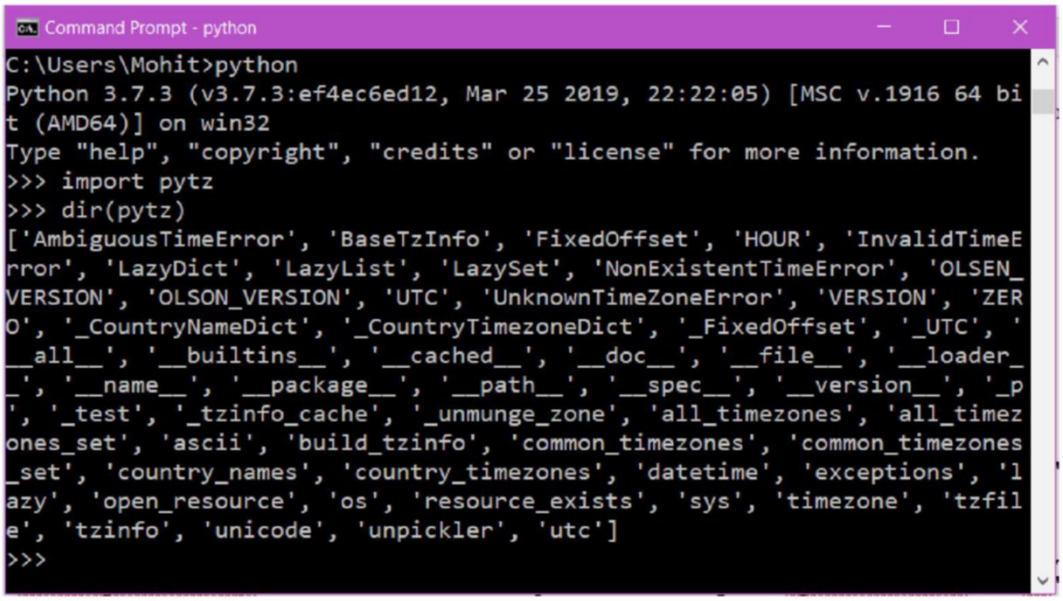

Figure 13.2

The timezone method we have already used:

```
>>> pytz.country_names
<pytz._CountryNameDict object at 0x000001A4A9D44DA0>
```

It is dictionary like object:

```
>>> pytz.country_names.items()
ItemsView(<pytz._CountryNameDict object at 0x000001A4A9D44DA0>)
```

See the following screenshot for more clarification:

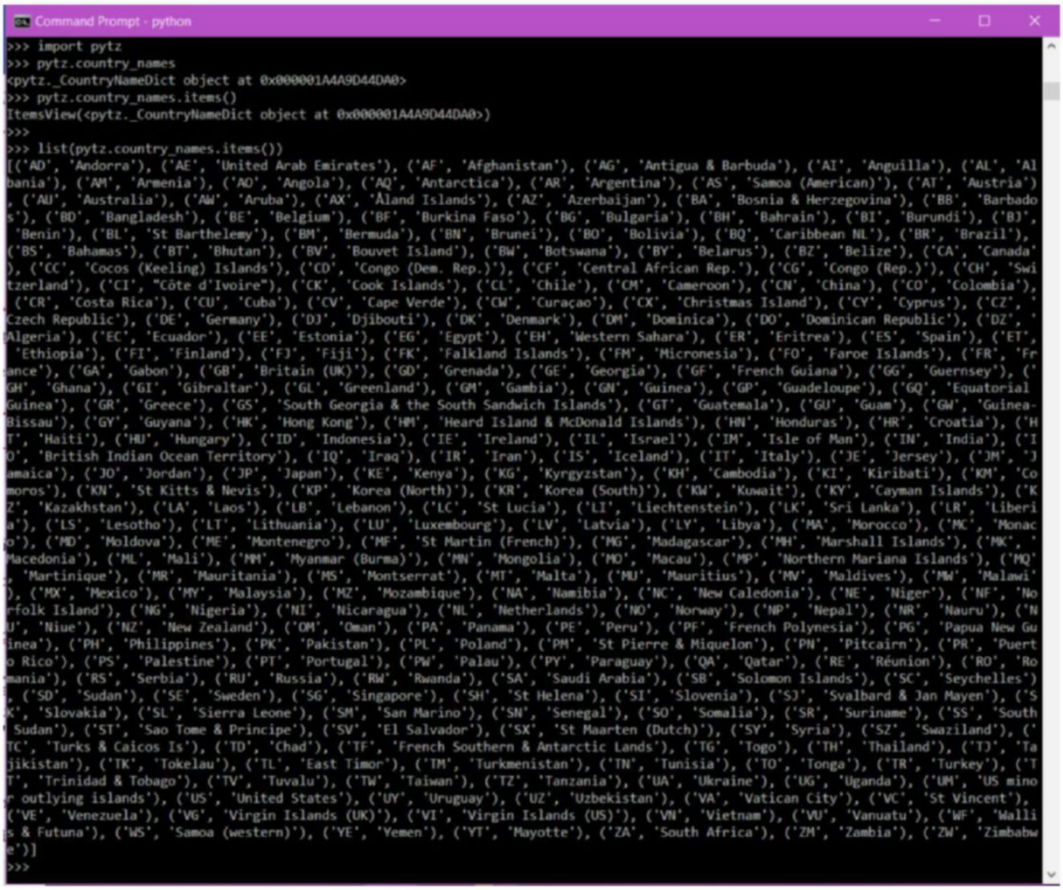

Figure 13.3

In this way, you can obtain the abbreviation with a country name.

Now, if you are certain about the abbreviation, then you can check the time zone using an abbreviation as shown in the following commands:

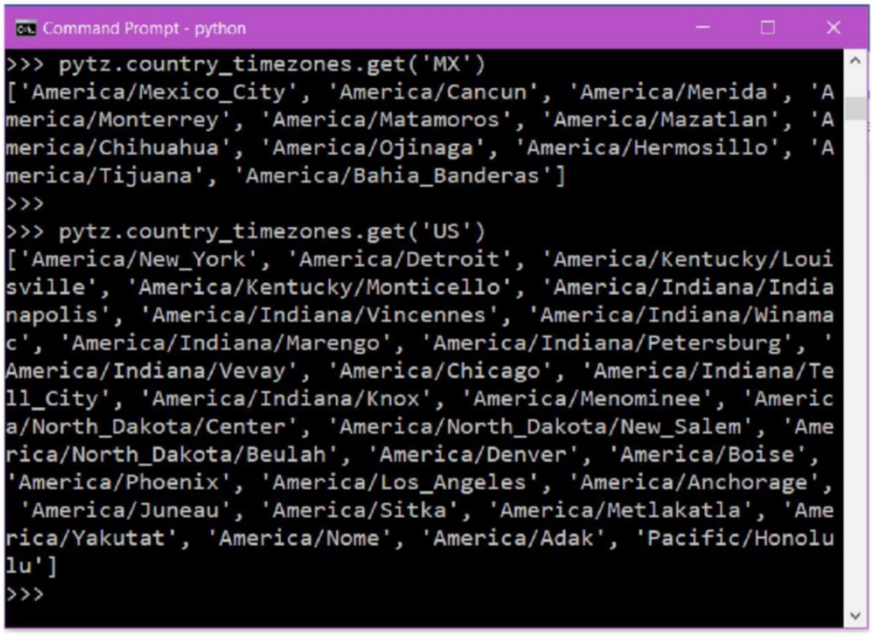

Figure 13.4

The preceding code shows the timezones of Mexico and US:

```
>>> import datetime
>>> print (datetime.datetime.now(pytz.timezone("America/Mexico_City")))
2019-05-24 12:00:27.763500-05:00
>>>
```

In this way, you can get time of any timezone of any country.

Let's complete one exercise.

Print all the timezones and the current time of US.

See the following code:

```
import pytz, datetime
print ("MY Time", datetime.datetime.now())
for tz in  pytz.country_timezones("US"):
  print (tz.ljust(30) ,datetime.datetime.now(pytz.timezone(tz)))
```

Let us check out the output in the following screenshot:

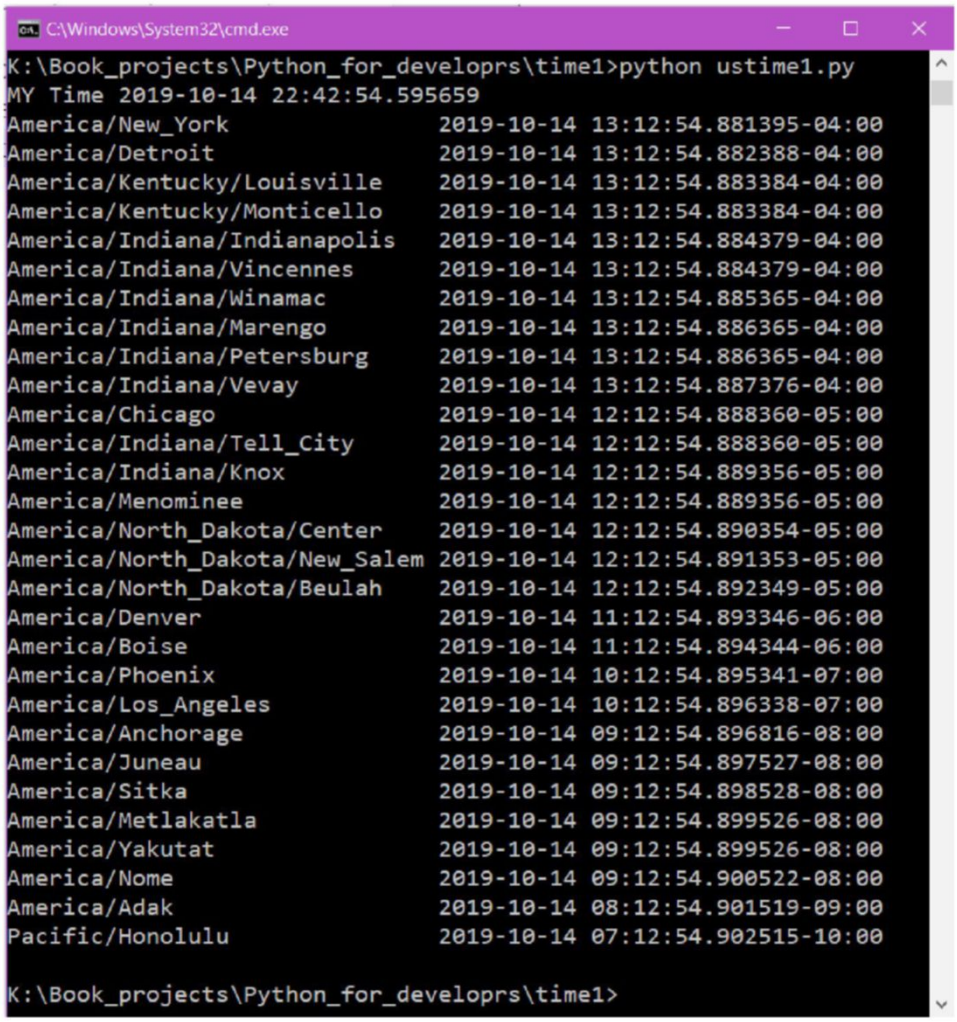

Figure 13.5

The calendar module

In this module, we'll learn the methods offered by the calendar module. With the help of this module, we can print any month, year, and even detect the leap year. Let's use some methods one by one.

Printing a full month

To print a full month, we take the help of the `month()` method. The `month()` method takes two arguments: year and month. See the following screenshot:

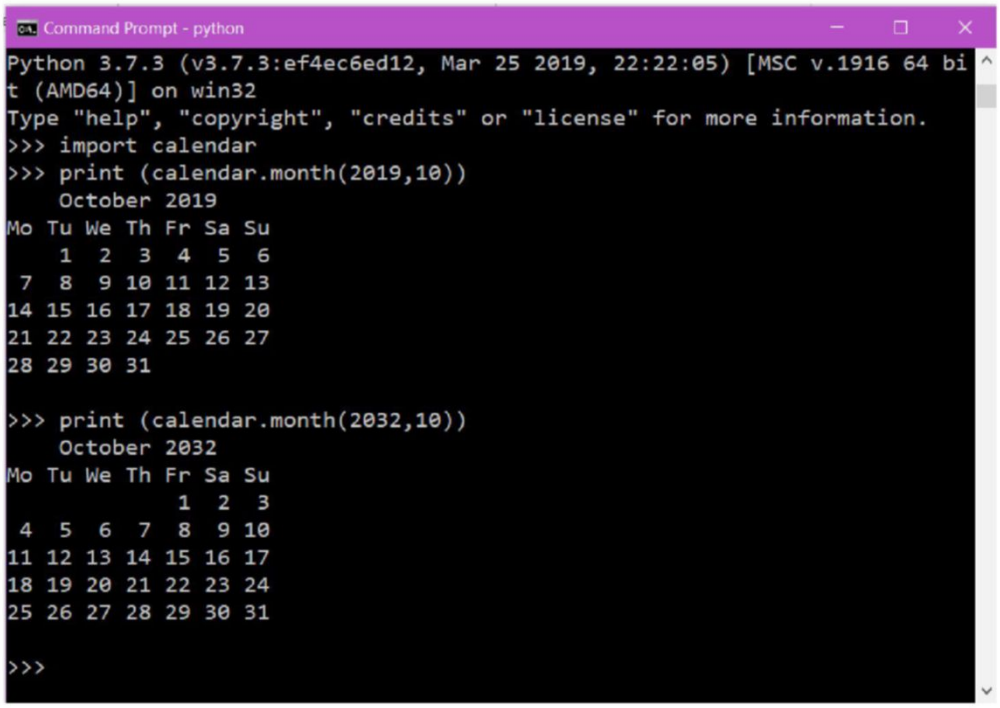

Figure 13.6

By providing a year and month, we can print any month.

Printing a year

With the help of the `calendar()` method, we can print a complete year. The `calendar()` method just takes one argument, that is, the year. See the following screenshot:

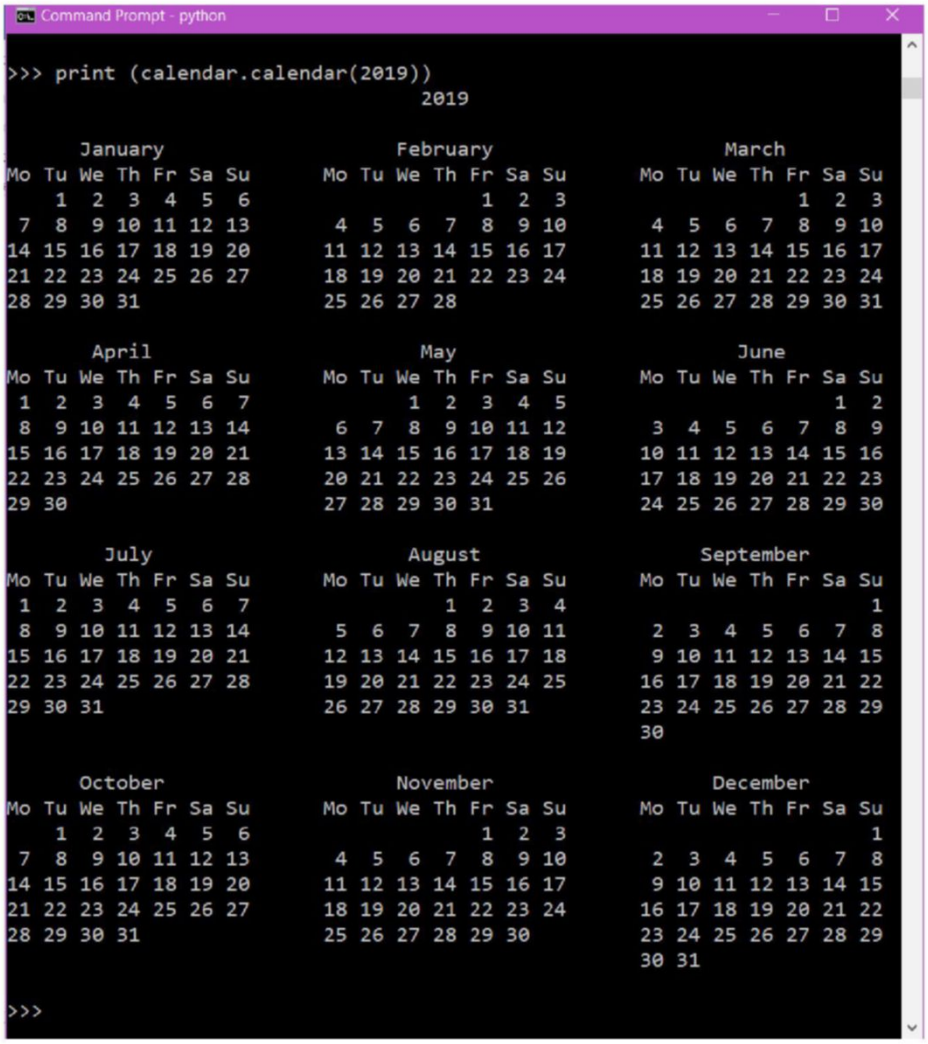

Figure 13.7

Curious case of 1752

In this section, we'll see the exciting case of 1752. Before jumping to the story, let's quickly print the calendar of 1752. See the following screenshot for the 1752 calendar:

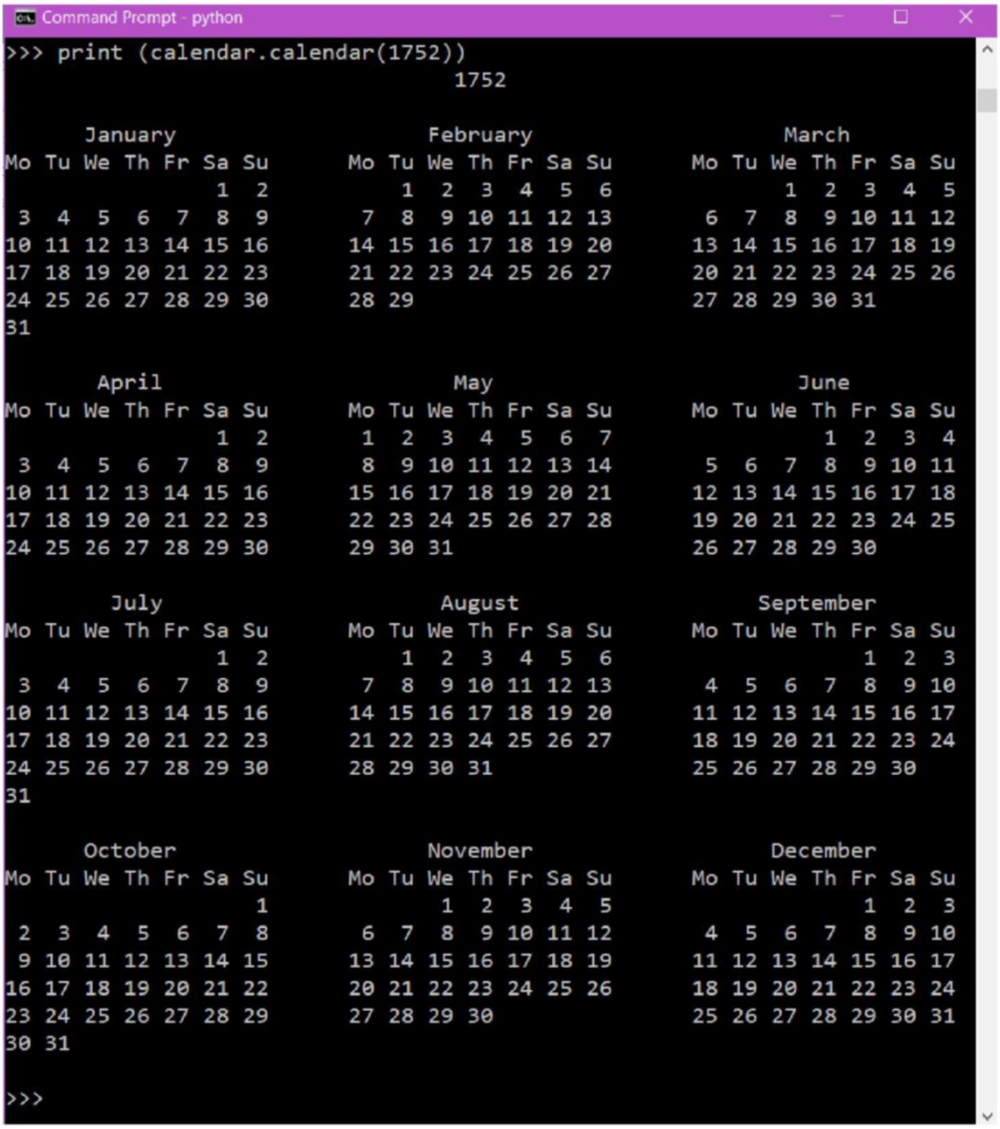

Figure 13.8

Now, print the calendar 1752 in Linux as well. See the following screenshot from Linux:

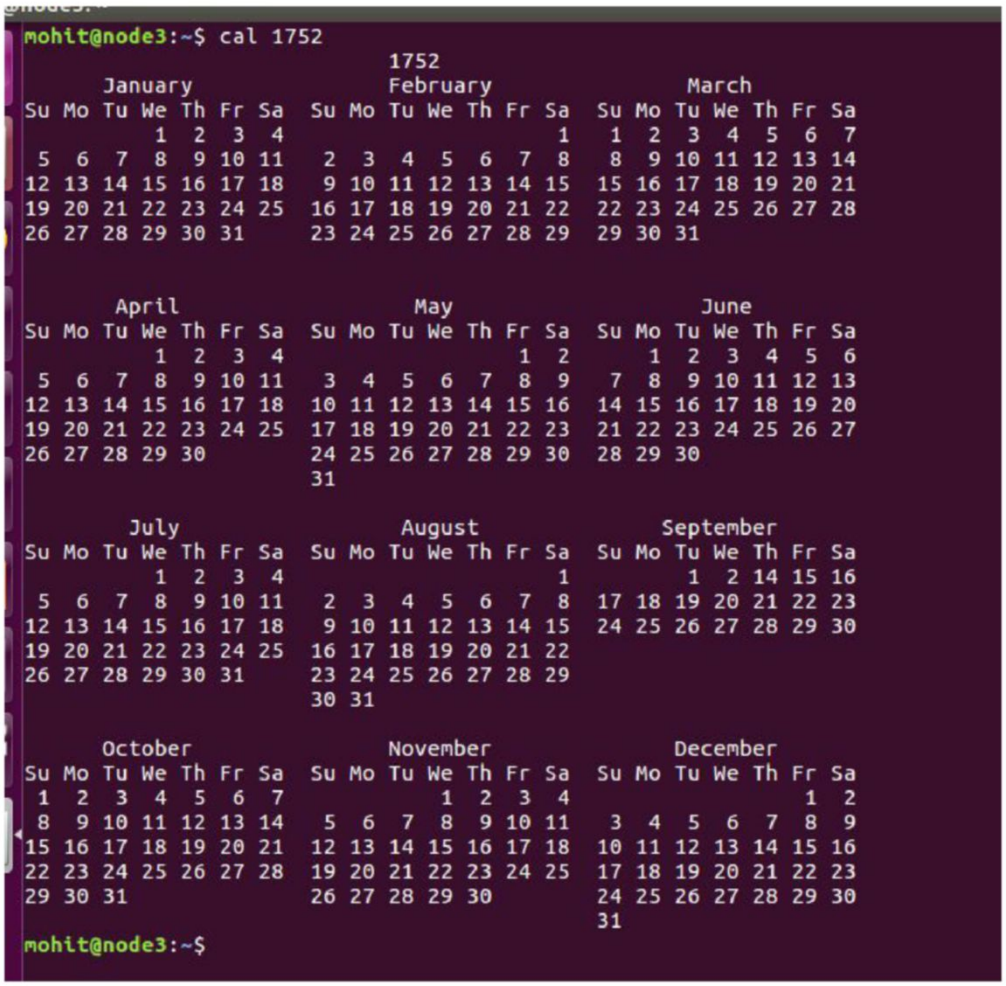

Figure 13.9

Please check the first day and last of both the calendars. The Python-generated year is showing, Saturday is first day. The Linux calendar shows Wednesday is the first day. But the last day (31 Dec) of both the calendars is Sunday. So, where is the problem? Check the month of September of the Linux-generated calendar. A total of 11 days are missed out. This is not any Linux error. Currently, we are using the Gregorian calendar, before September 3, 1752; two calendars were prevalent—one was Gregorian, and the other was Julian. The Julian calendar was fractionally too long and the Gregorian calendar was accurate with astronomy calculation. In 1752, in favor of the Gregorians, Britain decided to give up the Julian calendar. Thus, September 3 immediately got converted to September 14.

Checking the leap year

If you want to check whether a specified year is a leap year or not, then we can use the `isleap()` method.

See the following examples:

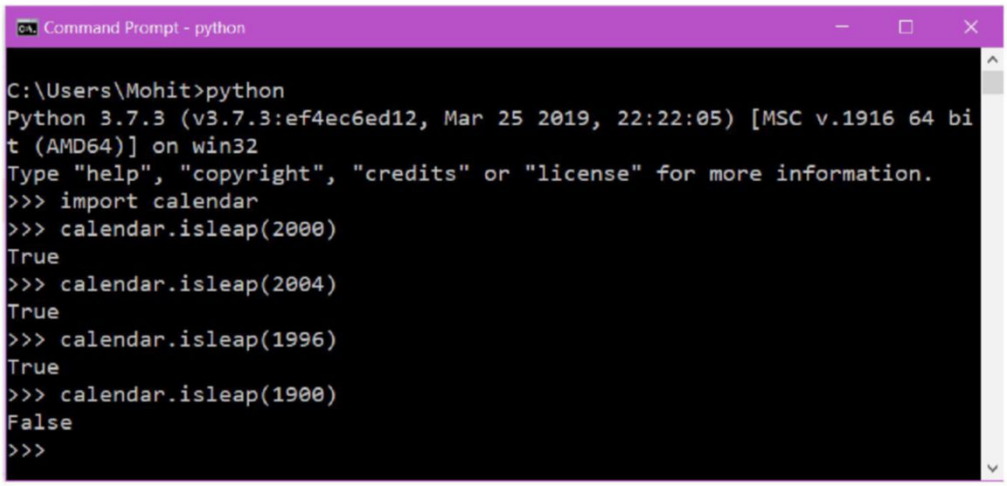

Figure 13.10

You can also check the leap days between a range of years. Check out an example demonstrated in the following screenshot:

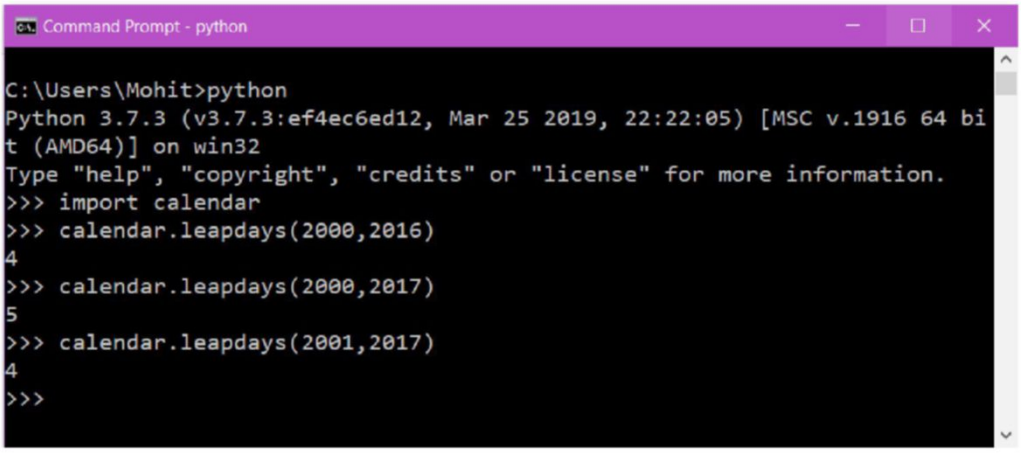

Figure 13.11

In the preceding examples, the method `leapdays()` returns the total number of days between the year range. In the range, the last year is not included.

Conclusion

In this chapter, you have learnt how to deal with time-related things. You have learned epoch time, its significance, and conversion of epoch to a human-readable format. There are some directives that help to convert epoch to a human-readable format. `ctime()` is the built-in function to convert epoch to human-readable format. The `time.sleep()` can be used to produce a delay in the execution. The `datetime` is a handy module that has cool methods such as display the current time and time difference. The pytz module allows you to create the time of different zones. The calendar module helps to produce the calendar of any month or year. In the next chapter, we'll learn the regular expression to search the patterns.

Questions

1. Display the current time and the date two months later.
2. Display the current time of New Zealand.
3. Consider a list of date ['2015-01-24', "2015-09-23", "2016-03-21", "2018-01-11"].
4. Get the date which is greater than "2017-01-01".
5. Which time module function is used to delay an execution?
6. Create a small program and find out the time taken by the program.

Chapter 14
Regular Expression

Sometimes we have to search something using patterns, such as, a file which starts with "intel" and ends with ".pdf", a movie starts with "avengers" and ends with "mp4". We often encounter such problems. In order to deal with such questions, we use a regular expression. The regular expression allows us to make a pattern to search for the desired data. For example, you want to search emails from a web page and you cannot predict the exact email names to be searched. In this case, we make a regular expression of the email and fetch all the emails. In this chapter, we will learn about the regular expression and the special characters that help in forming a regular expression.

Structure

- Regular expression
- Regular expression function
    - `match()`
    - `search()`
    - `sub()`
    - `findall()`
- Special characters

Objective

In this chapter, you will learn that the functions belong to the regular expressions. We will see the different types of functions to find the given regular expression. After functions, we will learn how to make a regular expression. To make a regular expression, we will learn about the special characters.

Regular expression

A regular expression is a text or string to identify a search pattern. The Python re-module provides the regular expression support. See the following examples:

1. Find the string which contains `mohit`, must end with pdf
2. IP address from the text
3. Email finding from the text

We will discuss regular expression and searching functions one-by-one.

Regular expression functions

In this section, we will learn about the regular expression, which helps in finding the regular expression patterns. First we will discuss the `match()` function.

match()

The `match()` function searches the given pattern from the given string at the beginning only. If the match is found, then it returns the match object; otherwise it returns None. See the following syntax:

```
re.match(pattern, string, flags=0)
```

The parameters are explained as follows:

- `pattern:` The regular expression to be matched
- `string:` Given a text in which pattern is matched
- `flags:` The flag, we will discuss later with example

Let see the example:

```
import re
str1 = "Tony: I am Iron-Man"
p = "to"
m = re.match(p,str1,re.I)
print (m)
if m :
```

```
  print (m.group())
```

In the preceding code, `re.I` means case insensitivity. To find the matched pattern, we use the group method, as showcased in the preceding code. See the output in the following screenshot:

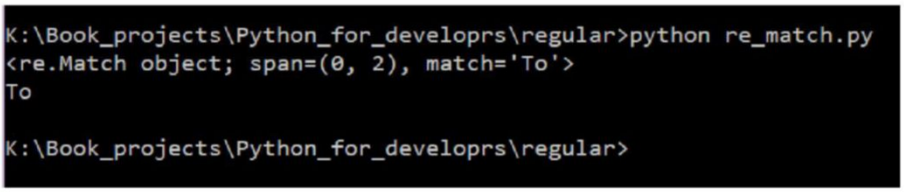

Figure 14.1

The next method is search, which searches the pattern throughout the string.

search()

The `search()` function searches the pattern's first occurrence in the given string. If the pattern is found, the function returns an object. With the help of the `group()` method, the matched pattern can2be retrieved. See the following syntax:

```
re.search(pattern, string, flags=0)
```

The pattern is the regular expression, which needs to be searched, and the string is the text we need to search the pattern from. The flag will be discussed later:

```
import re
str1 = " SOMETIMES YOU GOTTA RUN BEFORE YOU CAN WALK."
p = "you"
m = re.search(p,str1,re.I)
if m :
  print (m.group())
  print (m.start(), m.end())
```

In the preceding code, the `m.start()` finds the starting point of matched pattern and `m.end()` exposes the endpoint of the matched pattern. See the output in the following screenshot:

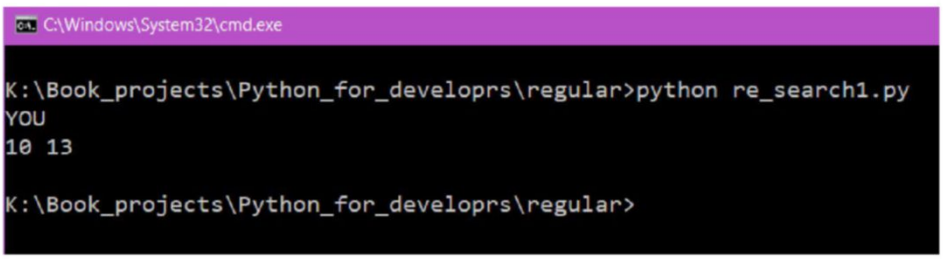

Figure 14.2

sub()

The `sub()` method is used to replace the sub-string with a given pattern. See the following syntax:

```
re.sub(pattern, to_be_replaced, string, max=0)
```

The `sub()` method replaces the occurrence of the RE pattern in the string with the `to_be_replaced` sub-string. The value of the max argument decides how many replacements need to be done; the default value is 0; it means all the occurrences will be replaced:

```
import re

str1 = "SOMETIMES YOU GOTTA RUN BEFORE YOU CAN WALK."

p = "YOU"

m = re.sub(p,"I", str1)

print (m)

m1 = re.sub(p,"I", str1,1)

print (m1)
```

Let us see the result in the following screenshot:

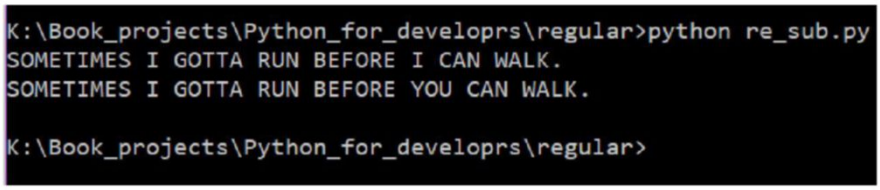

Figure 14.3

In the first example, all the fields get replaced, and in the second example, only one field is replaced.

findall()

This is the higher version of the search() function. If you want to find all the occurrences of the RE pattern in the string, use the findall() function. See the following syntax:

```
findall(pattern, string,flag=0)


Let us discuss an example:

import re

str1 = "Remember, with great power comes great responsibility"
```

```
p = "great"
m = re.findall(p,str1,re.I)
print (m)
```

In the preceding code, we have used `re.I`, which is a flag that removes the case sensitivity:

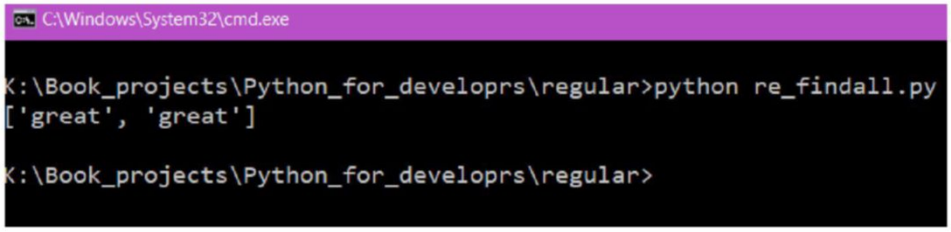

Figure 14.4

The `findall()` function returns a list of the matched pattern. If you want to obtain the position of the pattern, then use `finditer()`, as showcased as follows:

```
import re
str1 = "Remember, with great power comes great responsibility"
p = "great"
m = re.finditer(p,str1,re.I)
for each in m:
  print (each.group(), each.start(), each.end())
```

See the output in the following screenshot:

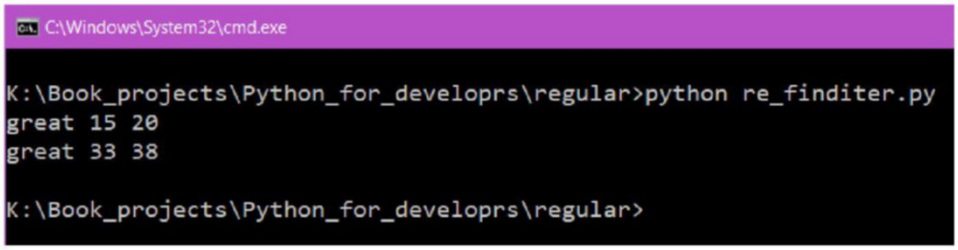

Figure 14.5

The `finditer()` returns an iterator that produces instances as returned by the `search()` function.

re.compile(pattern)

With the help of `compile()`, a search pattern can be converted into a pattern object:

```
prog = re.compile(pattern)
result = prog.match(string)
```

It is equivalent to the following syntax:

```
result = re.match(pattern, string)
```

If we want to use a complex regular expression multiple times, it is a good idea to use its compiled version, since it is more efficient.

Special characters

In Python, there are special characters that help in forming a regular expression. Now we will use `r` in front of a regular expression. Placing `r` or `R` before a pattern, generates, what is referred to as, a raw pattern. The raw pattern does not process the escape sequences (`\n`, `\b`, and so on) and is therefore frequently applied for regular expression patterns, which often contain several \ characters.

. (dot)

The ("`.`") matches any character in the default mode except for a newline. This matches anyny character, including a newline, if the `re`. `DOTALL` flag has been indicated. Let us discuss this with an example:

```
import re
str1 = "ab abbb add akkk adbdf abbbbbb a"
p= r'a.b'
m = re.finditer(p,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
```

See the following screenshot for the output:

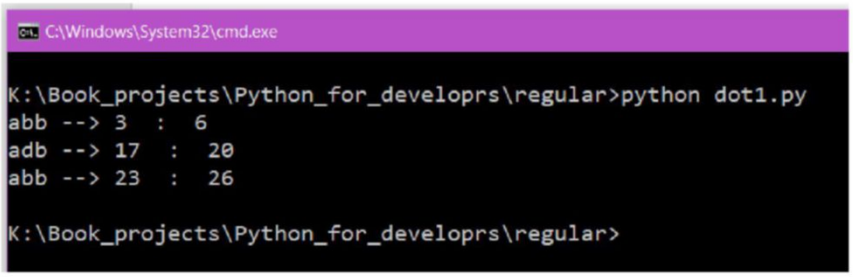

Figure 14.6

The meaning of the preceding expression is `a<any thing only one time>b`, but anything must be present.

^ (Caret)

The (^) matches the beginning of the given string, and if `re.M` (Multiline) flag is set, then it also matches immediately after each new line. See the following examples:

```
import re
str1 = "ab bbb ab"
p= r'^ab'
m = re.finditer(p,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
```

Let us see the result in the following screenshot:

```
K:\Book_projects\Python_for_developrs\regular>python caret.py
ab --> 0  :  2

K:\Book_projects\Python_for_developrs\regular>
```

Figure 14.7

$ (dollor)

The ($) matches at the end of the given string, or just before the newline at the end of the string.

Let us consider that you have a list of files, and you want to search the file(s) that end with txt:

```
import re
str1 = ["hello.txt", "ab.py", "game.py", "yui.jpg"]
p = r'py$'
for file in str1 :
  m = re.search(p,file)
  if m:
    print (file, m.group(),"-->", m.start()," : ", m.end())
```

We are getting all the files that end with py, as showcased in the screenshot:

```
K:\Book_projects\Python_for_developrs\regular>python dollar.py
ab.py py --> 3  :  5
game.py py --> 5  :  7

K:\Book_projects\Python_for_developrs\regular>
```

Figure 14.8

* (star)

The * operator is used in RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. The expression ab* will match a, ab, or a followed by any number of 'b's. In the following section you will see the example of *.

+ (plus)

The + operator is used in RE to match 1 or more repetitions of the preceding RE. ab+ will match a followed by any non-zero number of 'a's; it will not match just a.

Let us discuss with an example:

```
import re
str1 = "ab a abb abb a"
p = r'ab*?'
p1  = r'ab+?'
m = re.finditer(p,str1)
m1 = re.finditer(p1,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
print ("**********\nFor +")
for each in m1:
  print (each.group(),"-->", each.start()," : ", each.end())
```

Let us see the result in the following screenshot:

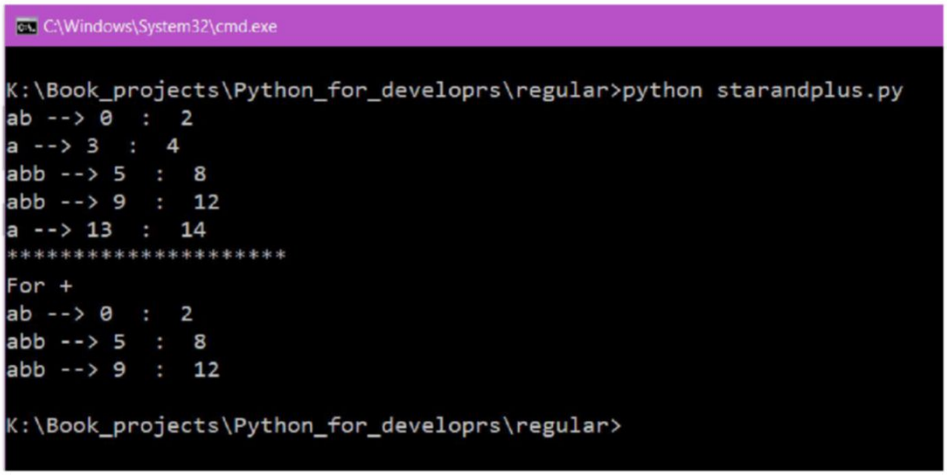

Figure 14.9

The preceding figure clearly showcases the difference between * and +. The pattern of * also accepts the strings that contain only an a.

?

The (?) is used in regular expression to match 0 or 1 repetitions of the preceding RE. The expression `ab?` will match either a, or ab. The * and + qualifiers are greedy; they fit as much text as they can. This action is sometimes not required. If the RE ab* is matched against '`abbbb`', then it will match the entire string and not just ab. Adding (?) after the qualifier, allows it to perform in a non-greedy or minimal order, since it matches as few characters as possible. Using *? in the previous expression will match only ab. Let us discuss with an example:

```
import re
str1 = "ab a abb abb a"
p = r'ab*?'
p1  = r'ab+?'
m = re.finditer(p,str1)
m1 = re.finditer(p1,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
print ("********************\nFor +")
for each in m1:
  print (each.group(),"-->", each.start()," : ", each.end())
```

Let us see the result in the following screenshot:

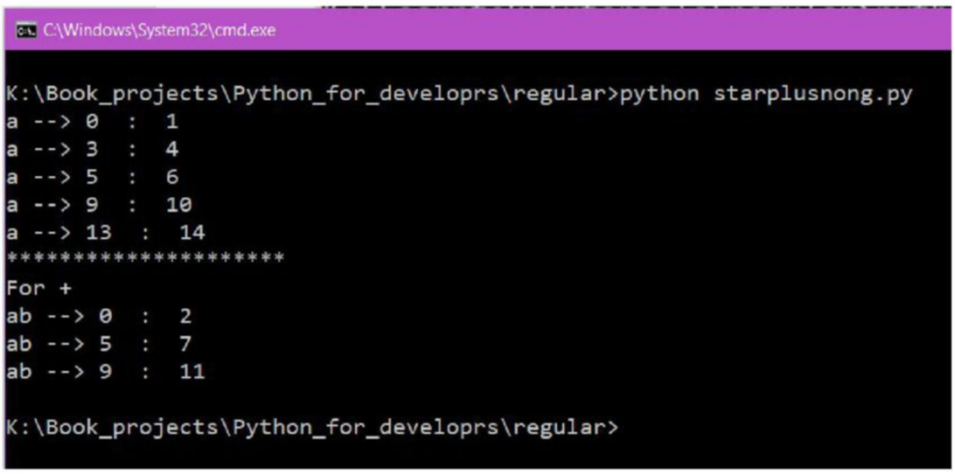

Figure 14.10

The difference is very clear from the previous result.

{m}

The expression {m} specifies that exactly the m copies of the previous RE should be matched. For example, a{3} will match exactly three 'a' characters, but not four. In the next section, you will see the example.

{m,n}

The expression {m,n} matches from m to n repetitions of the preceding RE. For example, a{4,6} will match from 4 to 6 'a' characters. The following example provides clarity:

```
import re
str1 = "ab abbb ac abbbb ak abbbbbb a"
p = r'ab{3}'
p1 = r'ab{3,5}'
m = re.finditer(p,str1)
m1 = re.finditer(p1,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
print ("*********************\nFor {3,5}")
for each in m1:
  print (each.group(),"-->", each.start()," : ", each.end())
```

Let us see the output in the following screenshot:

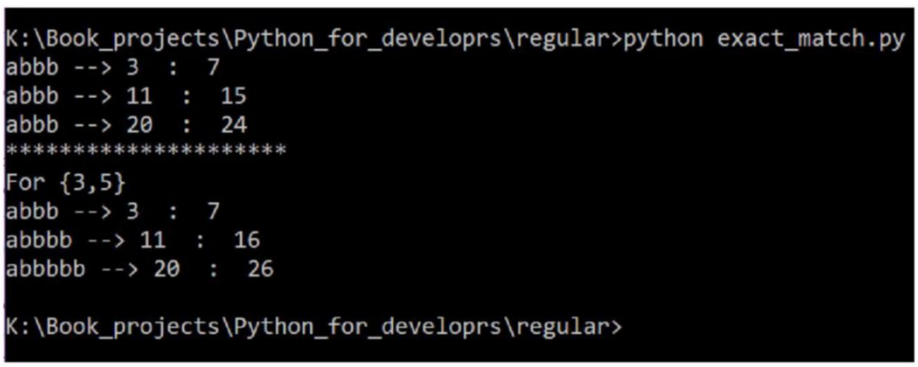

Figure 14.11

[]

The expression `[]` is used to indicate a set of characters.

Characters can be listed individually, for example, `[mn]` will match `m` or `n`.

Character ranges can be stated by providing and separating two characters by a `(-)`. See the following examples:

- `[a-z]` will match any lowercase ASCII letter
- `[0-9]` will match any number
- `[0-5][0-9]` will match all the two-digit numbers from 00 to 59
- `[0-9A-Fa-f]` will match any hexadecimal digit

Let us consider that we want to match an expression that can contain `a` or `z` or -:

```
import re
str1 = "kbce ratyui akl zio -"
p = r'[a-z]'
m = re.finditer(p,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
```

Let us see the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\regular>python escape1.py
k --> 0  :  1
b --> 1  :  2
c --> 2  :  3
e --> 3  :  4
r --> 5  :  6
a --> 6  :  7
t --> 7  :  8
y --> 8  :  9
u --> 9  :  10
i --> 10  :  11
a --> 12  :  13
k --> 13  :  14
l --> 14  :  15
z --> 16  :  17
i --> 17  :  18
o --> 18  :  19
```

Figure 14.12

In the preceding expression, `[a-z]`, the - have been taken as a regular expression, but we want to literal - not as a special character. In this case use escape `\`:

Use the following pattern:

```
p = r'[a\-z]'
```

See the result in the following screenshot:

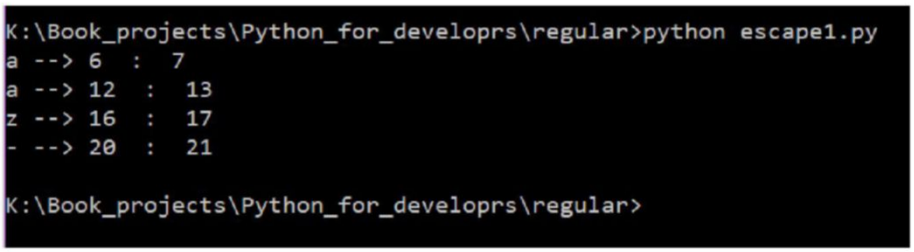

Figure 14.13

[^]

If the set's first character is ^, it matches all the characters that are not in the set. For instance, `[ ^D ]` matches any character other than ' `D` ' and `[ ^^ ]` matches any character other than ' ^'. Let's consider a string that contains a letter, a number and special characters, as showcased below:

```
"ab@#$%^&A*&^%k$345678bn"
```

We want to remove everything, except letters:

```
import re
string = "ab@#$%^&A*&^%k$345678bn"
p = r"[^a-zA-Z]"
resp = re.sub(p,"",string)
print (resp)
```

See the result in the following screenshot:

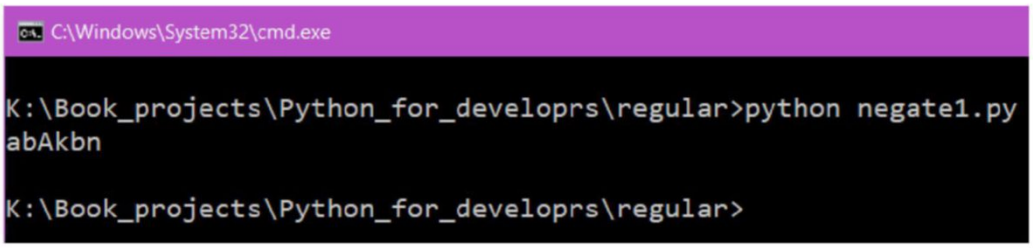

Figure 14.14

\w

The `\w` represent the word. A word is defined as a sequence of alphanumeric or underscore characters. The `\w` is equivalent to the set `[a-zA-Z0-9_]`. So, the end of a word is indicated by a whitespace or a non-alphanumeric, non-underscore character. `\b` is defined as the boundary between `a \w`. For example, `r'\bfoo\b'` matches `'foo'`, `'foo'`, `'(foo)'`, `'bar foo baz'`, but not `'foobar'` or `'foo3'`.

Exercise

In this section, we will perform two exercises. Let us look at each of them, individually:

1. Find the IP addresses from a string:

```
import re
str1 = "hello192.168.0.17878sqsa10.0.1.1"
p = r'[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
m = re.finditer(p,str1)
for each in m:
  print (each.group(),"-->", each.start()," : ", each.end())
```

   See the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\regular>python ip_address1.py
192.168.0.178 --> 5  :  18
10.0.1.1 --> 24  :  32

K:\Book_projects\Python_for_developrs\regular>
```

Figure 14.15

2. Get the email addresses from a webpage.
   In order to get text from a webpage, we will use the `requests` module.
   Consider an email address: `34h-u.lk_123@marvel.com.`
   An email address contains two essential things, `@ and ..`
   Let us define the regular expression:

```
<any letter, number,-,_,.>@<any letter, number,-,_,.>.<any letter>
r"[a-zA-Z0-9_.-]+@[a-zA-Z0-9_.-]+\.[a-zA-Z]+"
```

   Let us see the code:

```
import requests
```

```
import re
url    =    "http://www.thapar.edu/faculties/category/departments/
computer-science-engineering"
r = requests.get(url)
#34h-u.lk_123@marvel.com
p = r"[a-zA-Z0-9_.-]+@[a-zA-Z0-9_.-]+\.[a-zA-Z]+"
string = r.text
resp=re.finditer(p, string)
for each in resp:
  print (each.group())
```

See the following screenshot for the output:

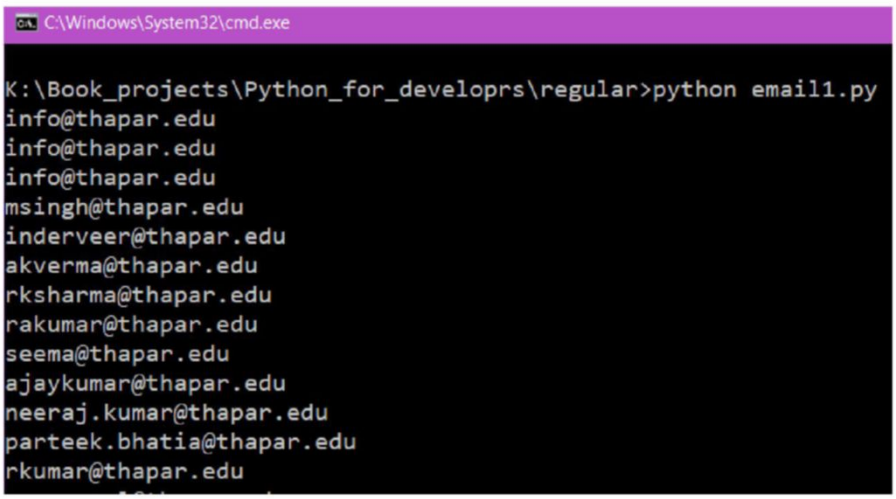

Figure 14.16

The program is working fine. We cannot say that the email's regular expression is 100% correct, but it works 100% correctly for us.

Conclusion

In this chapter, you have learned about the regular expression - we use a regular expression when we don't know the exact string, but know about a certain pattern of the string. We have seen the different functions to search for the pattern. The `match()` function searches the beginning of the string, the `search()` function finds the pattern throughout the string, but returns only the first occurrence. The `findall()` function returns the list of all the occurrences of the pattern found in the

string. The `finditer()` function allows to find the index of the beginning and end of a pattern found in the string. We have learned about the special characters that help in generating the patterns. In the next chapter, you will learn about interaction with the operating system. The OS module allows the user to interact with the operating system. You will learn how to run the OS command with the help of Python.

Questions

1. What is the difference between the findall and finditer functions?
2. Consider that you have a python list of names of songs, find all songs that contain the string “wada” or “waada”.
3. Consider that you have a python list of names of files, find all the files that start with “mohit” and ends with “pdf”
4. What is the meaning of r’[^A-Z]’?

Chapter 15
Operating System Interfaces

The Python OS module allows us to interact with OS. With the help of the OS module, you can perform a lot of exciting tasks. The OS module facilitates you to run any command of OS. You can create a directory, rename a file, and delete a file. You can also find the properties of a file, like date of creation, modification, and last accessed. You can also perform a copy, and move a file from one directory to another directory. In this chapter, you will learn how to use the OS module to create a lot of interesting things.

Structure

- Getting the OS name
- Directory and file accessing functions
    - Directory functions
- File and folder listing
- Executing OS command
- Exercises

Objective

In this chapter, we will study how to interact with OS using the Python command. To communicate with OS, we will have to import the OS Module. We will learn

about the different functions that help to create, delete, and rename the folder and files. We will learn how to run the OS command using the Python program.

Getting the OS name

In this section, we see the ways to get the OS name. Let us consider that we are going to run or execute the OS command using the Python program, but we are not sure about the operating system on which we can run the code. Let us see the different ways to check the operating system.

```
sys.platform
```

```
platform.system()
```

```
os.name
```

First, we will run the above piece of code on Windows, then on Linux:

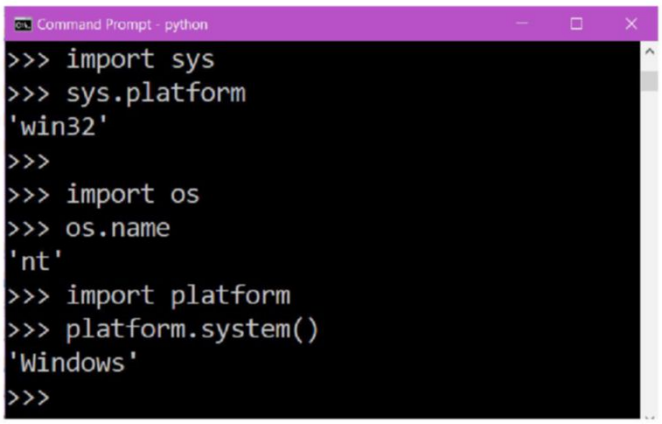

Figure 15.1

Let us run the preceding command on Linux:

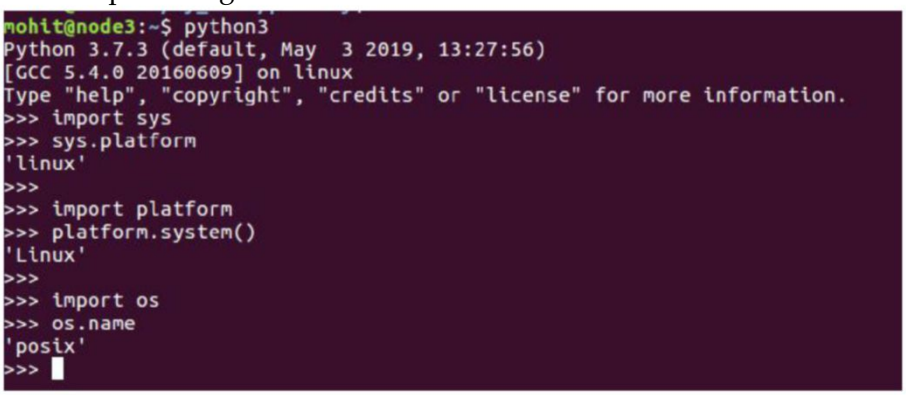

Figure 15.2

You can use any command of your choice to detect the operating system.

os.environ

The `os.environ` gives a variety of information. It returns a dictionary of information. Let us look at the output:

Figure 15.3

Let us check some interesting output of `os.environ`. See the following screenshot:

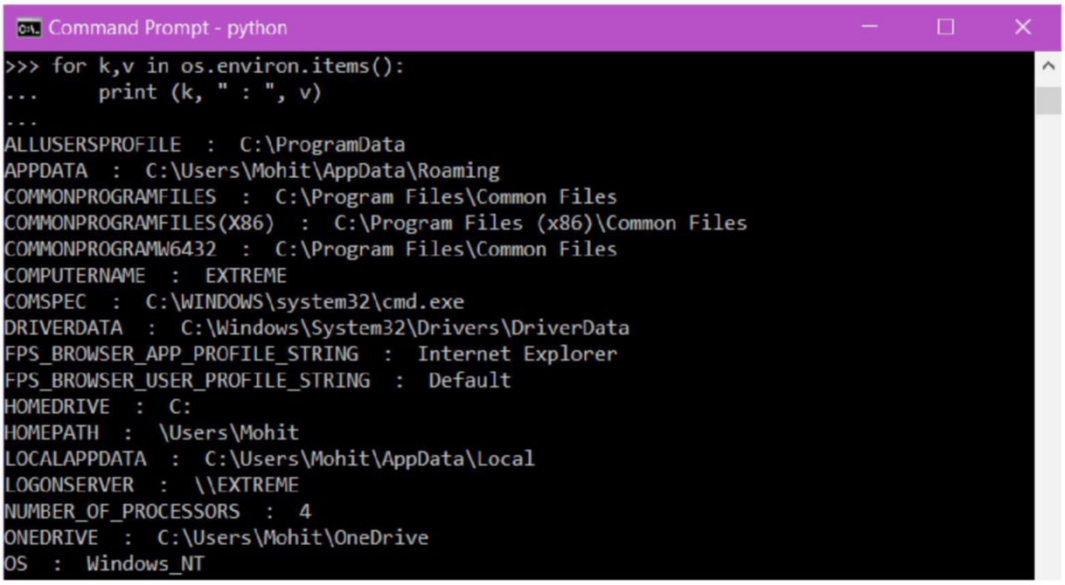

Figure 15.4

We can iterate `os.environ.items()`.The preceding screenshot is showing a piece of information. The `os.environ['COMPUTERNAME']` returns the machine name. But these keys may not be the same for Linux.

Directory and file accessing functions

In this section, we will see the OS functions that deal with file and folders.

os.access()

The `os.access` is used to check different types of authorizations. It is always an excellent exercise to check the authorization on the file before doing any operations. See the following syntax:

```
os.access(path, mode)
```

The parameters are described as follows:

- `path:` Path means file or directory with its path
- `mode:` Mode means read write and execute

There are four types of modes:

- `os.F_OK:` This mode checks whether the file or folder at the specified path exists or not
- `os.R_OK:` This mode checks whether the file is authorized to read

- `os.W_OK`: This mode checks whether the file is authorized to write
- `os.X_OK`: This mode checks whether the file is authorized to execute

```
mohit@node1:~$
mohit@node1:~$ ls -ltr | grep game.py
-rw-rw-r-- 1 mohit mohit    0 Sep 21 22:02 game.py
mohit@node1:~$ python3.7
Python 3.7.3 (default, Jul 13 2019, 12:25:37)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.access("game.py",os.F_OK)
True
>>> os.access("game.py",os.R_OK)
True
>>> os.access("game.py",os.W_OK)
True
>>> os.access("game.py",os.X_OK)
False
>>>
```

Figure 15.5

See the preceding screenshot, the `game.py` file has permission 654, and other users have only read permission. The user `Mohit` is running the Python, and creates the file. That is why, `os.W_OK` and `os.R_OK` are showing True.

os.rename(old, new)

`os.rename()` is used to rename a file. The argument old means the present file name, and new means a new name. See the examples in the following screenshot:

```
>>>
>>> import os
>>> os.chdir("K:\\Book_projects\\Python_for_developrs\\op\\exp")
>>> os.rename("iron.png", "tony.png")
>>>
>>>
```

Figure 15.6

The file `iron.png` is renamed as `tony.png`.

os.stat()

If you want to know the details of a file, like the creation time, modification time and last accessed time, and so on use `os.stat("file_with_path")`. See the example in the following screenshot:

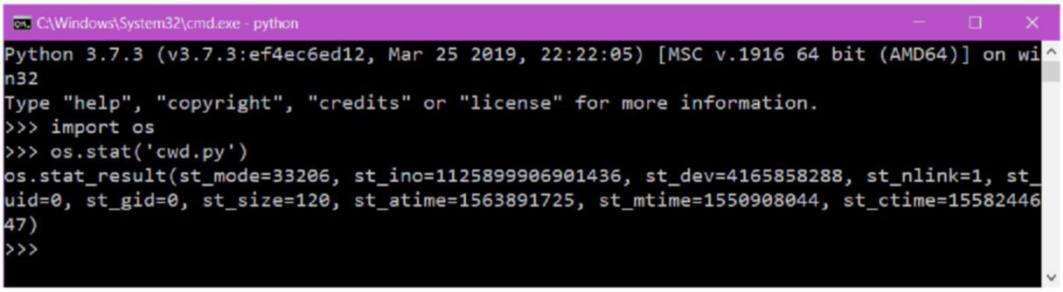

Figure 15.7

Directory functions

There are a couple of functions that deal with the directories.

os.getcwd()

`os.getcwd()` returns the current working directory. It shows that directory from where you run the Python interpreter. See the examples in the following screenshot:

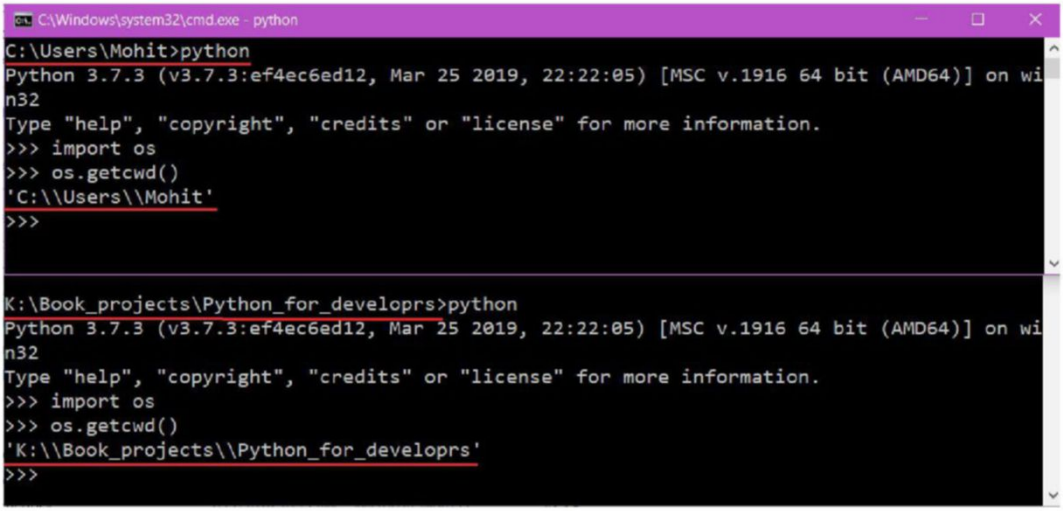

Figure 15.8

In the above examples, the Python interpreter was being run from two different directories, and `os.getcwd()` returns the same directories.

os.chdir()

In the previous example, you saw that the `os.getcwd()` returned the current working directory. However, you can change the current working directory at the run time too. With the help of `os.chdir()`, the current directory can be changed.

See the following example:

```
import os
print ("Current working dir :", os.getcwd())
os.chdir("k:\\" )
print ("Current working dir :", os.getcwd())
```

See the directories in the following screenshot:

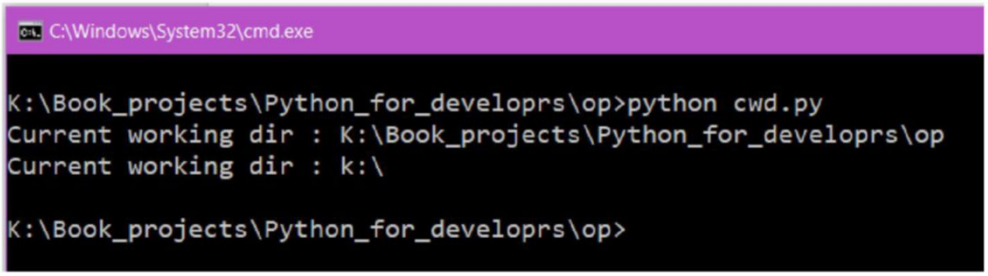

Figure 15.9

To check whether a folder exists or not, use `os.path.isdir("foldername")`. See an example in the following screenshot:

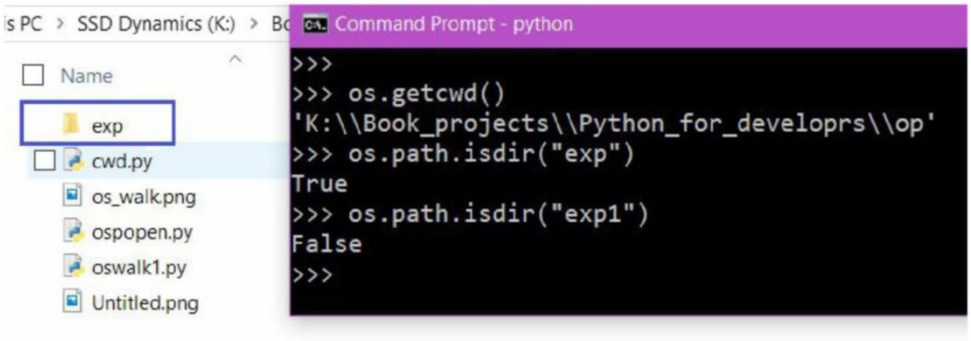

Figure 15.10

Let us consider a folder is not present, and we want to make a new folder. With the help of `os.mkdir("foldername")`, we can make a new folder. See the example in the following screenshot:

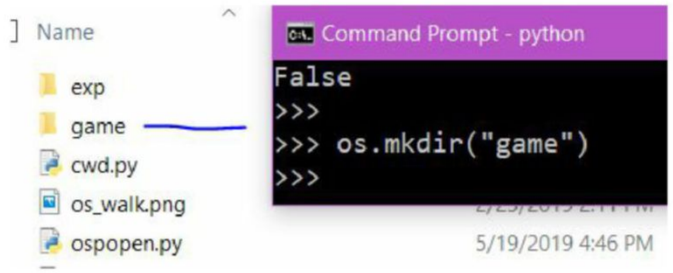

Figure 15.11

After the execution of the code, you can see the folder game.

Let's consider that you want to create a hierarchy of the folder, like `hit/qwe/flag`. If you use `os.mkdir()`, then it returns an error. In order to create a directory hierarchy, we use `os.makedirs()`. See the example in the following screenshot:

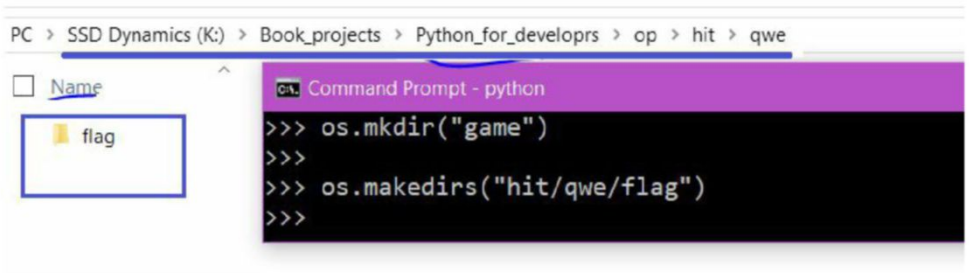

Figure 15.12

To remove the file, you can use `os.remove(file_name)`. The method does not remove the directory. To remove the directory, use `rmdir(dir_path)`.

File and folder listing

In this section, we will see how to list all the files and folders in a specific folder.

os.listdir(path)

The `listdir()` returns a Python list that contains the files and folders present at the path, specified as an argument. See the examples in the following screenshot:

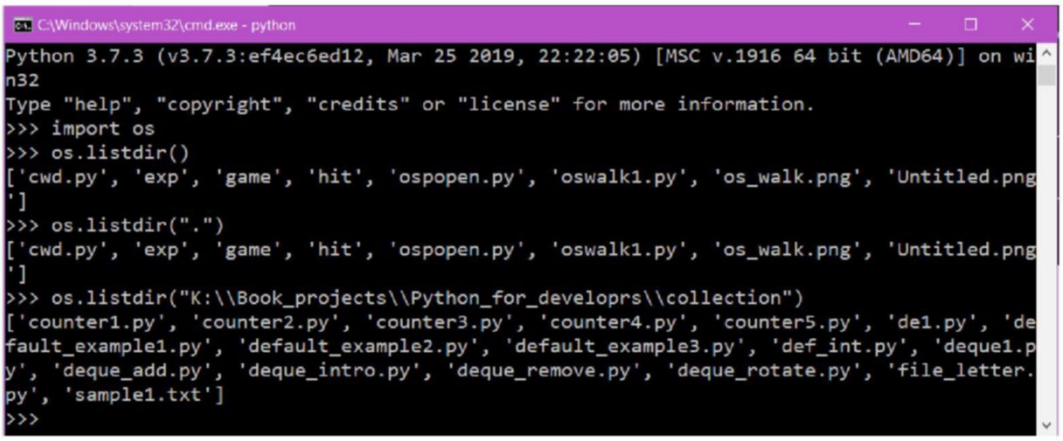

Figure 15.13

If you are already at the destined path, then use only ., which denotes the current directory path. If the path is not specified, then it takes . (current directory) as the path.

os.walk()

We have seen the `os.listdir()`, which returns the list of files and folder; however, this `os.listdir()` does not search the folder and files recursively. If you want to explore the folder and files recursively, use `os.walk`.

`os.walk()` is used to generate the list of directories and files in a top-down or bottom-up manner. See the following syntax of `os.walk()`:

```
os.walk(top, topdown=True, onerror=None, followlinks=False)
```

The top is the top directory given by the user; the top contains the root, sub-directories, and files. Here, the root is the topmost directory provided by the user. Sub-directories are the directories contained by the root, files are the files contained in the root directory and its subdirectories:

- `if topdown=True:` It means that the directory search starts from a top directory then goes to the next directory. In other words, top to down manner.
- `if topdown=False:` It means that the directory search begins from the last directory, then goes to the top directory. In other words, bottom to up manner.
- `error:` This can show the error to continue with the walk, or raise the exception to abort the walk.

- `follow links:` The search follows symlinks, pointed by symlinks if set to true.

Before discussing the example, let's see the directory structure:

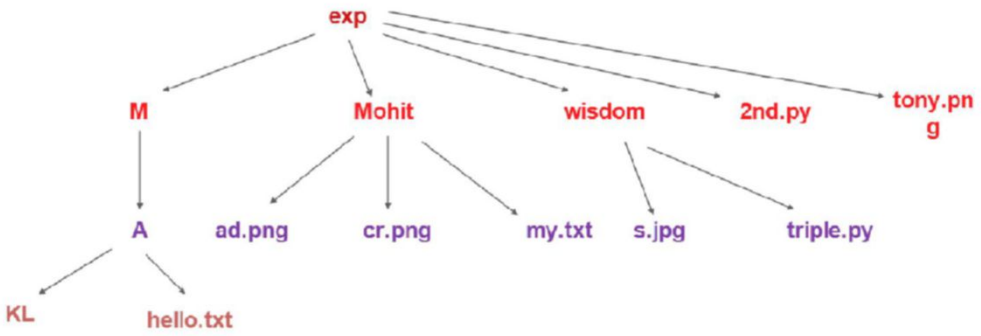

Figure 15.14

The `K:\Book_projects\Python_for_developrs\op\exp` is the top directory, and `M`, `Mohit` and `wisdom` are the subdirectories. The rest are the files.

Let us see the program:

```
import os

for root, dirs,files in os.walk("K:\\Book_projects\\Python_for_developrs\\
op\\exp"):
  print (root)
  print (dirs)
  print (files)
  print ("*"*60)
```

The code returns a tuple, which contains three things - the top directory, directories of the top directory and files of the top directory.

See the following output:

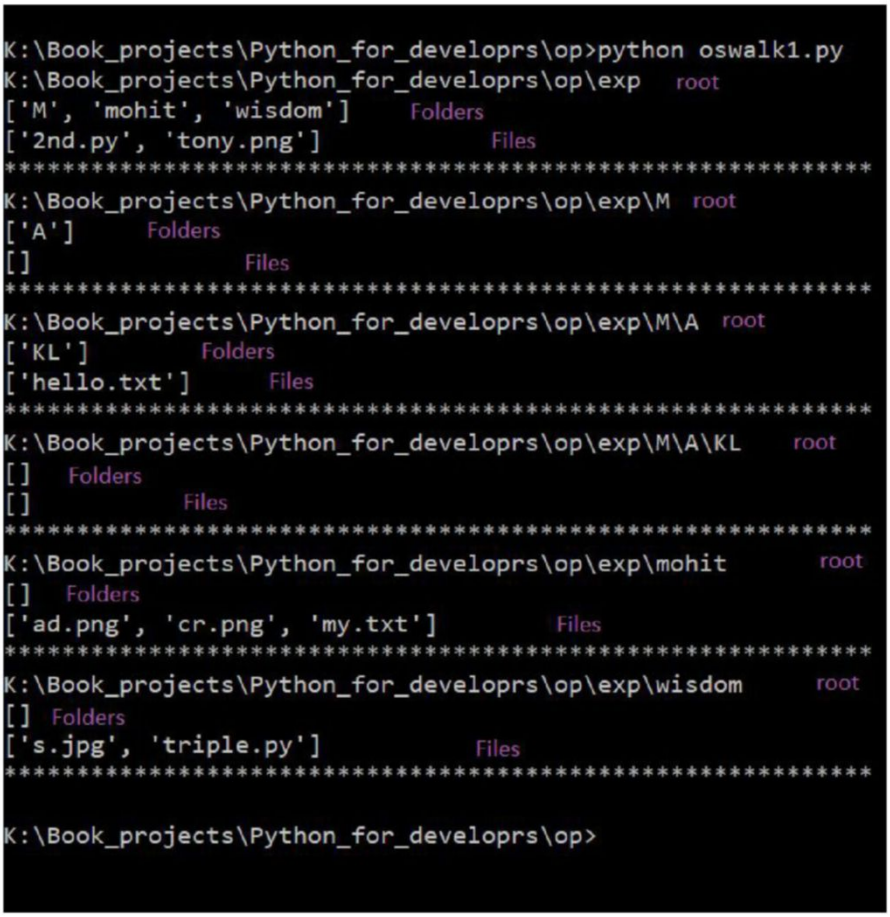

Figure 15.15

In the preceding figure, the root, folders, and files are labeled.

Executing OS command

In this section, we will see the different ways to execute the OS commands.

os.popen()

Let's consider that you don't know how to perform a specific task in the operating system through the Python program. You can use the `os.popen()` to run any OS command, in that case. See the following syntax:

```
os.popen(command, mode, bufsize)
```

`os.popen(command)` takes a DOS or shell command passed in as a string and returns a file-like object connected to the command's standard input or output streams.

The second argument is the mode, it can be `r` (default) or `w`.

The optional `bufsize` argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means using a buffer of (approximately) that size (in bytes). A negative buffering means using the system default:

```
import os
list_ip= ["127.0.0.1", "facebook.com", "thapar.edu"]
cmd = "ping -n 1 "
for domain in list_ip:
  cmd = cmd + domain
  resp = os.popen(cmd)
  text=resp.read().lower()
  if "ttl" in text:
    print (domain,  "is live")
  else :
    print (domain, "is not live")
```

See the following screenshot for the output:

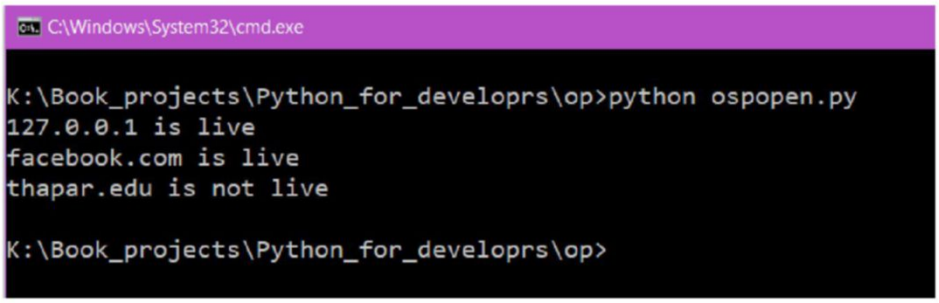

Figure 15.16

The code checks the `ttl` string in the output. If `ttl` is found, then it means the domain name is live and giving response.

Note: It is not mandatory for all live machines to give a ping reply.

os.system()

Let's consider that you want to run the command only, and don't care about the output; for example, you want to run a specific service, like httpd in the OS, then you don't need to use `os.popen()`. In this case, we use `os.system()` to run the DOS or shell command. See the following syntax:

```
os.system("command")
```

Let us discuss the ping command, which runs infinitely in the Windows system:

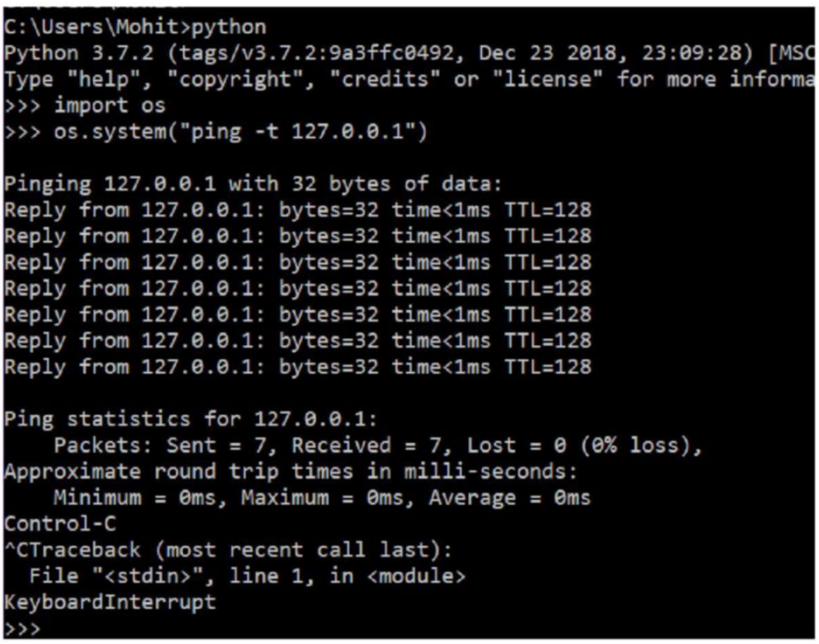

Figure 15.17

In Windows, the `ping -t ip_address` command is used to send the ping packet continuously.

If you want to copy or move a file from one folder to another folder, you can take the help of the `shutil` module. The `shutil.copy(src, dest)` and `shutil.move(src, dest)` help in copying and moving the `src` file, correspondingly, to the `dest` folder.

Exercise 1

In this section, we will perform exercises.

1. Rename the .jpg file present at the given folder.

   Change the extension .jpg to .png.

Let us see the situation, before making the program:

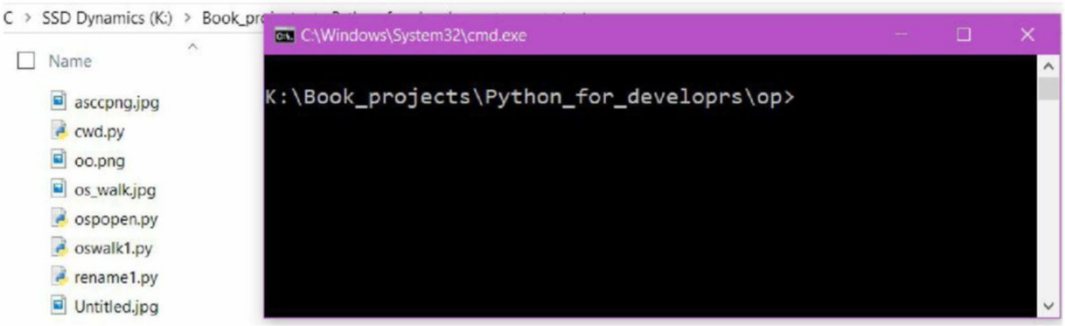

Figure 15.18

In the preceding screenshot, three files are `.jpg` files. You can change the extension manually, this will not take so much time. Let's consider that if there are more than 100 JPG files, then it will be time-consuming as well as a mundane task. Let us write the code that will perform the job within a fraction of second:

```
import os
print (os.getcwd())
os.chdir("K:\\Book_projects\\Python_for_developrs\\op\\test")
print (os.getcwd())
for file in os.listdir("."):
  if file.endswith(".jpg"):
    f_name = file.rsplit(".",1)[0]
    n_name = f_name+"."+"png"
    os.rename(file,n_name)
```

Let us run the code and check the output. See the following screenshot for the output:

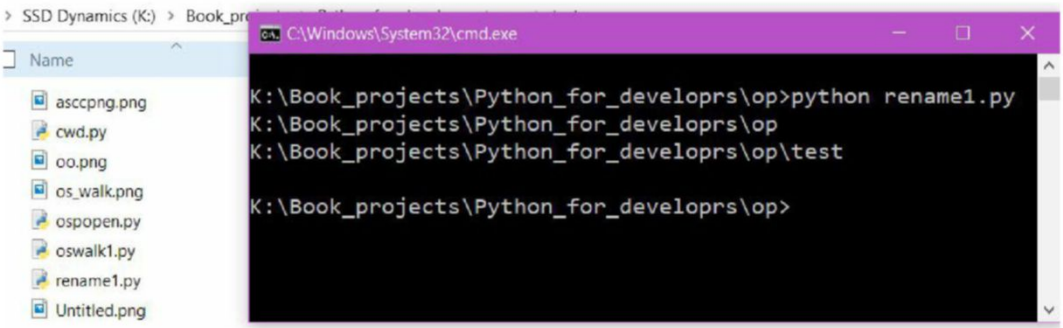

Figure 15.19

As you can see, the file's name successfully changed.

Exercise 2

Write a program to find out how many svchost.exe processes are being run currently.

Open the **`Task Manager`**, there are so many `svchost` services. See the following screenshot of **`Task Manager`**:

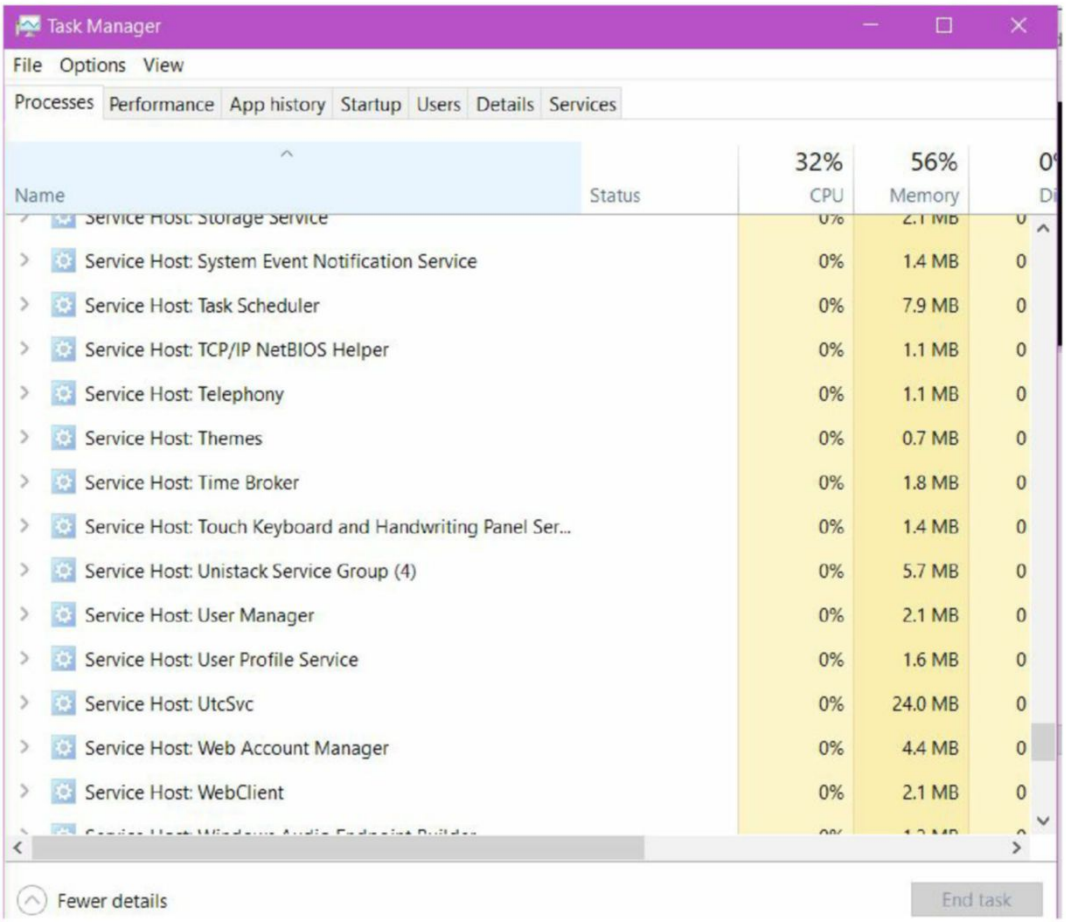

Figure 15.20

In the preceding figure, you can see a lot of processes are being run. The DOS command, tasklist, shows that all the services are running at that current moment. Let us use that command in our Python program:

```
import os
resp=os.popen("tasklist")
print (resp.read().count("svchost.exe"))
```

See the output in the following screenshot:

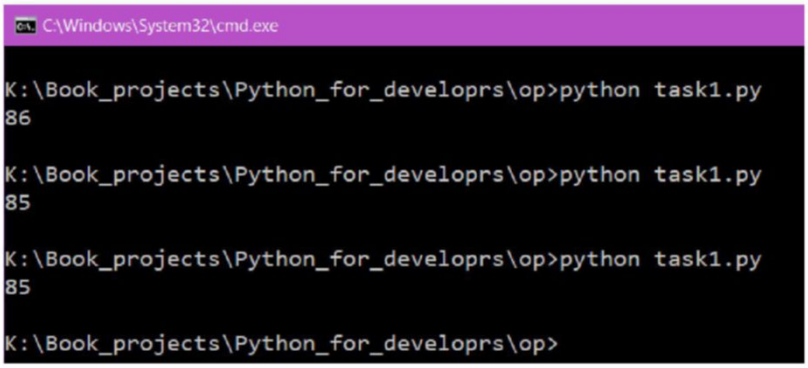

Figure 15.21

Conclusion

In this chapter, you have learned about dealing with your operating system. To do so, we used the os module, which allows us to interact with the OS. With the help of the OS module you can create directories and files. We can also rename and delete the files and folders. `os.access()` helps to check the permissions on the file, os.stat() returns the metadata about the file, like creation, modification time. The os.listdir() returns the list of files and folders present at the specified path, but it does not look recursively. To check recursively, we can use `os.walk()`. The `os.system()` and `os.popen()` facilitate a user to run any command on the operating system through Python. The `os.popen()` returns the output of the command and `os.system()` just runs the command without caring about the output. In the next chapter, you learn about the classes and objects.

Questions

1. Write a program to find the file in the given folder, which has been modified recently.
2. What is the return type of os.listdir() and what does that object contain?
3. Write a ping program that can run on the Windows as well as on Linux.

Chapter 16
Class and Objects

Python classes provide all the standard features of Object-Oriented Programming. Class is like a blueprint, where all the methods are defined. The instance of the class is called an object. An object is an actual thing that can use all the methods of the class. Let us consider an analogy of a human class, which has two hands, feet, and brain, etc. But you, I and all human beings of planet earth are the objects of human class. Almost all human beings possess the same basic functionality, like eating, sleeping, and working. These are the methods of the class. In this chapter, you will learn about the class, object, and method. We will also study the classes of built-in data structures, like string, tuple, and list.

Structure

- Class
- Object
- __init__ method
- Instance variable
- Class variable
- Inheritance
- Static method

- Class method
- Private variable and methods
- Decorator @property @setter @deleter
- Callable objects

Objective

In this chapter, we will study about the class and the object. We will learn about the instance variable, class variable, regular method, static method, and class method. We will also understand the importance of making a class by after learning inheritance and operator overloading. After that, you will learn about the private method, decorators like property, setter, and getter.

Class

A class is a blueprint or prototype that defines the variables and the methods (functions) common to all objects of a certain kind.

See the following diagram:

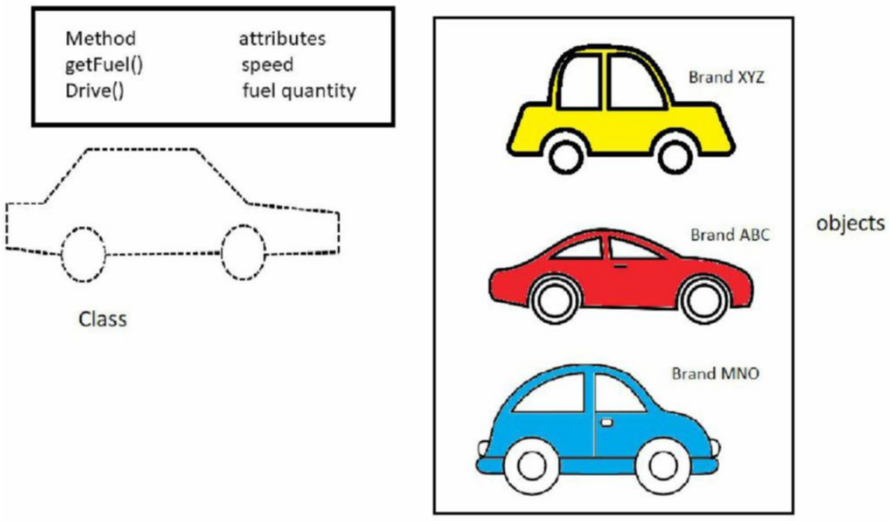

Figure 16.1

In the preceding diagram, class is represented by the dotted line. it possesses the methods—`getfuel()`, `Drive()`, and attributes - speed, and fuel quantity. All the cars, of different brands, are the objects and possess the same methods and attributes.

Object

An object is an instance of a class. Software objects are often used to model real-world objects you find in everyday life. In the *Figure 16.1*, the brands XYZ, ABC, and MNO are the objects.

Creating a class in python is very straightforward.

See the following code:

```
class <class name >(<parent class name>):
<method definition-1>
<method definition-n>
< Variables>
```

Let us write our first program of class. In this program, we are creating an empty class:

```
class Mohit_Org():
pass
```

In the preceding code, a class, `Mohit_org` has been created. The class is empty and the class body just filled with a `pass` statement. The class is a blueprint for creating instances. Let us create the instances:

```
class Mohit_Org():
pass
obj1 = Mohit_Org()
obj2 = Mohit_Org()
print (obj1)
print (obj2)
```

See the following screenshot of the output:

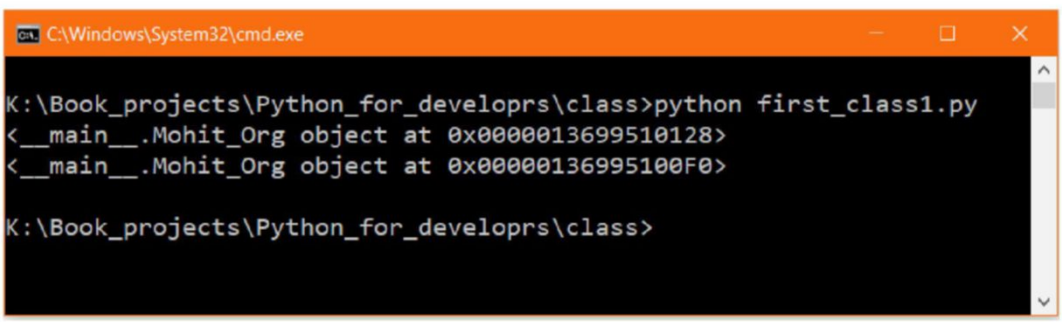

Figure 16.2

In the preceding screenshot, you can see two different instances.

Instance variable

Instance variable referred data which are unique to instances. The instance variable is often confusing for beginners. Through the series of the program, you will understand the significance of the instance variable.

Let us create an instance variable:

```
class Mohit_Org():
pass
obj1 = Mohit_Org()
obj2 = Mohit_Org()
obj1.First_name = "Mohit"
obj1.Last_name = "Raj"
obj1.Pay = "90000"
obj2.First_name = "Bhaskar"
obj2.Last_name = "Das"
obj2.Pay = "80000"
print ("Full Name ",obj1.First_name+" "+obj1.Last_name )
print ("Full Name ",obj2.First_name+" "+obj2.Last_name )
```

In the preceding code, `obj1.First_name`, `obj1.Last_name` and `obj1.Pay` are instance variables, which are unique to the instance `obj1`. Similarly, `obj2.First_name`, `obj2.Last_name` and `obj2.Pay` are instance variables of `obj2`.

See the following screenshot for the code's output:

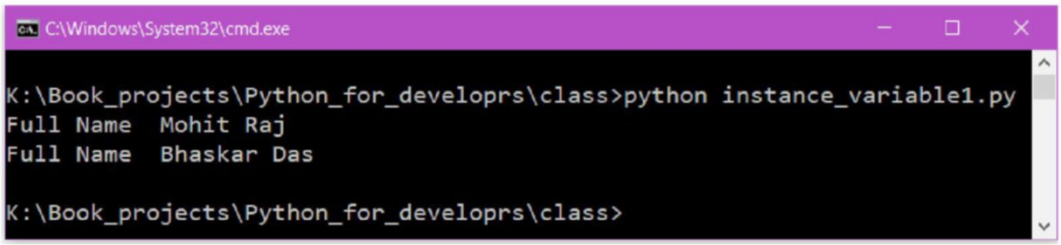

Figure 16.3

You can see the repeatable code for both instances. We don't need to set the variable all the time. To make it automatically, we are going to use the `__init__()`method.

The __init__ method or constructor

Here `the__init__()` method works as theconstructor of the class. When a user instantiates the class, it runs automatically. There is no need to call the `__init__` method. Let us see the code and understand with the help of it. Here we are going to write the full code, `const1.py`, and then we will follow it line-by-line:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
Self.Full_name = self.First_name+" "+self.Last_name


obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
print (obj1.First_name) # instance variable
print (obj2.First_name)
```

See the output in the following screenshot:

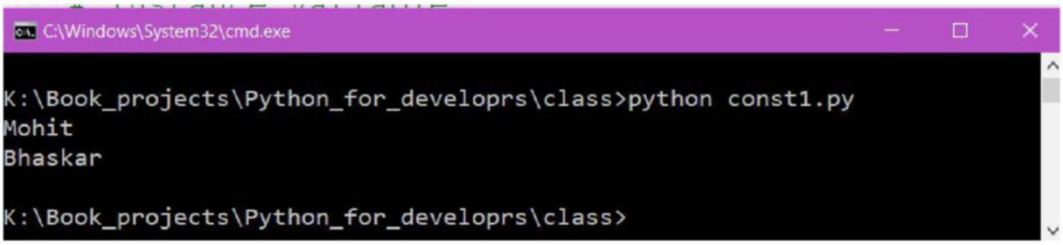

Figure 16.4

The code seems difficult to understand. Let's understandit line-by-line. The first line defines a class, as we already know. A method,`__init__(self, First, last, pay),` has been created inside the class then the first argument, self, of __init__() method, receives the instance of the class automatically. By convention, we call it self. You can use another name, but that is not a pythonic way, so it is advisable to stick to the convention. After declaring the variable self, we can specify other arguments that we want to accept. So, `__init__()` is going to receive three values -First, Last, and Pay. Inside the `__init__()` method, we are declaring an instance variable. So the `self.First_name`, `self.Last_name`, `self.Pay` and `self.Full_name` are instance variables.

The `self.First_name = First` statement is the same thing as the `obj1.First_name = "Mohit"` mentioned in the previous code, `instance_variable.py`. The `obj1` is the instance of the class, and the variable `self` is referred to an instance of the class, which is almost similar.

When we create an instance like `obj1 = Mohit_Org("Mohit", "Raj", 90000)`, the values `("Mohit", "Raj", 90000)` automatically get passed to the`__init__(self, First, last, pay)` method. We do not need to pass the value of the `self` variable, because the `obj1` instance has passed it automatically. It is similar for the `obj2` instance. In the same manner, we can say `self.First_name`, `self.Last_name`, `self.Pay` and `self.Full_name` are instance variables, which are unique to object `obj1` and `obj2`.

If you still have a doubt, persist for the `self` and instance variable. See the next section to know about the regular method.

Regular method

The function defined inside the class is called a **method**. Let us create a regular method, which generates an email address for users:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def email(self):
return self.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
print (obj1.email()) # instance variable
print (obj2.email())
```

The code is very similar to the previous code. A method, `email()`, has been added, which used the `self.Full_name` instance variable. By using the syntax `obj1.make_email()` instance, `obj1` calls the method `email()`.

The `email()` is the regular method. The regular method in the class automatically takes an instance as the first argument. That is why, by convention, we use self as the first argument that expects instance.

If you remember in the Python list chapter, we did the same thing using `list1.append()`. If you relate list with the class mentioned above, thenlist1 is the instance and `append()` is the regular method of the class list. You can also define the list as showcased below:

```
List1 = list()
```

See the output in the following screenshot:

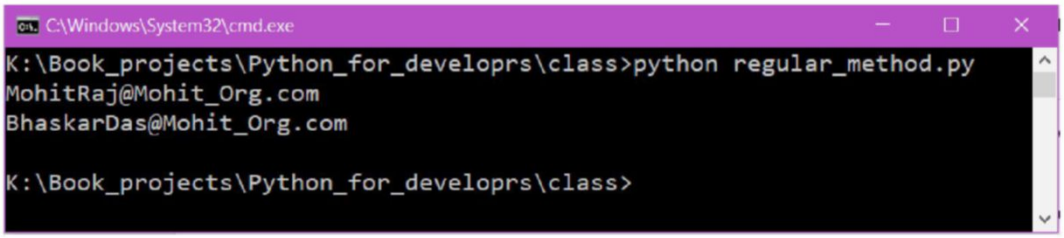

Figure 16.5

Let us explore theself variable in depth. If you still have a doubt, the next example will clear it:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def email():
return self.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
print (obj1.email()) # instance variable
```

In the preceding code, the `self` variable from email has been deleted. For experiment purpose, only one instance is used.

See the following screenshot for the output:

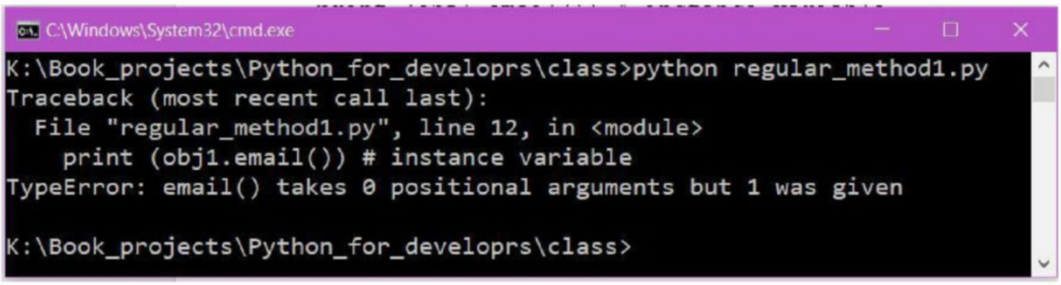

Figure 16.6

You can see the error, **`email() takes 0 positional arguments, but 1 was given`**. This may be confusing, as no argument has been passed in the **`obj1.email()`** syntax. So what is the `email()` method expecting? In this case, the `obj1` instance is getting passed automatically. That is why, we use the `self` argument to the methods of the class.

For better understanding, see the following code:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def email(self):
return self.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
print (obj1.email()) # instance variable
print (Mohit_Org.email(obj1))
```

See the result in the following screenshot:

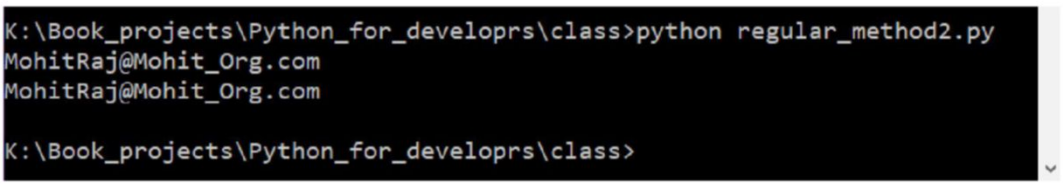

Figure 16.7

In the preceding code, the `self` variable has been put in the `email (self)` method. In the last line, `Mohit_Org.email(obj1)` signifies what is happening in the background.The syntax, `obj1.email()` and `Mohit_Org.email(obj1)`, are the same thing.

The syntax `Mohit_Org.email(obj1)` stats that `class.method(instance)`. In this syntax, we are passing the instance to the `email()` method and the `self` argument is accepting that instance. So `obj1.email()` is transformed into `Mohit_Org.email(obj1)`.

For more clarity, let us do one more amendment:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name
print ("In constructor", id(self))

def email(self_new):
print ("In email",id(self_new))
return self_new.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
print (obj1.email()) # instance variable
print ("Outside class", id(obj1))
```

In the preceding program, we have changed the `self` variable to `self_new` and printed the memory address of the instance at different places:

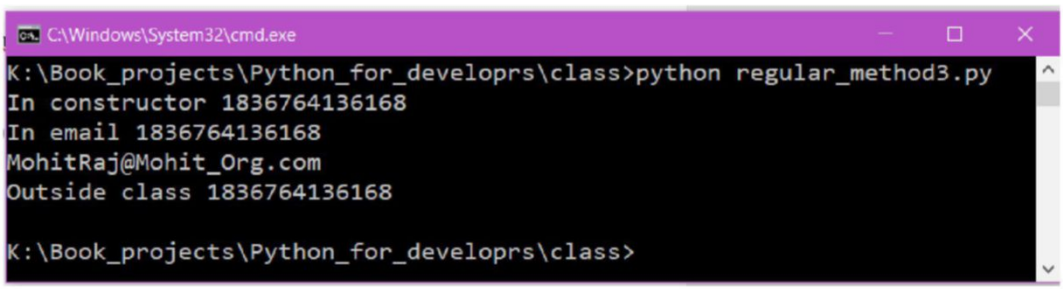

Figure 16.8

Class variable

Class variables are shareable among all the objects of the class. The `class` variable must be the same for all the instances. To understand with an example, let's assume that the `Mohit_Org` gives 30 percent increment based upon `Pay`:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*1.3
return self.Pay

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)

print (obj1.increment_pay())
print (obj2.increment_pay())
```

Let us see the output of the `class_variable1.py` in the following screenshot:

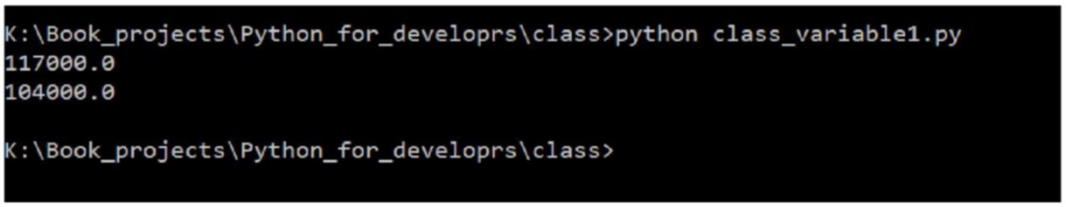

Figure 16.9

In the preceding code, we have added one method, `increment_pay()`. In the method, we have hardcoded the value, and it is not a good practice to hardcode any variable. So, we can make a `class` variable.

See the following program `class_variable2.py`:

```
class Mohit_Org():
inc_factor = 1.3
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*Mohit_Org.inc_factor
return self.Pay

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)

print (obj1.increment_pay())
print (obj2.increment_pay())
```

See the following screenshot for the result:

```
K:\Book_projects\Python_for_developrs\class>python class_variable2.py
125999.99999999999
112000.0

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.10

The program is very similar; a new class variable, `inc_factor = 1.3`, is declared and used. A `class` variable can only be used by a class or an object. In the preceding program, we have to use the class name. Let us use the `class` variable with the object. See the following `class_variable3.py` code:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
```

```
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)

print (obj1.increment_pay())
print (obj2.increment_pay())
```

See the output in the following screenshot:

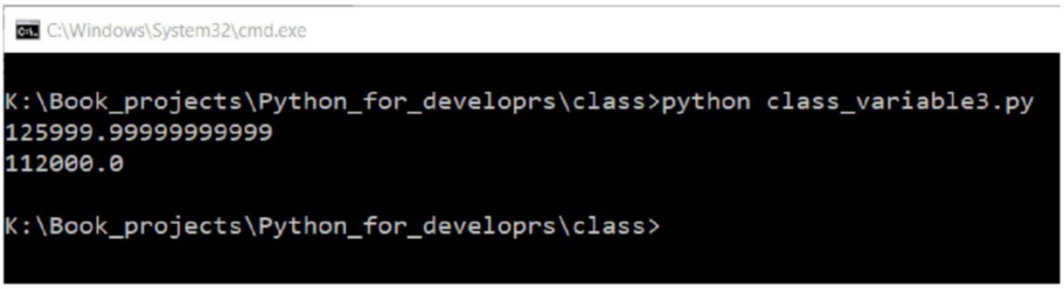

Figure 16.11

So, both the outputs are the same, so what is the difference between accessing the class variable by using object and class? See the following code to understand the same:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name
```

```
def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)

obj1.inc_factor = 1.5
print (obj1.increment_pay())
print (obj2.increment_pay())
print (obj1.__dict__)
print ("************************************************")
print (obj2.__dict__)
print ("************************************************")
print (Mohit_Org.__dict__)
```

See the output in the following screenshot:

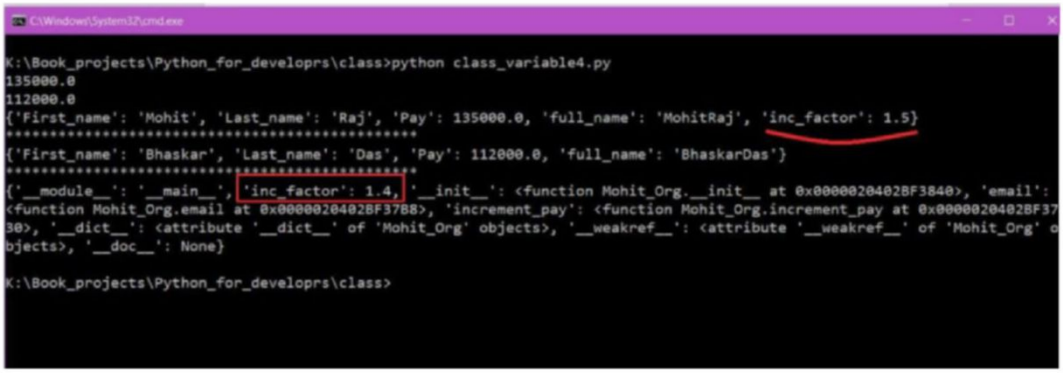

Figure 16.12

In the preceding code, you can see that we are accessing the `class` variable using the instance.

The interpreter first checks the instance's dictionary. If it contains the attribute, then it uses the attribute from instance's dictionary. If the instance does not contain the attribute, then it checks whether the class or its parent class contain that attribute.

The obj1instance first checks if we have theinc_factor in our namespace. Since it has, the `obj1` uses inc_factor 1.5. The `obj2` instance checks the same thing, but obj2 does not have theinc_factor in its namespace.That is why, obj2 takes from the class namespace. The `obj1.__dict__`, `obj2.__dict__` and `Mohit_Org.__dict__` shows all the attributes of `obj1`, `obj2`, and `Mohit_Org`, respectively. of the importance of using `self.inc_factor` is that it gives more ability to change the value for a single instance.

Class inheritance

In this section, we are going to learn about inheritance. Inheritance allows us to inherit the method and attribute of the parent class. By using inheritance, the new child class automatically gets all the methods and attributes of the existing parent class. The parent class is also regarded as a base class and general class. Similarly, the child class is also called a derived class and specific class. See the following syntax:

```
class DerivedClassName(BaseClassName):
<statement-1>
    .
    .    .
<statement-N>
```

See the example in the following diagram:

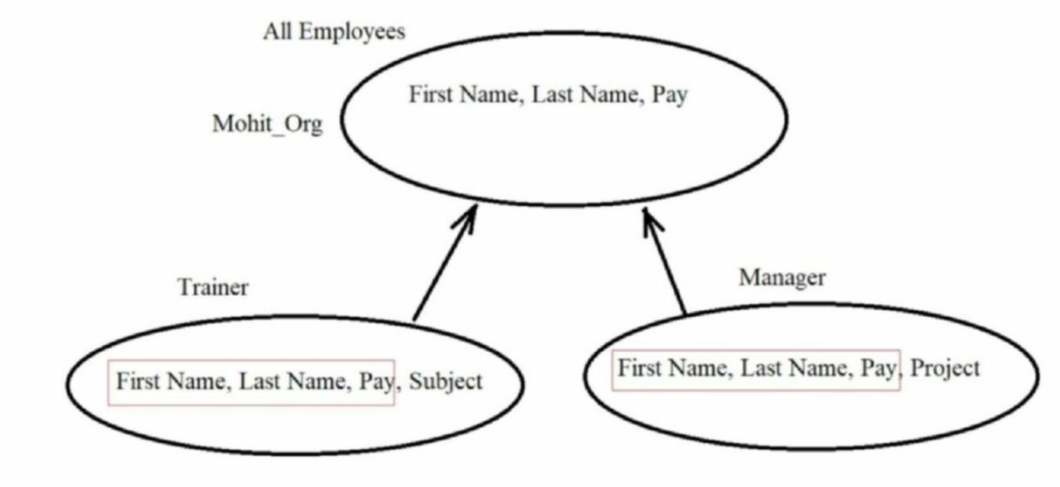

Figure 16.13

In the preceding diagram, the `Mohit_Org` is a class, which contains the necessary information of all the employees. The organization comprises of different types of

employees, like `Trainer` and `Manager`. The `Trainer` class is a specific class, which includes general attributes and particular attributes. In this scenario, the `Trainer` class inherits the necessary details of the employees from the base class (`Mohit_org`). With the help of inheritance, we can reuse the code.

See the following example for `class_inheritance1.py`:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

class Trainer(Mohit_Org):
pass

obj1 = Trainer("Mohit", "Raj", 90000)
obj2 = Trainer("Bhaskar", "Das", 80000)

print (obj1.email())
print (obj2.email())

print (obj1.increment_pay())
print (obj2.increment_pay())
```

See the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\class>python class_inheritance1.py
MohitRaj@Mohit_Org.com
BhaskarDas@Mohit_Org.com
125999.99999999999
112000.0

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.14

In the preceding code, a new class,`Trainer`, which inherits the base class, has been defined. The preceding example shows that the child class instance can call the attributes and the method of the base class.In the above example, instances of `Trainer` class call the `email()` and `increment_pay`()methods. When we instantiate the `Trainer` class, it first looks at the `__init__` method of the `Trainer` class. As the `Trainer` class is empty, the interpreter checks the chain of inheritance.

If you want to check the chain of inheritance, you use the `help()` function:

```
print (Help(Trainer))
```

Check the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\class>python class_inheritance1.py
MohitRaj@Mohit_Org.com
BhaskarDas@Mohit_Org.com
125999.99999999999
112000.0
Help on class Trainer in module __main__:

class Trainer(Mohit_Org)
 |  Trainer(First, last, pay)
 |
 |  Method resolution order:
 |      Trainer
 |      Mohit_Org
 |      builtins.object
 |
 |  Methods inherited from Mohit_Org:
 |
 |  __init__(self, First, last, pay)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  email(self)
 |
 |  increment_pay(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Mohit_Org:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Mohit_Org:
 |
 |  inc_factor = 1.4

None
```

Figure 16.15

In the preceding snapshot, `Method resolution` means that if an instance called the method, the interpreter first checks the `Trainer` class and then `Mohit_Org`. Let us use a complicated example for `class_inheritance2.py`:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
```

```
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

class Trainer(Mohit_Org):
def __init__(self,First, last, pay, subject):
Mohit_Org.__init__(self,First, last, pay)
self.subject = subject

def Subject(self):
return self.full_name+" is teaching "+self.subject

obj1 = Trainer("Mohit", "Raj", 90000,"Python")
obj2 = Trainer("Bhaskar", "Das", 80000, "JAVA")

print (obj1.email())
print (obj2.email())
print (obj1.Subject())
print (obj2.Subject())
```

See the output in the following screenshot:

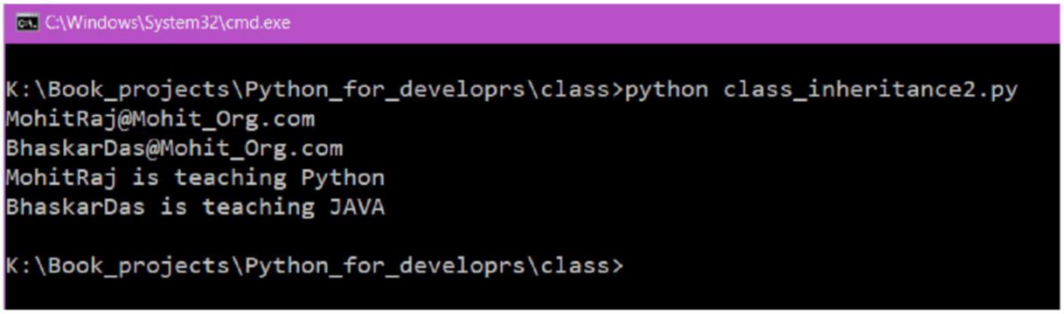

Figure 16.16

A couple of things are new here. The `Trainer` class contains its constructor, and a new method called `Subject()`. In the previous program, when we made the object, the object automatically invoked the class's constructor. But the `Trainer` class does not contain the constructor, so the object checked the parent class and invoked the parent class's constructor. See the following snapshot:

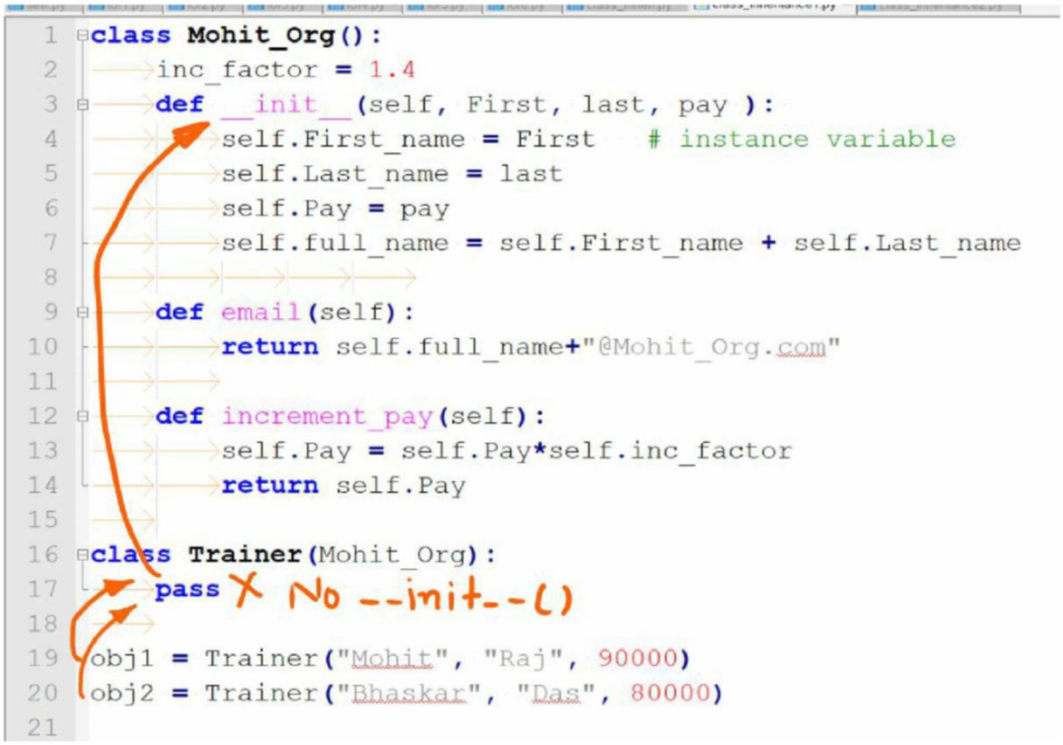

Figure 16.17

But in the `class_inheritance2.py` program, we declared the constructor of the `Trainer` class. So, now when we create the object of the `Trainer` class, only the constructor of the `Trainer` class will be invoked. The constructor of the base class, `Mohit_Org`, will not be called. Consequently, the instance variables `self.First_name`, `self.Last_name` and `self.Pay` would not get initialized. In order to invoke the base class constructor, we used one statement `Mohit_Org.__init__`

`(self,First, last, pay).` The statement invoked the constructor base class, `Mohit_Org`, explicitly. See the following figure:

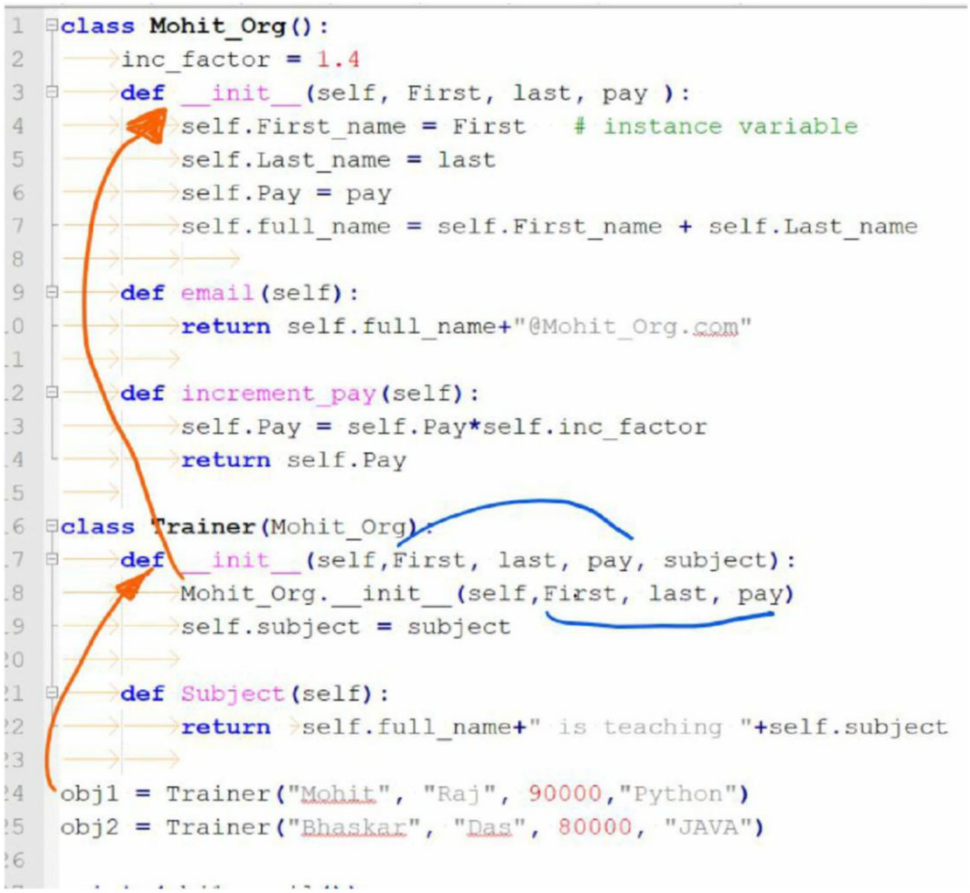

Figure 16.18

In the preceding figure, as you can see, the `Trainer` class constructor takes five arguments - `self`, `First`, `last`, `pay` and `subject`.The `self`, `First`, `last` and `pay` arguments are handled by the base class constructor and the `subject` argument is handled by the `Trainer` class.

We can also use the `super()` method to inherit the base class.

See the following code:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
```

```
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

class Trainer(Mohit_Org):
def __init__(self,First, last, pay, subject):
super().__init__(First, last, pay)
self.subject = subject

def Subject(self):
return self.full_name+" is teaching "+self.subject

obj1 = Trainer("Mohit", "Raj", 90000,"Python")
obj2 = Trainer("Bhaskar", "Das", 80000, "JAVA")

print (obj1.email())
print (obj2.email())
print (obj1.Subject())
print (obj2.Subject())
```

See the results in the following output:

```
K:\Book_projects\Python_for_developrs\class>python class_inheritance3.py
MohitRaj@Mohit_Org.com
BhaskarDas@Mohit_Org.com
MohitRaj is teaching Python
BhaskarDas is teaching JAVA

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.19

What is the benefit of a super method? If you change the base class name then, you do not need to change the class name for invoking the constructor.

Multilevel inheritance

Python supports the multilevel inheritance. Multilevel means that the child class **A** inherits the parent class **B**, and class **B** inherits class **C**:

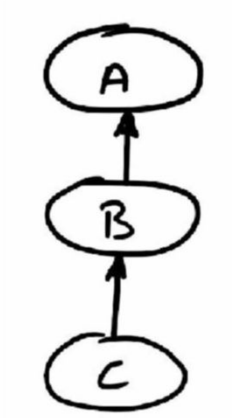

Figure 16.20

Let us look at an example:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

class Manager(Mohit_Org):
def __init__(self,First, last, pay, dep):
super().__init__(First, last, pay)
self.dep = dep
```

```
def Department(self):
return self.full_name+" is manager in "+self.dep

class Assitant_Manager(Manager):
def __init__(self,First, last, pay, dep, sub_dep):
super().__init__(First, last, pay,dep)
self.sub_dep = sub_dep

def Sub_Department(self):
return self.full_name+" is manager in "+self.sub_dep

obj1 = Assitant_Manager("Mohit", "Raj", 90000,"Computer", "Cloud")

print (obj1.email())
print (obj1.Department())
print (obj1.Sub_Department())
print (Assitant_Manager.__mro__)
```

See the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\class>python class_inheritance4.py
MohitRaj@Mohit_Org.com
MohitRaj is manager in Computer
MohitRaj is manager in Cloud
(<class '__main__.Assitant_Manager'>, <class '__main__.Manager'>, <class '__main__.Mohit_Org'>, <class 'object'>)

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.21

In the preceding example, a total of three class are there.The `Assitant_Manager` class inherits the `Manager` class and the `Manager` class inherits the `Mohit_Org` class. See the following diagram you will get the idea of passing arguments:

Figure 16.22

In the preceding example, the new argument, `sub_dep`, is handled by the `Assitant_Manager` class, while the rest of the argument will be handled by the base class (`Manager` class) and the grandparent class (`Mohit_Org`). The syntax, `Assitant_Manager.__mro__`, will display the hierarchy level of the inheritance.

Multiple inheritance

Python supports multiple inheritances too. Multiple inheritance means that one class inherits the attribute of two independent classes. See the following diagram:

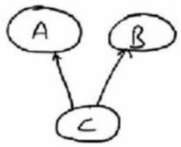

Figure 16.23

Let us see the code for `class_inheri_multiple.py`:

```
class A():
def sum1(self,a,b):
c = a+b
return c

class B():
def mul(self,a,b):
c = a*b
return c

class C(B,A):
pass

obj1 = C()
print (obj1.sum1(10,20))
```

The preceding example is quite simple to understand. If both the base classes contain the same method, then the object of the child class would invoke the method of class which comes first in the syntax.

See the following example:

```
class A():
def sum1(self,a,b):
c = a+b
return c
class B():
def sum1(self,a,b):
c = a*b
return c
class C(B,A):
pass


obj1 = C()
print (obj1.sum1(10,20))
```

See the following output:

```
K:\Book_projects\Python_for_developrs\class>python class_inheri_multiple2.py
200

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.24

In the syntax class `C(B, A)`, class B has more priority. Therefore, the object of class `C` would use the method of class `B`.

Let's look at more complicated examples:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = "Mohit123"   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

class XYZ_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@XYZ_Org.com"

class EMP(XYZ_Org,Mohit_Org):
def __init__(self, first,last, pay, type):
super().__init__(first,last,pay)
self.type = type
```

```
obj1 = EMP("Mohit", "Raj", 90000,"Trainer")
print (obj1.email())
```

See the following screenshot for the output:

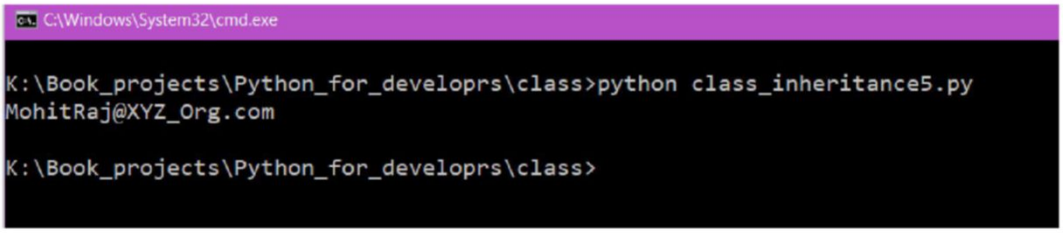

Figure 16.25

In the preceding example, class `Mohit_Org` and `XYZ` offered the same methods and attributes, but the `super()` function would call the base class `XYZ`, because in the syntax class, `EMP(XYZ_Org, Mohit_Org):,XYZ` comes first.

Let us see one more example:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = "Mohit123"   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

class XYZ_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@XYZ_Org.com"
```

```
class EMP(XYZ_Org,Mohit_Org):
def __init__(self, first,last, pay, type):
Mohit_Org.__init__(self,first, last,pay)
self.type = type

obj1 = EMP("Mohit", "Raj", 90000,"Trainer")
print (obj1.email())
```

In the preceding example, theorder of the inheritance is `XYZ` first and then `Mohit_Org`. But the statement.`Mohit_Org.__init__(self,first, last,pay)`, invokes the constructor of `Mohit_Org`. So, in this situation, the instance variables of the `Mohit_Org` class will be initialized, but the email method of class `XYZ` will be called.

Check the following screenshot for the output:

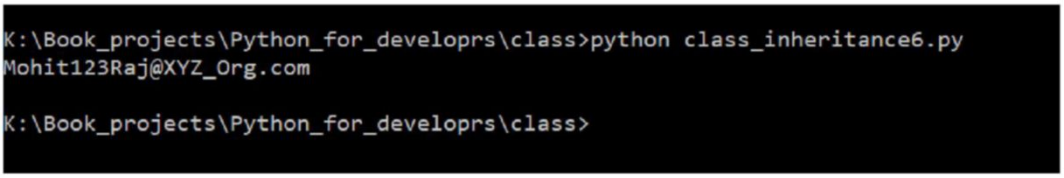

Figure 16.26

Operator overloading

In this section, we will learn about the operator overloading with special methods. Generally, people call them magic methods. We will use these methods in operator overloading. First, let us understand what operator overloading is. By using a special method, we will be able to change the built-in behavior of the operator. The special method is surrounded by a double underscore (__). Some people called it the `dunder` method.

Let us take the example of the+ operator.

Take a look at the following example:

```
>>> 1+8
9
>>> "IBM" + "REDHAT"
'IBMREDHAT'
>>>
```

You can see the different behavior of the + operator. The integer number is added, and the strings are concatenated. It depends upon the object that you are using with the+ operator. The + calls a special method in the background:

```
>>> (1).__add__(8)
9
>>> int.__add__(1,8)
9
>>> ("IBM").__add__("RedHat")
'IBMRedHat'
>>>
>>> str.__add__("IBM","RedHat")
'IBMRedHat'
>>>
```

If you use the `dir("string")` function, then it will show the magic method `__add__`:

Figure 16.27

Actually, when you add two strings, the + operator calls `__add__` method defined the `str` class.

Let us try to add two objects of the `Mohit_Org` class. Look at the following example:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name
```

```
def email(self_new):
return self_new.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
print (obj1+ obj2)
```

See the result of code in the following:

```
K:\Book_projects\Python_for_developrs\class>python opeartor_over1.py
Traceback (most recent call last):
  File "opeartor_over1.py", line 13, in <module>
    print (obj1+ obj2)
TypeError: unsupported operand type(s) for +: 'Mohit_Org' and 'Mohit_Org'

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.28

The output showcased above, is showing an error - `TypeError: unsupported operand type(s) for +: 'Mohit_Org' and 'Mohit_Org`. So, the question here is, what are we trying to achieve after addition?

As we have not defined the `__add__` method. Check the `dir()` on the object.

Just add the following line:

```
print (dir(obj1))
```

Check the output in following screenshot:

```
K:\Book_projects\Python_for_developrs\class>python opeartor_over1.py
['First_name', 'Full_name', 'Last_name', 'Pay', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '
__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '_
subclasshook__', '__weakref__', 'email']

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.29

You can see that there is no `__add__` method specified. Consider the requirement, when we add two objects, and then add their pay:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
```

```
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def email(self_new):
return self_new.Full_name+"@Mohit_Org.com"

def __add__(self,other):
return self.Pay + other.Pay

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
print (obj1+obj2)
```

See the result of code in the following screenshot:

```
K:\Book_projects\Python_for_developrs\class>python opeartor_over1.py
170000

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.30

There is no error, because we have defined the `__add__` method. The syntax, `obj1+obj2`, calls the `__add__` method, like `Mohit_Org.__add__(obj1, obj2)`.

The method `Mohit_Org.__add__(obj1, obj2)` is passing two objects. Therefore in `the__add__(self, other)` method, two arguments have been specified to take two objects.

Similarly, if we try to compare the two objects, then we need to define our expected function of the comparison. Let us consider that we want to compare the Pay of the employees. Look at the following example:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def email(self_new):
return self_new.Full_name+"@Mohit_Org.com"
```

```
def __add__(self,other):
return self.Pay + other.Pay

def __gt__(self, other):
return self.Pay> other.Pay

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
print (obj1+obj2)
print (obj1> obj2)
```

See the following screenshot for the output of the preceding code:

```
K:\Book_projects\Python_for_developrs\class>python opeartor_over2.py
170000
True

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.31

In the syntax `obj1> obj2`, the > calls the `__gt__` method. In the `__gt__` method, we are comparing the Pay of the employees. If you remember, we have used the `len()` function. Actually, we use the `len(object)` and then the `len()` function to call the class.`__len__(object)`. Let us try to print the length of the `obj1`:

You can import the `opeartor_over2` program as a module:

```
>>> from opeartor_over2 import Mohit_Org, obj1
>>> len(obj1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'Mohit_Org' has no len()
>>>
```

The precedingerror indicates that there is no functionalitydefined for the `len()` function. Let's add the magic `__len__` method. In the `__len__` method, we find the length of the object's full name.

Look at the new code:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
```

```
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def email(self_new):
return self_new.Full_name+"@Mohit_Org.com"

def __add__(self,other):
return self.Pay + other.Pay

def __gt__(self, other):
return self.Pay> other.Pay

def __len__(self):
return len(self.Full_name)

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
print (obj1+obj2)
print (obj1> obj2)
print (len(obj1))
```

See the following screenshotfor the output:

```
K:\Book_projects\Python_for_developrs\class>python opeartor_over2.py
170000
True
8

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.32

Let us consider that we want to print the object. See the following code:

```
>>> from opeartor_over2 import Mohit_Org, obj1
170000
True
8
>>> print (obj1)
```

```
<opeartor_over2.Mohit_Org object at 0x00000284F2C30F98>
>>>
```

If you want to print your customized message, then add the following method in the preceding code:

```
def __str__(self):
return "Object related to Mohit_Org"
```

Rerun the code again, as showcased below:

```
>>> from opeartor_over2 import Mohit_Org, obj1
170000
True
8
>>> print (obj1)
Object related to Mohit_Org
>>>
```

See the following table for special methods.

Operator overloading special functions in Python.

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Table 16.1

Class method

You have seen the regular methods of a class. The regular method automatically takes an instance as the first argument, and by convention, we called it `self`. How can we pass the `class` as an argument, so that we can change the `class` variable in the method? To change the `class` variable, we use the `class` method. The `class` method takes the `class` as the first argument. To turn the regular method into the class method, we would use a decorator `(@classmethod)` at the top of the method. Let us see an example for `class_methods.py:`

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

@classmethod
def inc_change(cls, amt):
cls.inc_factor = amt

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 80000)
obj1.inc_change(1.5)

print (obj1.increment_pay())
print (obj2.increment_pay())
```

In the preceding example, a new method, `inc_change()`, has been added. The method has been converted to the class method by using the `decorator - @classmethod`. The class method takes the class as the first argument, so by convention, we use cls to represent a class. The `cls.inc_factor` is the class variable here. The line, `obj1.`

`inc_change(1.5)`, calls the `class` method. You can also use the syntax, `Mohit_Org.inc_change(1.5,)` to call the method.

See the following screenshot for the output:

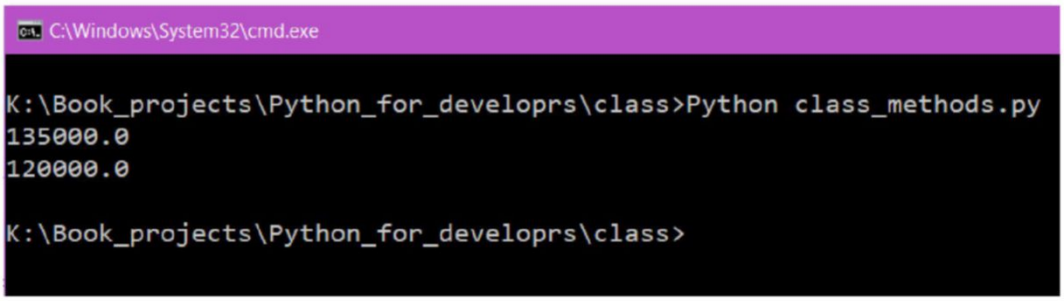

Figure 16.33

The class method is regarded as an alternative constructor. While making the object, we passed three values - `Mohit`, `Raj` and `90000` as showcased below:

```
obj1 = Mohit_Org("Mohit", "Raj", 90000)
```

Let us consider that we get a string,"`Mohit Raj 90000`".In this case, you will use the string's split method to split the string, and then create the object as showcased below:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

str1 = "Mohit Raj 90000"
a,b,c = str1.split()
```

```
obj1 = Mohit_Org(a,b,int(c))
print (obj1.increment_pay())
```

Check the output of code in the following screenshot:

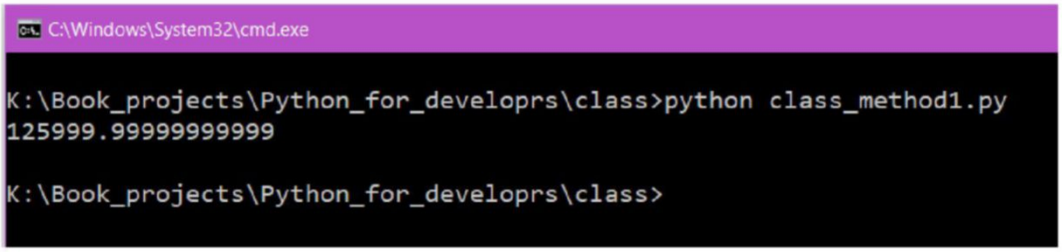

Figure 16.34

The preceding code is technically correct; however, it does not look mature.

See the following code:

```
class Mohit_Org():
inc_factor = 1.4
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
self.Pay = self.Pay*self.inc_factor
return self.Pay

@classmethod
def alt_const(cls,str1):
name, last, pay = str1.split()
return cls(name,last,int(pay))

name = "Mohit Raj 90000"
```

```
a1 = Mohit_Org.alt_const(name) # Mohit_Org("Mohit",'Raj',pay)
print (a1.email())
```

In the preceding code, the `class` method, `alt_const()`, acts as an alternative constructor. The line, `a1 = Mohit_Org.alt_const(name)`, calls the `class` method `atl_const()`, which returns the `cls(name,last,int(pay))`.

The return value is equivalent to `Mohit_Org("Mohit",'Raj',pay)` and `a1` becomes the object.

Check the output of code in the following screenshot:

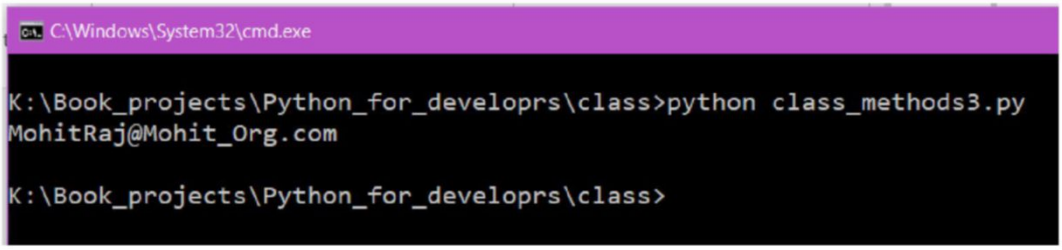

Figure 16.35

Static method

The static method does not take an instance or a class as the first argument. The static methods are just simple function. But we include the static method in the class because it has some logical connection with the class.

Consider a situation in, when the pay of a person is less than 50000 then increment would be pay is 1.40, otherwise 1.30. To turn a regular method into a class method, we would use a decorator `(@staticmethod)` at the top of the method.

Let us look at the program:

```
class Mohit_Org():
inc_factor = 1.4
inc_factor1 = 1.5
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.full_name = self.First_name + self.Last_name

def email(self):
```

```
return self.full_name+"@Mohit_Org.com"

def increment_pay(self):
if self.decide(self.Pay):
self.Pay = self.Pay*self.inc_factor1
else :
self.Pay = self.Pay*self.inc_factor
return self.Pay

@staticmethod
def decide(pay):
if pay <= 50000:
return True
else :
return False

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj2 = Mohit_Org("Bhaskar", "Das", 50000)

print (obj1.increment_pay())
print (obj2.increment_pay())
```

See the following screenshot for the output:

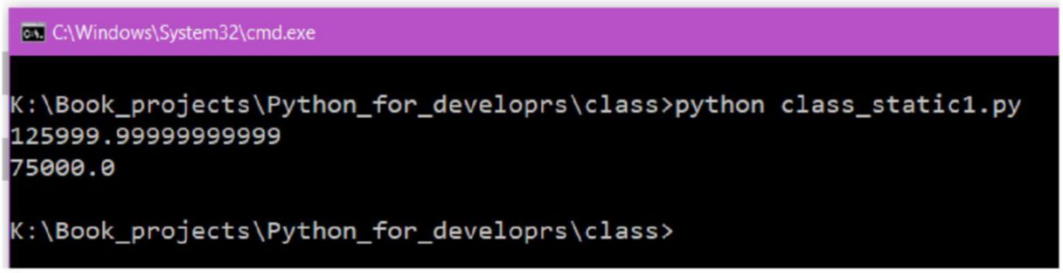

Figure 16.36

Private method and private variable

Python does not have real private methods, so two underlines in the beginning, make a variable and a method private.

Let us see a simple example:

```
class Mohit_Org:
__amount = 60  # private variable
def __info(self):  # private method
print ("I am in Class A")

def hello(self):
print ("Amount is ",Mohit_Org.__amount)
self.__info()

a = Mohit_Org()
a.hello()
a.__info()
```

You can see the benefit of a private variable. Outside the class, you cannot access a private method as well as a private variable, but inside the class, you can access the private variables. In the `hello()` method, the `__amount` can be obtained as showcased in the output (`Amount is 60`):

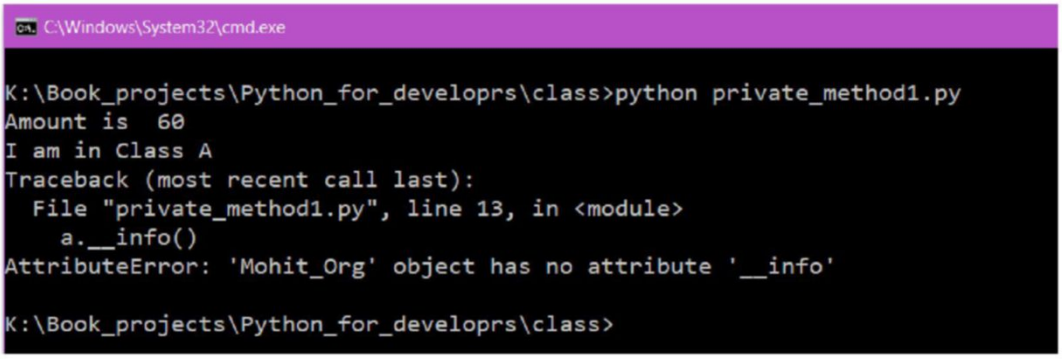

Figure 16.37

However, you can access private variables and private method from outside the class by using the following syntax:

```
instance._class-name__private-attribute
```

Add the following line:

```
a._Mohit_Org__info()
print (a._Mohit_Org__amount)
```

Check the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\class>python private_method1.py
Amount is  60
I am in Class A
I am in Class A
60

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.38

Decorator @property @setter and @deleter

In this section, we will see how to use the `@property` decorator. The `@property` decorator turns a method into a "getter" for a read-only attribute, with the same name.

Simply put, if you want to treat the method as an instance variable, then use the `@property` decorator. Let us see an example:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

@property
def email(self):
return self.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
print (obj1.email)
```

In the preceding program, we used `obj1.email`; however, `email()` is a method. With the help of the property decorator, the method behaves like an instance variable, as showcased in the following screenshot:

```
K:\Book_projects\Python_for_developrs\class>python property1.py
MohitRaj@Mohit_Org.com
```

Figure 16.39

Let's try to assign a value to `obj1.email`, as showcased as follows:

```
obj1.email = Mohitadmin@Mohit_Org.com
```

We are getting an error, as can be seen in the following screenshot:

```
K:\Book_projects\Python_for_developrs\class>python property1.py
Traceback (most recent call last):
  File "property1.py", line 13, in <module>
    obj1.email = "Mohitadmin@Mohit_Org.com"
AttributeError: can't set attribute

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.40

We do not want to change the existing code. In order to assign value, we will use the `@setter` method.

Take a look at the full code:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

@property
def email(self):
return  self.Full_name+"@Mohit_Org.com"

@email.setter
def email(self, value):
self.Full_name = value

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj1.email = "Mohitadmin"
print (obj1.email)
```

When we set `obj1.email = "Mohitadmin"`, it calls `email(self, value)` due to the `@email.setter` decorator.

See the following screenshotfor the output:

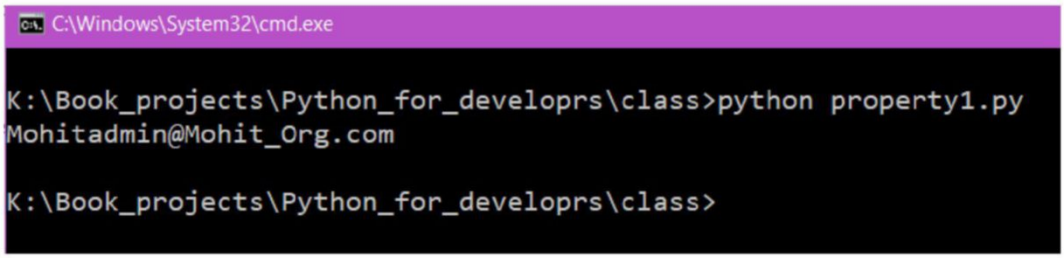

Figure 16.41

If you want to delete the `Full_name`, you can use the `@deleter` decorator.

Take a look at the full code:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

@property
def email(self):
return  self.Full_name+"@Mohit_Org.com"

@email.setter
def email(self, value):
self.Full_name = value

@email.deleter
def email(self):
print ("Delete the name")
self.Full_name= None

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj1.email = "Mohitadmin"
```

```
print (obj1.email)
del obj1.email
```

In the preceding code, the `del obj1.email` syntax calls the `email(self)` method of the`@email.deleter`.

This is how `getter, setter` and `deleter` work in Python.

Callable objects

If you want to check whether any object is callable or not, pass it to a built-in function, `callable`. The `callable` function returns `True` or `False`:

```
>>> callable(tuple)
True
>>> callable(tuple.count)
True
>>> callable(list)
True
>>> callable(str)
True
>>> callable("hello")
False
```

The question is that whether the object of `Mohit_Org` is callable or not. Look at the following example:

```
>>> from property1 import Mohit_Org
Mohitadmin@Mohit_Org.com
```

Delete the name:

```
>>> obj12 = Mohit_Org("m", "r", 90000)
>>> callable(obj12)
False
>>> obj12("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Mohit_Org' object is not callable
>>>
```

In order to make an object callable, we need to add the `__call__` method.

Look at the following example:

```
class Mohit_Org():
def __init__(self, First, last, pay ):
self.First_name = First   # instance variable
self.Last_name = last
self.Pay = pay
self.Full_name = self.First_name+self.Last_name

def __call__(self,first):
self.Full_name = self.First_name+"admin"

def email(self):
return  self.Full_name+"@Mohit_Org.com"

obj1 = Mohit_Org("Mohit", "Raj", 90000)
obj1("Mohit")
print (obj1.email())
```

In the preceding code, we added the `__call__` method, which adds `admin` after the first name. The `obj1("Mohit")` syntax calls the `__call__` method.

See the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\class>python call1.py
Mohitadmin@Mohit_Org.com

K:\Book_projects\Python_for_developrs\class>
```

Figure 16.42

Conclusion

In this chapter, we have learned about the concept of class and object. An object is a real entity, and a class defines the properties of the object. The object is the instance of a class. We learned about the instance variable and class variable. The instance variable is unique to the instance, and the class variable is shareable among the objects. The function defined inside the class is called methods. To reuse the code, we can use inheritance. Python supports multilevel and multiple inheritances. With the help of magic methods, we can use operator overloading. The instance variable

can be accessed from outside the class; if you want to give access to outside the class, you can make private variables. In order to change the behavior of class, we use decorators. To make an object callable, we can use the `__call__` method. In the next chapter, you will learn about the use of multithreading.

Questions

1. What is the 'self' in the class method?
2. What is the difference between instance variable and class variable?
3. Can we access the class variables with objects?
4. What is the importance of private variables?
5. What is the need to initialize the base class constructor in the child class?

Chapter 17
Threads

So far, we have seen the single thread-based applications. Consider you want to make an application that takes input from the user or is always ready to take input from the user or any client. After taking the data, the program processes the data and gives the result. Now consider a single process or thread-based program where you provide the input in the `while` loop and the interpreter process the data and returns the result. If the interpreter spends more time to process the data, then, at that time, it would not be ready to take the input from the user. In a multithreading environment, we dedicate one thread to take input and another thread to process the data. In this way, taking inputs and processing the data can work parallelly. In this chapter, you will learn how to develop multithreaded applications.

Structure

- Thread creation using class
- Thread creation using function
- Important threading methods
- The join method
- Daemon thread
- Locks
- GIL

Objective

In this chapter, you will start by creating the thread, and then learn the essential threading methods, such as `join()` and `isalive()`. You will also learn about the significance of the Daemon thread, GIL, and Locks.

Thread

Thread is like a process, but it takes less time and less memory to create as compared to a process. Multithreading helps create a parallel program. Threads can run on a single processor.

In some applications, multithreading fits better than multiprocessing. In this chapter, you will learn about the applications where thread suits better.

In Python, we use the Thread class to create threads. The Thread class is defined in the multithreading module. There are two ways to produce threads: one by inheriting the Thread class, and the other is by calling the Thread function.

Thread creation using class

To create a thread using the class function, we can inherit the Thread class defined in the threading module. Let's understand it using the following code:

```
import threading
class mythread(threading.Thread):
  def __init__(self,i):
    threading.Thread.__init__(self)
    self.h =i

  def run(self):
    print "Value send ", self.h

thread1 = mythread(1)
thread1.start()
```

Following is the details of the functions and methods used in the preceding code:

- `mythread(1)`: This is the new `mythread` class inherits the Python `threading.Thread` class.
- `__init__(self [,args])`: This is the constructor of the `mythread` class that overrides the constructor of the `Thread` class.

- `run()`: In this method, you can place your code logic or any different function or method, which can be called using the `run()` method.
- `start()`: Here, the Python Interpreter starts a thread by executing the `start()` method, which is defined in the `Thread` class.
- The user-created `mythread` class' constructor (`__init__`) overrides the base class constructor, so the base class constructor (`Thread.__init__()`) must be invoked.

Thread creation using function

Creating threads using function appears to be easy. Let's take a look at the following code:

```
import threading
def sum1(a,b):
  c = a+b
  print (c)


th1 = threading.Thread(target=sum1, args=(10,40))
th1.start()
```

The preceding code is straightforward; after creating a function, it is run by a thread. The `th1 = threading.Thread(target=sum1, args=(10,40))` statement produces a thread, `target=sum1` signifies the function to be called, and `args` specifies the Python tuple of the arguments that needs to be passed to the `sum1` function.

Important threading methods

Let's look at some critical methods defined in the threading module:

```
threading.activeCount()
```

The preceding code line returns the number of active Python `threads` at time of execution of `threading.activeCount()`:

```
threading.enumerate()
```

The preceding code line returns the list of the total Python threads that are currently active.

Let's understand this with the help of the following example:

```
import time
```

```
from threading import Thread,enumerate,activeCount
def sum1(x,y):
  time.sleep( 2 )
  c = x+y
  print (c)
th1 = Thread(target=sum1, args=(11,20))
th1.start()
th2 = Thread(target=sum1, args=(30,40))
th2.start()
print ("Active threads are ", activeCount())

print ("All Threads : ", enumerate())
```

Let's see the following screenshot for the output:

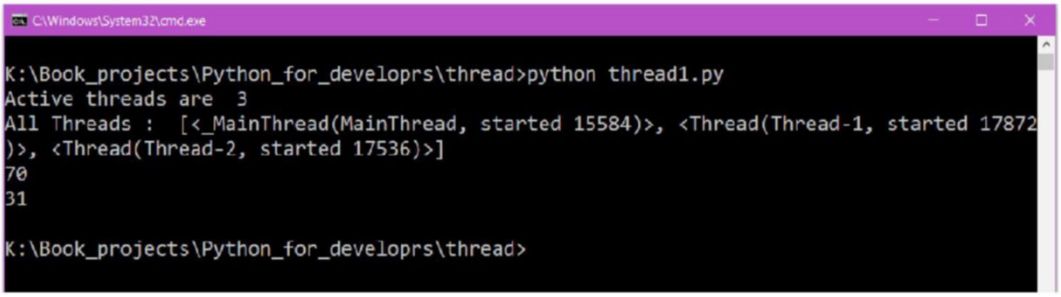

Figure 17.1

Now delete the `time.sleep(2)` line from preceding code and execute it again.

Have a look at the following screenshot for output, where the code is executed without sleep time:

```
K:\Book_projects\Python_for_developrs\thread>python thread1.py
22
27
Total number of threads 2
List of threads :  [<_MainThread(MainThread, started 14256)>]

K:\Book_projects\Python_for_developrs\thread>
```

Figure 17.2

You can see the difference. The main thread or the main process has run the `threading.activeCount()` and `threading.enumerate()` statements.

So, it's the main thread that runs the entire program. This time `th1` and `th2` were active in the figure when the `activeCount()` method was executed by the main thread. When we remove the sleep time and execute the program again, `th1` and `th3` may not be active.

The `threading.Timer()` method is used to set the time. Let's understand the following syntax:

```
threading.Timer(interval, function, args=[], kwargs={})
```

The preceding syntax means that after a specified interval in seconds, the interpreter will execute the function with `args` and `kwargs`.

The join method

In simple words, the `join()` method holds the main thread until it completes its task. Let's look at the following program before addressing the importance of the `join()` method:

```
from threading import Thread
import time
list_th = []
def test(num):
  time.sleep(2)
  list_th.append(num)
th1 = Thread(target=test, args=(1,))
th1.start()

th2 = Thread(target=test, args=(60,))
th2.start()

print ("List1 is : ", list_th)
```

The two threads are created with arguments in the preceding program. A global Python list, list1, is appended with the arguments. Let's see its outcome in the following screenshot:

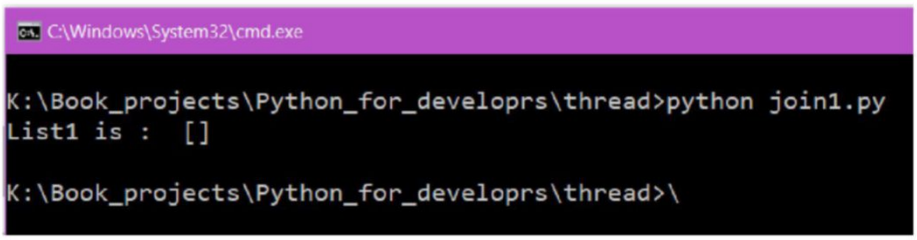

Figure 17.3

The preceding figure shows the blank list, but the list is supposed to be filled with values 1 and 60. The main thread executes the print ("`List1 is: ", list1_th`) statement. The threads were waiting for one second while the main thread printed the list before getting it filled.

Hence, the main thread has to be paused until all threads are finished. We will use the join method to do the same.

Let's change the code as follows:

```
from threading import Thread
import time
list_th = []
def test(num):
  time.sleep(2)
  list_th.append(num)
th1 = Thread(target=test, args=(1,))
th1.start()

th2 = Thread(target=test, args=(60,))
th2.start()
th1.join()
th2.join()

print ("List1 is : ", list_th)
```

Let's check the preceding code's output in the following screenshot:

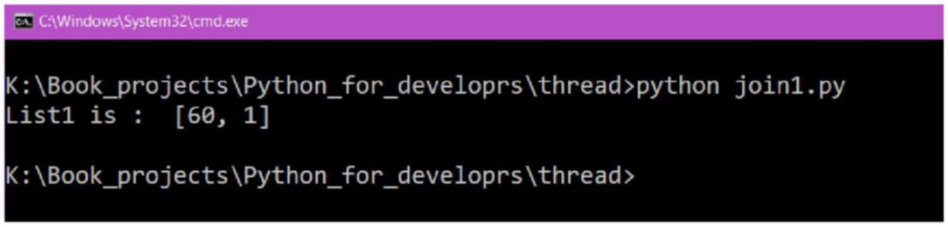

Figure 17.4

In the preceding code, the `th1.join()` syntax paused the main thread until thread `th1` finishes its task. Similarly, for thread `th2, th2.join()` paused the main thread. To attain parallelism, it is necessary to call the join method after the creation of all threads.

Let's take a look at more use cases of the join method:

```
import threading
import time
import datetime

t1 = datetime.datetime.now()
list1 = []
def fun1(a):
  time.sleep(1)
  list1.append(a)


list_thread = []
for each in range(10):
  thread1 = threading.Thread(target=fun1, args=(each,))
  list_thread.append(thread1)
  thread1.start()

for t in list_thread:
  t.join()

print ("List1 is : ", list1)
t2 = datetime.datetime.now()
print ("Time taken", t2-t1)
```

In the preceding code, we have used the for loop to create ten threads. Each thread has been appended in the `list_thread` list. After the creation of all the threads, the join method is used with the for loop. Let's see its output in the following screenshot:

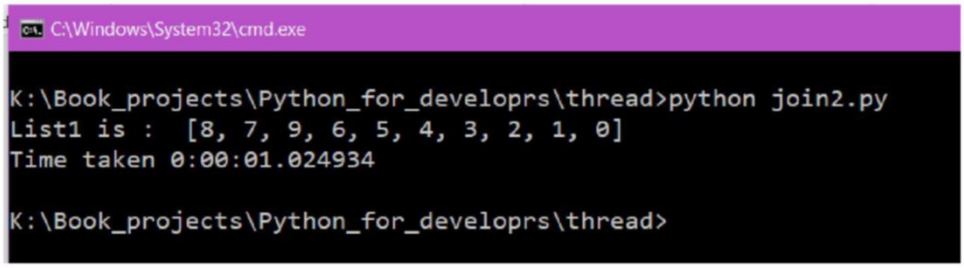

Figure 17.5

The program has taken 1 second to execute. As every thread was taking 1 second, and all the threads had started at the same time, they run in parallel. Consequently, the total time taken to execute is approximately 1 second. If the `join()` method of

each thread had been called after the creation of the corresponding thread, then the time taken to execute the entire program would be more than 10 seconds.

For further understanding, let's take a look at the following code:

```
import threading
import time
import datetime

t1 = datetime.datetime.now()
list1 = []

def fun1(a):
  time.sleep(1)
  list1.append(a)

for each in range(10):
  thread1 = threading.Thread(target=fun1, args=(each,))
  thread1.start()
  thread1.join()

print ("List1 is : ", list1)
t2 = datetime.datetime.now()
print ("Time taken", t2-t1)
```

Now let's look at the preceding code's output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\thread>python join3.py
List1 is :  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken 0:00:10.011321

K:\Book_projects\Python_for_developrs\thread>
```

Figure 17.6

From the preceding screenshot, it is obvious that the time it took is 10 seconds; this means that no parallelization was accomplished as the `join()` method of the first thread was called before the second thread was created, and so on.

The join method with time

In the following program, we used time delay with the `join()` method:

```
import time
import threading
def test():
  time.sleep(4)

thread1 = threading.Thread(target=test)
thread1.start()

thread1.join(2)
print (thread1.isAlive())
```

A couple of things are new here. If child thread is active during the main thread executing the `isAlive()` method , then the method returns `true`; otherwise, it returns `false`.

Check the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\thread>python join4.py
False

K:\Book_projects\Python_for_developrs\thread>
```

Figure 17.7

The output shows that at the time of print, the thread statement was not active. Pass the argument to the `join()` method, by changing `thread1.join()` to `thread1.join(2)`. This means that the join method can only block the main thread for two seconds.

Let's take a look at the following screenshot for the output:

```
K:\Book_projects\Python_for_developrs\thread>python join4.py
True

K:\Book_projects\Python_for_developrs\thread>
```

Figure 17.8

From the preceding figure, we can see that the thread was still alive as `join(2)` continued to block the main thread for two seconds, but it took the thread three seconds to complete its task.

The Daemon thread

In this section, you will learn about the non-daemon thread and its behavior. When the main thread exits, it terminates all the running daemon threads. For the non-daemon thread, until they finish their task, the main thread waits; however, for the daemon threads, the main thread does not wait.

Let's take an example of GUI windows, as shown in the following screenshot:

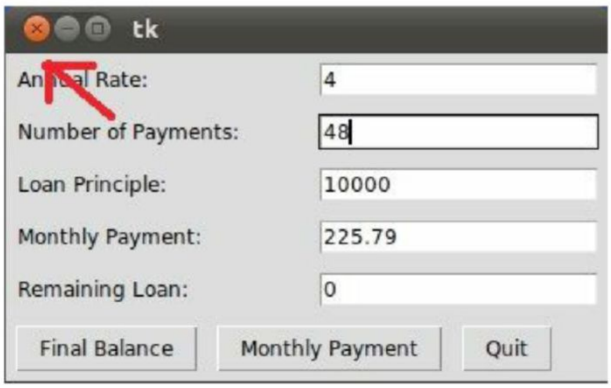

Figure 17.9

In the preceding figure, a calculator can be seen calculating something after getting an input.

This calculation is performed in the background, and it may take some time. If you click the close button, then two actions may take place:

- After clicking the Close button, the entire GUI Window exits
- After clicking the Close button, the GUI windows must wait until the background calculation gets completed

If the first action happens, then the daemon thread has been used for background calculation; and if the second action happens, then the non-daemon has been thread used.

Let's take a look at the following code to understand:

```
from threading import Thread
import time


def non_d():
  print ("Non daemon enters")
  time.sleep(1)
  print ("Non daemon exits")
```

```
def de():
  print (" daemon enters")
  time.sleep(3)
  print (" daemon exits")

th1 = Thread(target = non_d)
th1.start()

th2 = Thread(target = de)
th2.setDaemon(True)
th2.start()
```

Check the following screenshot for the output:

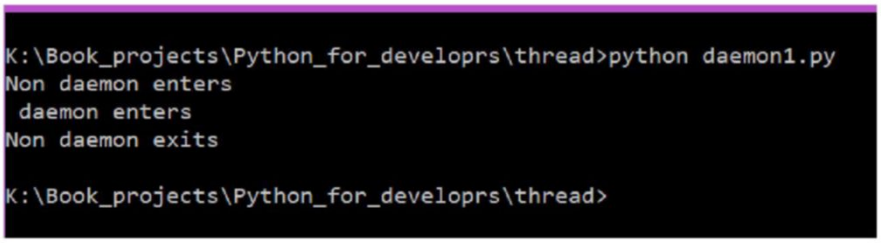

Figure 17.10

The `th2.setDaemon(True)` syntax or the `th2.daemon = True` syntax can be used to make the daemon thread.

In the preceding program, the daemon thread was expected to take three seconds to accomplish its task. As we know, the main thread attempts to exit without taking care of the daemon thread. Consequently, we did not get the **`daemon exits`** statement. In the next program, the `time.sleep(3)` statement has been removed from `de()` and added in the `non_d()` function.

Let's take a look at the following code:

```
from threading import Thread
import time
def non_d():
  print ("Non daemon enters")
  time.sleep(3)
  print ("Non daemon exits")

def de():
  print (" daemon enters")
```

```
  time.sleep(1)
  print (" daemon exits")

th1 = Thread(target = non_d)
th1.start()

th2 = Thread(target = de)
th2.setDaemon(True)
th2.start()
```

Here's the output for the preceding code:

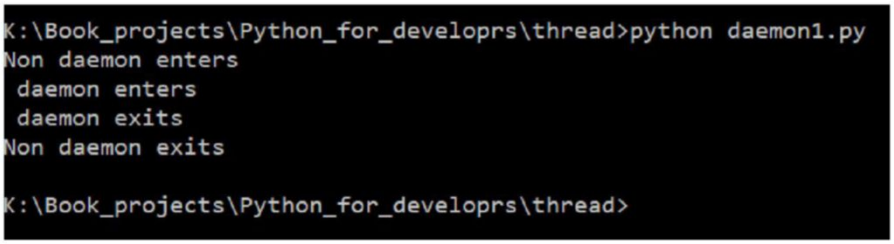

Figure 17.11

In the preceding program, the `non_d()` function was executed by the non-daemon thread, and it took 3 seconds to execute; thereby, the main thread had to wait for 3 seconds. Meanwhile, the daemon thread had executed the `de()` function completely.

Note: If a Daemon thread is used with its join method, then the join() method blocks the main thread until the Daemon thread finished its task.

Lock

In multithreading, lock is basically used for maintaining synchronization. Let's consider two threads trying to access a variable, such as a bank balance amount.

Let's suppose the first thread is initiated by withdrawing money from the ATM, and the second thread is started by a net banking APP. If the second thread accesses the balance amount before being updated by the first thread, then the balance will be in an inconsistent state. To prevent such situations, we use the lock mechanism. The lock can be only in two states – locked or unlocked. If the first thread puts the lock on the balance amount, then the second thread will have to wait until the first thread releases the lock.

The `lock.acquire()` method is used to acquire the lock. When the lock is acquired in one thread, then the `lock.acquire()` method blocks the other method until the lock is released. The `lock.release()` method is used to release the lock.

Let's take the following code as an example:

```
import time
from threading import Thread, Lock
lk = Lock()
list_th = []
def test(num):
  lk.acquire()
  list_th.append(num)
  lk.release()

for i in range(10):
  th1 = Thread(target= test, args=(i,))
  th1.start()

print ("The List is: ",list_th)
```

See the following screenshot for the preceding code's output:

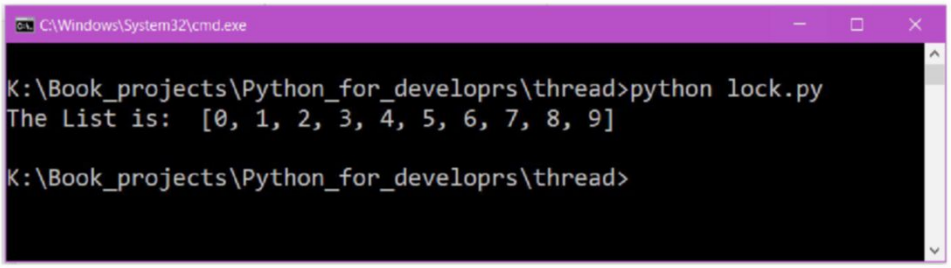

Figure 17.12

The `lk = Lock()` statement is used to make a lock object.

The primary issue with the lock is that it does not identify which thread has acquired the lock. Due to this behavior, two problems could occur.

Problem 1

Let's take a look at the following code:

```
from threading import Lock, Thread
import time
from datetime import datetime
lk = Lock()
```

```
t1 = datetime.now()
def second_test(num):
  lk.acquire()
  print (num)

def third_test():
  time.sleep(3)
  lk.release()
  print ("By 3rd Thread ")

th1 = Thread(target = second_test, args=("First_Thread",))
th1.start()

th2 = Thread(target= second_test, args=("Second_Thread",))
th2.start()

th3 = Thread(target= third_test)
th3.start()

th1.join()
th2.join()
th3.join()
t2 = datetime.now()
print ("Total time taken", t2-t1)
```

In the preceding program, the `th1` thread acquires the lock object, the `th3` thread releases it, and the `th2` thread is trying to acquire the lock. See the following screenshot for the output:

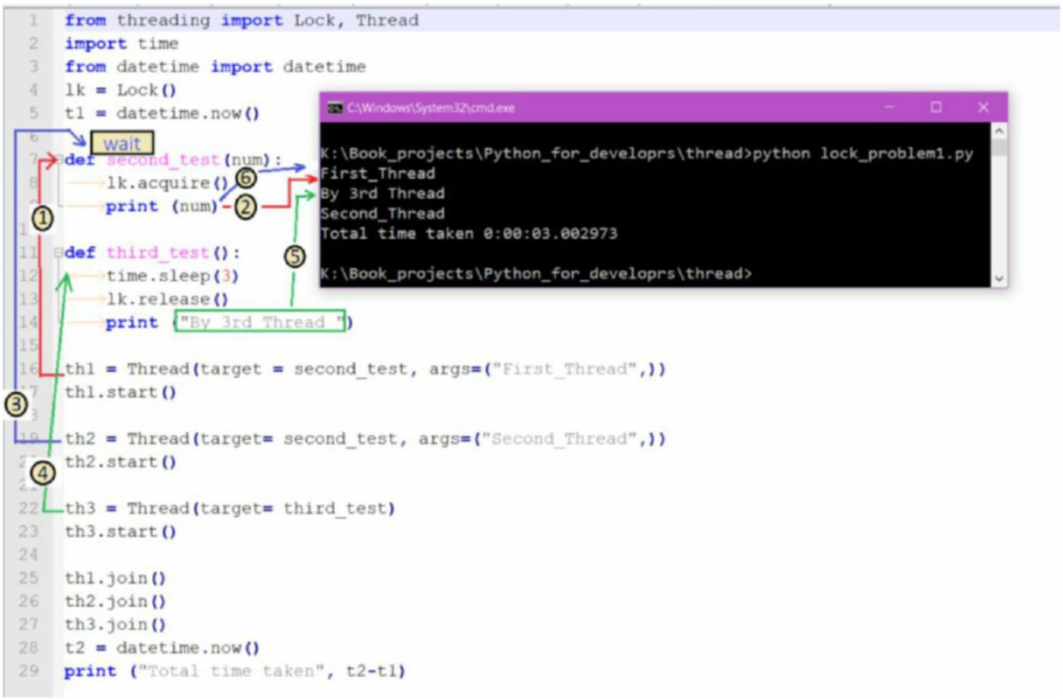

Figure 17.13

The preceding figure is showing the flow of execution. The `th1` thread acquires a lock and the `th2` thread waits for the lock to be released. The lock is released by the th3 thread, after which, the `th2` thread acquires the lock. The above execution shows the bug in the locks.

Problem 2

Let's take a look at the following code:

```
import threading
lk = threading.Lock()

def first_test(n1):
  print ("In first",n1)
  lk.acquire()
  num =12+n1
  lock.release()
  print (num)

def second_test(n2):
```

```
  lock.acquire()
  num = 12+n2
  lk.release()
  print (num)

def start():
  print ("starting")
  lk.acquire()
  first_test(50)
  second_test(60)
  lk.release()

th1 = threading.Thread(target= start)
th1.start()
```

If you execute the preceding program, there would be a deadlock. In the `start()` function, a thread acquires a lock; after acquiring the lock, the `first_test()` function is called. The thread sees the `lock.acquire()` statement as this lock is acquired by the same thread. But the lock does not identify the thread which has acquired it.

To solve these issues, we use the Reentrant lock (`RLock`).

Just replace `threading.Lock` with `threading.RLock`.

Now take a look at the following syntax:

```
lock = threading.RLock()
```

Let's look at the output of *Problem 1* and *Problem 2* after replacing the above line.

Output of *Problem 1:*

```
K:\Book_projects\Python_for_developrs\thread>python lock_problem1.py
Thread1
Exception in thread Thread-3:
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\threading.py", line 917, in _bootstrap_inner
    self.run()
  File "C:\ProgramData\Anaconda3\lib\threading.py", line 865, in run
    self._target(*self._args, **self._kwargs)
  File "lock_problem1.py", line 13, in third
    lock.release()
RuntimeError: cannot release un-acquired lock
```

Figure 17.14

In the preceding figure, the interpreter is giving an error because, with the Reentrant lock, it is not possible to release the lock until the thread acquires it.

The following screenshot shows the output of *Problem 2:*

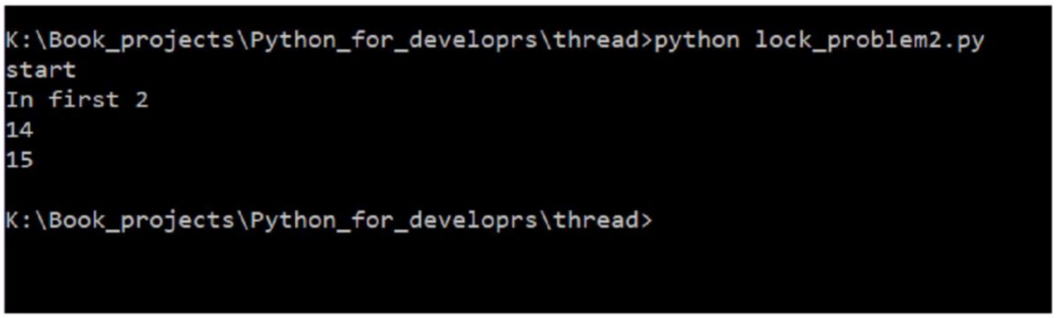

Figure 17.15

The `threading.RLock()` statement returns a modified lock called reentrant. The thread which obtains a reentrant lock must release it. Once a thread acquires a reentrant lock, it can be acquired again by the same thread without blocking.

Lock versus Rlock

The main difference between Lock and Rlock is that the object of Lock can only be acquired once; until it is released, it cannot be acquired again. (It can be re-acquired by any thread after it is released).

On the other side, an RLock's object can be obtained by the same thread several times; however, to make it "unlocked," it must be released the same number of times.

Another difference is that an obtained or acquired Lock can be released by any thread; whereas, an acquired RLock object can only be released by the thread that acquired it.

GIL

Thread-based parallelism is considered as a convenient fashion of writing parallel programs. The Python interpreter, however, is not completely thread-safe. A global lock, described as the **Global Interpreter Lock (GIL)**, is used to assist multithreaded Python programs.

This indicates that only one thread can execute the given code at one time; after a short period of time, the Python interpreter automatically switches to the next thread, or when a thread does something that may take some time. The GIL isn't enough in your own programs to prevent issues. If more than one threads try to access the same data for writing purposes, then the data may end up in an inconsistent or conflicting state.

Let's take a look at the `gil1.py` example:

```
import datetime
def calculate(n):
  t1 = datetime.datetime.now()
  while n > 0:
    n = n-1
  t2 = datetime.datetime.now()
  print ("time taken ", t2-t1)
calculate(100000000)
```

In the preceding code, the calculate function is being run by the main thread. Let's see the time taken by the thread:

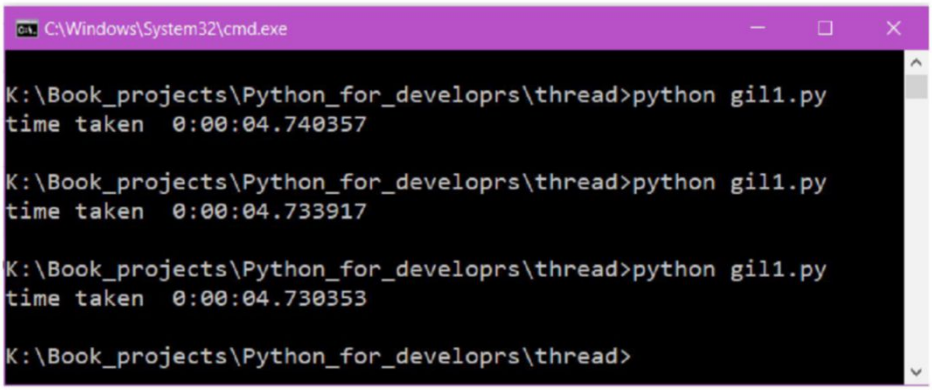

Figure 17.16

Based on the preceding output, it can be deduced that the execution time is around 5 seconds.

Let's see the threaded version of the code:

```
from threading import Thread
import datetime
def calculate(n):
  while n > 0:
    n = n-1

def calculate1(n):
  while n > 0:
    n = n-1
```

```
t1 = datetime.datetime.now()
th1 = Thread(target = calculate, args=(100000000, ))
th2 = Thread(target = calculate1, args=(100000000, ))

th1.start()
th2.start()

th1.join()
th2.join()
t2=datetime.datetime.now()
print ("Time taken ", t2-t1)
```

In the preceding code, two threads have been created to run parallelly.

Let's see its result in the following screenshot:

```
K:\Book_projects\Python_for_developrs\thread>python join4.py
False

K:\Book_projects\Python_for_developrs\thread>
```

Figure 17.17

You can see that the script in the preceding figure lasted nearly 9.5 seconds, which is twice the time of the earlier script. This means that only the main thread acts as multithreading, as shown in the following figure:

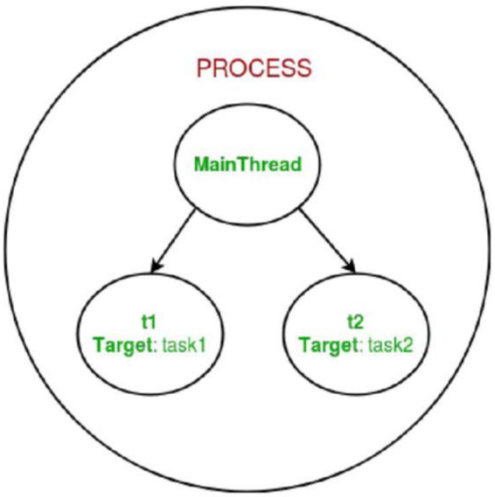

Figure 17.18

We can say from the preceding experiment that multithreading is described as a processor's capacity to run various threads simultaneously.

In a single-core CPU, it is accomplished using repeated switching between threads; this is referred to as context switching. The state of a thread is preserved in context switching, and the state of another thread is resumed or loaded whenever an interruption occurs (due to I/O or manual setting).

Context changing occurs so often that all threads seem to run parallel to each other (which is called **multitasking**).

Where to use multithreading?

The big question is: where to use multithreading? Let's discuss a mathematical case to answer this question.

- **Ta:** Total time to execution of program
- **Ti:** Internal delay of program or Input/Output (I/O) Bound
- **Tt:** Time taken by thread or CPU Bound

Tt: Ta - Ti

What is internal delay?

Let's consider you are sending the ping packet to a machine, as shown in the following code:

```
import os
import time
t1 = time.time()
resp=os.popen("ping -n 1 14.139.242.109")
print (resp.read())
t2 =time.time()
t = (t2-t1)*1000
print ("Time take %d in milliseconds"%( t))
```

Let's check its output in the following screenshot:

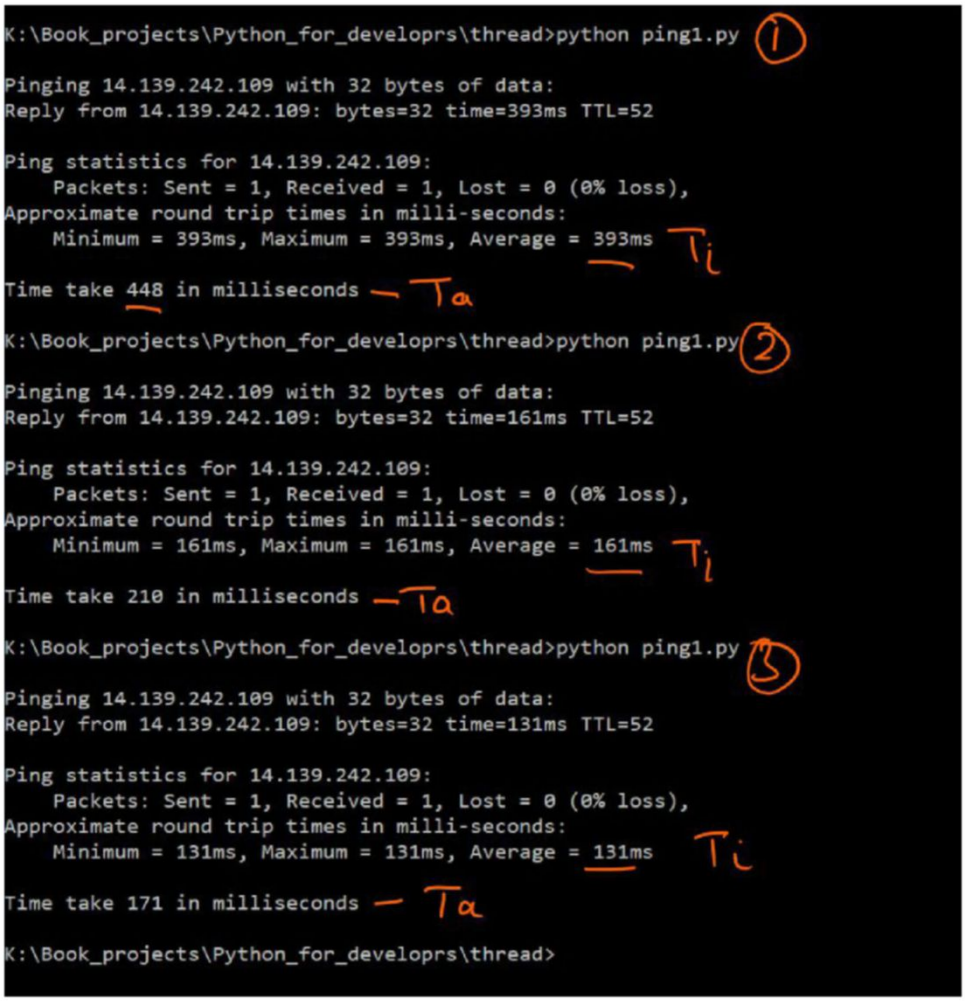

Figure 17.19

In the preceding screenshot, you can see `Ti` is the round-trip time that depends on the network bandwidth and the number of gateways or routers in the path.

Let's take a look the `Ta, Ti,` and `Tt` statements from the preceding output here:

```
First run

Ta = 448

Ti = 393

Tt = 448-393 => 55
```

```
Second Run
Ta = 210
Ti = 161
Tt = 210-161 => 49

Third Run
Ta = 171
Ti = 131
Tt = 171-131 => 40
```

Now you know what total time, internal delay, and time taken by threads.

If *Ti (I\O Bound) = 0*, then multithreading is not useful.

If *Ti* is present, then multithreading is useful. As the external conditions produce the internal delay, the thread uses this time for context switching.

If we need to send ping packets to 100 IP, then multithreading would be useful.

How many threads?

The next big question is, how many big threads can be used?

Let us assume `Ti` is the same for each task. If we send ping to 100 IPs, we can assume that the `Ti` for each ping packet is the same.

Hence, the equation would be:

$$Ti \geq N*Tt$$

Let's consider `Ti` is 100ms and the time taken by the thread is 20ms. You can create maximum 5 threads.

Let's run the `gil1.py` and `gil2.py` codes with some changes, as shown here:

```
import datetime
import time
def count(n):
  t1 = datetime.datetime.now()
  while n > 0:
    n = n-1
    time.sleep(.01)
  t2 = datetime.datetime.now()
```

```
  print (t2-t1)

count(1000)
```

In the preceding code, we have added the sleep(.01) time, which acts like an internal delay.

Let's take a look at the following screenshot for the output:

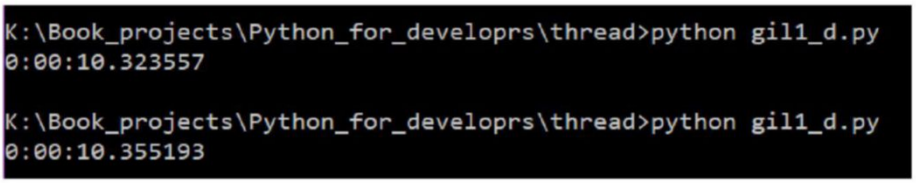

Figure 17.20

Now let's see the multithreaded version of the preceding program:

```
import datetime
from threading import Thread
import  time
def calculate(n):
  while n > 0:
    n = n-1
    time.sleep(.01)
def calculate1(n):
  while n > 0:
    n = n-1
    time.sleep(.01)

t1 = datetime.datetime.now()
thread1 = Thread(target = calculate, args=(1000,))
thread2 = Thread(target = calculate1, args= (1000,))

thread1.start()
thread2.start()
thread1.join()
thread2.join()

t2 = datetime.datetime.now()
print (t2-t1)
```

Here's the preceding code's output:

```
K:\Book_projects\Python_for_developrs\thread>python gil2_d.py
0:00:10.522793

K:\Book_projects\Python_for_developrs\thread>python gil2_d.py
0:00:10.515704

K:\Book_projects\Python_for_developrs\thread>
```

Figure 17.21

Now, multithreading has been successful. If there were no internal delays, then the time would have been doubled.

In Python 3.2, a new topic called `ThreadPoolExecutor` has been introduced; it provides a simple high-level interface for asynchronously executing input/output bound tasks. The `ThreadPoolExecutor` command is not in the threading module; however, it is present in the `concurrent.futures` module. Let's take a look at the following example:

```
import datetime
import concurrent.futures
import  time
t1 = datetime.datetime.now()
def calculate(n):
  while n > 0:
    n = n-1
    time.sleep(.01)
  return "Done first"
def calculate1(n):
  while n > 0:
    n = n-1
    time.sleep(.01)
  return "Done Second"

with concurrent.futures.ThreadPoolExecutor() as exe:
  f1 = exe.submit(calculate, 1000)
  f2 = exe.submit(calculate1, 1000)
  print (f1.result())
  print (f2.result())
```

```
t2 = datetime.datetime.now()
print (t2-t1)
```

In the preceding code, the submit method schedules the calculate function to be executed and to return a future object. The `f1.result()` method grabs the return value of the corresponding function and the `f1.result()` method also waits around until the `calculate()` function gets completed. The `result()` method works like the `join()` method of the thread. See the following output for an example:

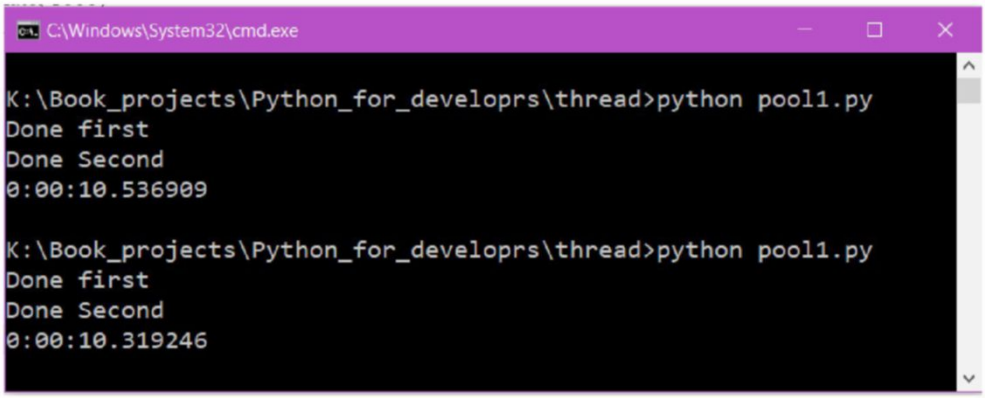

Figure 17.22

You can see the program is taking around 10 seconds. Let's consider you want to run the same function 10 times, for which, you will use the list comprehension. Let's take a look at the following full code:

```
import datetime
import concurrent.futures
import  time
t1 = datetime.datetime.now()
def calculate(n):
  while n > 0:
    n = n-1
    time.sleep(.01)
  return "Done first"

with concurrent.futures.ThreadPoolExecutor() as exe:
  response = [exe.submit(calculate, 1000) for i in range(10)]
```

```
  for  each in concurrent.futures.as_completed(response):
    print (each.result())


t2 = datetime.datetime.now()
print (t2-t1)
```

In the preceding code, the `[exe.submit(calculate, 1000) for i in range(10)]` statement runs the `submit()` method 10 times with the same argument. To get a result, we use an `as_completed()` function of the `concurrent.futures` module. This will give us an iterator that we can loop over, and yield the result of thread as complete, as shown in the following output:

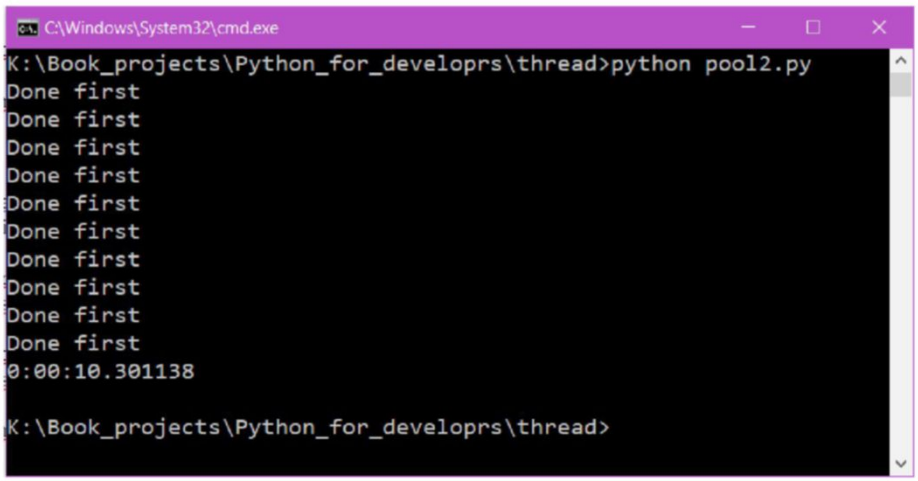

Figure 17.23

In preceding output, the total time taken is around 10 seconds. Now, instead of passing 1000 to the calculate function, let's pass a different argument to the calculate function, as shown in the following code:

```
import datetime
import concurrent.futures
import  time
t1 = datetime.datetime.now()
def calculate(n):
  k = n
  while n > 0:
    n = n-1
    time.sleep(.01)
  return "Done first "+str(k)
```

```
list1 = [1000,100, 900, 800]
with concurrent.futures.ThreadPoolExecutor() as exe:
  response = [exe.submit(calculate, i) for i in list1]

  for  each in concurrent.futures.as_completed(response):
    print (each.result())

t2 = datetime.datetime.now()
print (t2-t1)
```

In the preceding code, a list of numbers has been passed to the calculate function. Let's run the code and analyze its output, as shown here:

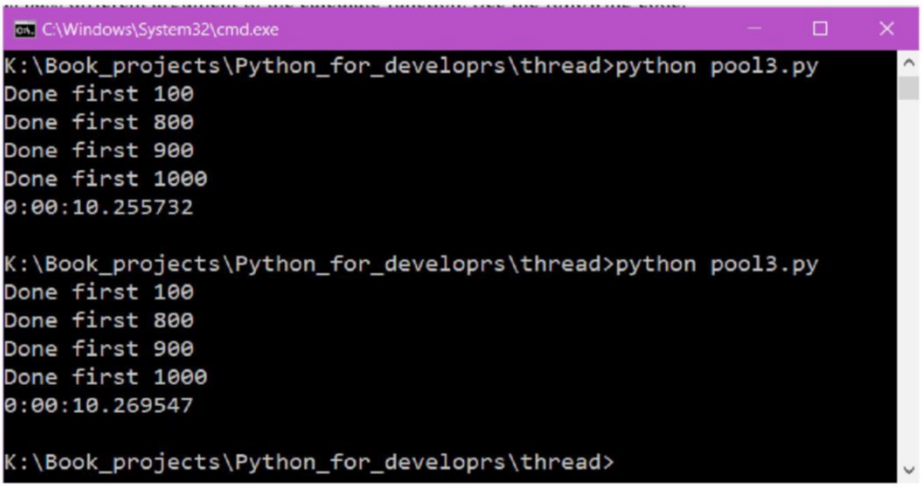

Figure 17.24

In the preceding code, the threads are printed in the order they get completed because we used the `concurrent.futures.as_completed()` function. We can also use the `map()` function to pass each argument to the submit method. Take a look at the following example with the `map()` function:

```
import datetime
import concurrent.futures
import  time
t1 = datetime.datetime.now()
def calculate(n):
  k = n
  while n > 0:
    n = n-1
```

```
    time.sleep(.01)
  return "Done first "+str(k)
list1 = [1000,100, 900, 800]
with concurrent.futures.ThreadPoolExecutor() as exe:
  response = exe.map(calculate, list1)

  for  each in response:
    print (each)

t2 = datetime.datetime.now()
print (t2-t1)
```

The `map()` function returns the iterator that can be looped over to get the result. The iterator in turn returns the result as the thread starts, as shown in the following screenshot:

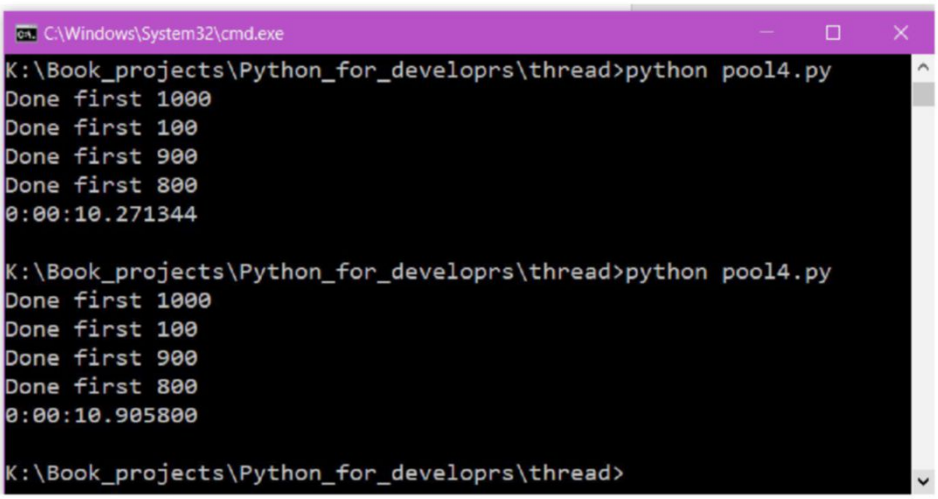

Figure 17.25

You can compare the output, now they are giving the result as they started instead of giving the output as threads completed its task.

Conclusion

In this chapter, you learned about the thread-based program. The threading module contains the Thread class, which creates the thread. The join method blocks the main thread until the child thread finishes its task. The Daemon thread allows us to create applications that could be terminated after closing the main program. The daemon

thread has to exit when the main thread exits. If multiple threads try to access the same data, then the data may end in an inconsistent state. To maintain consistency, we use the lock mechanism. At the end of the chapter, you have seen the GIL that enables the Python interpreter, which is to be managed by only one thread. In the next chapter, you will learn about queues and queues with thread.

Questions

1. What is the purpose of the run() method?
2. What is the meaning of join(5)?
3. Does lock remember which thread acquires the lock?
4. Why do we use daemon threads?
5. What are the cases where we can use multi-threading?

Chapter 18
Queue

In our day-today lives, we can see the different type of queues, like a customer waiting for some service. In most of the cases, the customer is attended to, on a first come, first server basis. At supermarkets, a polite customer might let someone having only a few items, go first. The queuing policy decides who goes next. In this chapter, we will learn about the different types of queues offered by Python. Python queues are very useful in a multithreaded program.

Structure

- Queue
    - FIFO queue
    - LIFO queue
    - Priority queue
- Queue with threads

Objective

In this chapter, we will learn about the queue implementation in Python. We will see FIFO, LIFO, and Priority Queue. After learning this chapter, you will be able to make efficient multithreaded applications. You will learn the producer-consumer

concept. A lot of companies use the producer-consumer concept with queue in their production. Although they are using third party queues, the basic idea is same.

Queue

A queue is a linear data structure with two ends. One end is called **head** (Front), and the second end is called the **tail** (Rear). The item is added from the tail end and gets eliminated as per the policy to be used. We would see three policies that define how to pop the items. In queue, we use terms push and pop; pushing an item means inserting an item, and popping an item means delete and obtain the item.

Python queue is of three types:

- FIFO (First in First Out)
- LIFO(Last in First Out)
- Priority

Let us understand the above-mentioned queues one by one.

FIFO queue

The most straightforward queuing system is called **FIFO**, and it stands for **first-in-first-out**. This is the simplest type of queue. The value we enter in the queue first, will get first, as showcased in the following diagram:

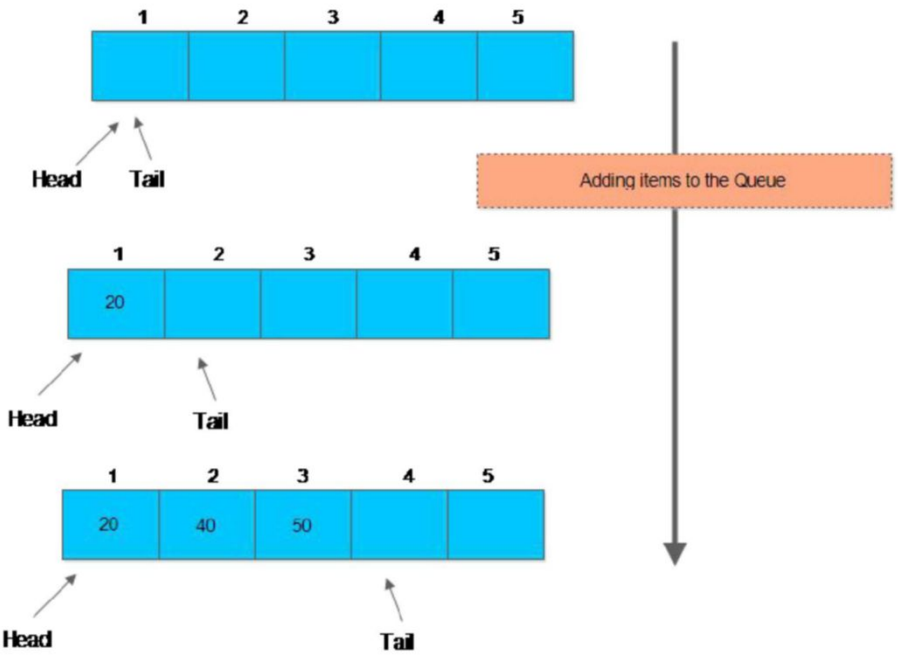

Figure 18.1

In the preceding figure, the insertion of an item in the queue is illustrated. The following diagram is showcasing the obtaining as well as the removal of the item:

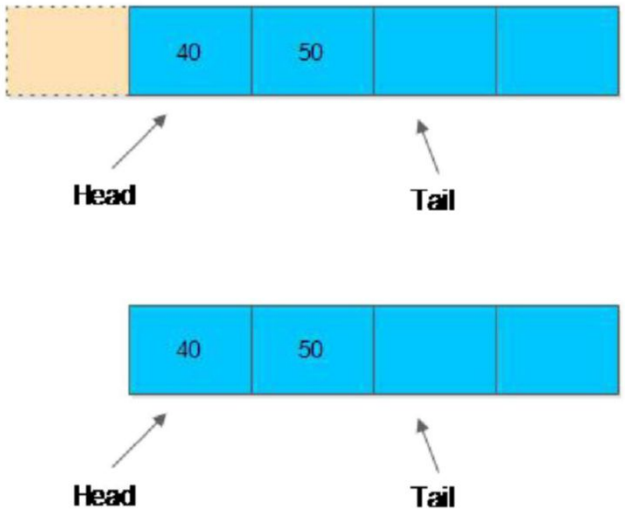

Figure 18.2

When we obtain the item pointed by the head, then the head of the queue shifts its place to the next items, as shown in the preceding diagram.

Let us see the FIFO implementation in Python:

```
import queue
Q1 = queue.Queue()  # FIFO
for each in range(5):
Q1.put(each)
while not Q1.empty():
print (Q1.get())
```

In the preceding code, syntax `Q1 = queue.Queue()` creates a FIFO queue object. The queue is the module and `Queue` is the class. The syntax `Q1.put()` put the item in the queue `Q1`. The `Q1.empty()` returns `True`, if the queue is empty. If there is something in the queue, then `Q1.empty()` returns `False`. The syntax `Q1.get()` is used to retrieve the item.

Check the output in the following screenshot:

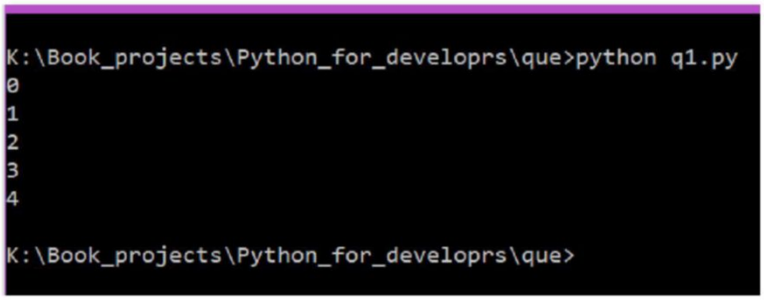

Figure 18.3

Now let us discuss the Python LIFO queue policy.

LIFO queue

LIFO queue uses the last-in, first-out ordering (generally associated with a stack data structure). The push and pop operations on an item are performed at the same end.

See the following implementation of the LIFO code:

```
import queue
Q1 = queue.LifoQueue()
for each in range(5):
Q1.put(each)
while not Q1.empty():
print (Q1.get())
```

Check the following screenshot for the output:

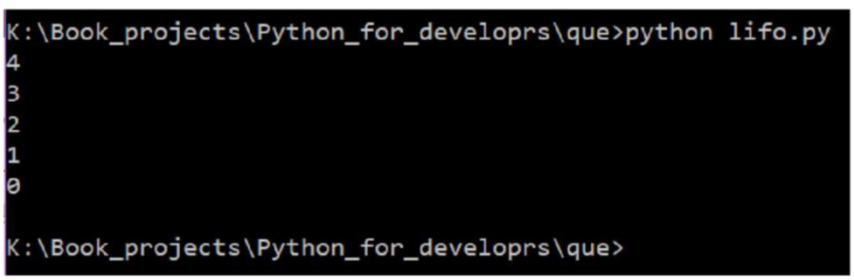

Figure 18.4

In the preceding output, Last entered value come first, like a stack of disks.

Priority queue

Let us see the priority-based queue. In the priority-based queue, we enter a tuple that contains the item and priority number, lower the number higher the priority.

Let us see the code:

```
import queue
Q1 = queue.PriorityQueue()
Q1.put((5,"Mohit"))
Q1.put((1,"admin"))
Q1.put((6,"Sahil"))
Q1.put((10,"bhaskar"))
Q1.put((45,"ajay"))
while not Q1.empty():
print (Q1.get()[1])
```

In the preceding code, we have pushed the tuples. Each tuple contains two items, as showcased below:

```
(Priority number, data)
```

See the following screenshot for the output:

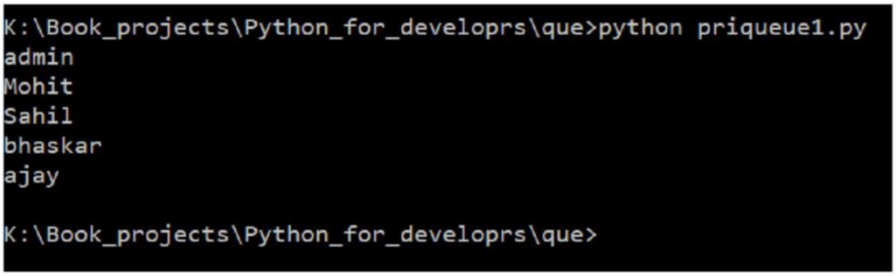

Figure 18.5

As you can see now, we are gettingthe value according to the priority.

Queue with threads

So far, you have seen the thread and the queue. In this section, we will learn how two or more threads can communicate with the help of a queue.

Let us see the producer and consumer problem. The program is slightly long. We would see the explanations section by section.

In the following section, the mandatory modules have been imported:

```
import time
import random
import queue
from threading import Thread
```

Here, we have defined the queue `q1`:

```
q1 = queue.Queue()
```

The `producer1` function, which is run by the thread `th2`,creates a random number and puts the number in the queue:

```
def producer1():
while True:
a = random.randint(1,100)
time.sleep(1)
q1.put(a)
print ("Size ",q1.qsize())
```

The `consumer` function, which is run by the thread `th2`, consumes the data of the queue:

```
def consumer():
while True:
print ("GOT ", q1.get())
time.sleep(1)
```

The two threads,`th1` and `th2`, which execute the `producer1` and `consumer` functions, respectively.

```
th1 = Thread(target = producer1)
th1.start()

th2 = Thread(target= consumer)
th2.start()
```

See the following screenshot for the output:

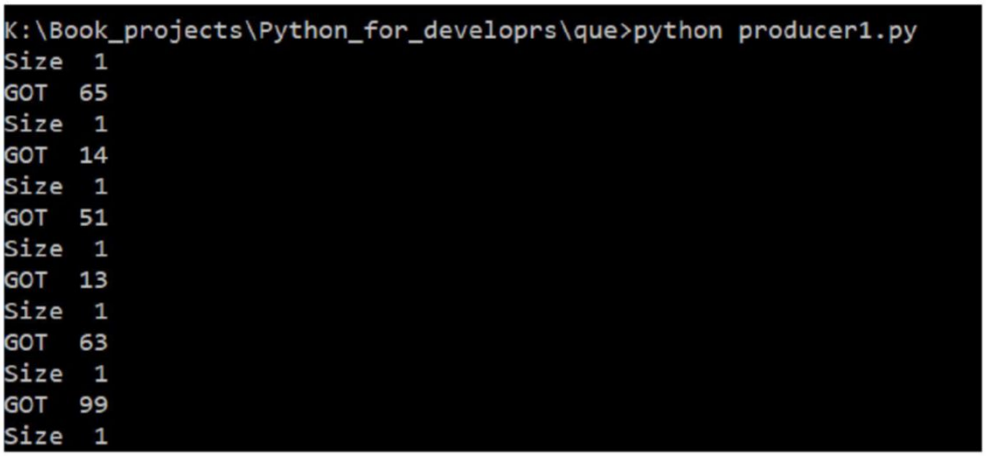

Figure 18.6

The preceding output shows everything in synchronization. The producer is filling the queue at the rate of one item per second. The consumer is consuming the queue at a rate of 1 second. Now, let us change the rate of production and consumption.

Just change one line, `time.sleep(.5)` in the `producer1` function.

Let us see the output in the following screenshot:

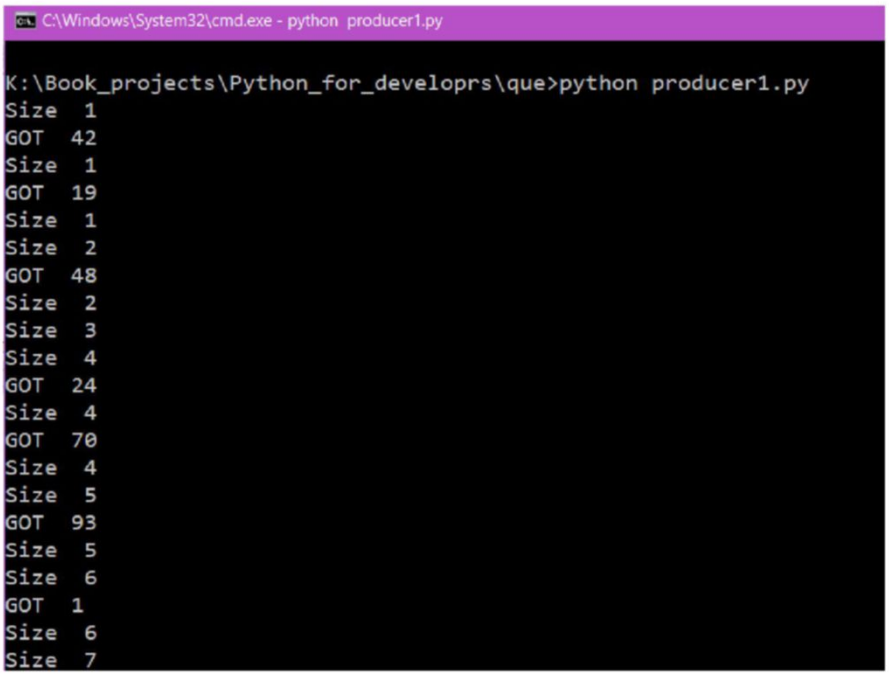

Figure 18.7

From the preceding output, it is clear that the size of the queue is being increased continuously. Now, in such a situation, either optimize the code, or increase the number of threads to establish the sync. See the following code:

```
import time
import random
import queue
from threading import Thread

q1 = queue.Queue()

def producer1():
while True:
a = random.randint(1,100)
time.sleep(.5)
q1.put(a)
print ("Size ",q1.qsize())

def consumer():
while True:
print ("GOT ", q1.get())
time.sleep(1)

th1 = Thread(target = producer1)
th1.start()

th2 = Thread(target= consumer)
th2.start()

th3 = Thread(target= consumer)
th3.start()
```

Check the following screenshot for the output:

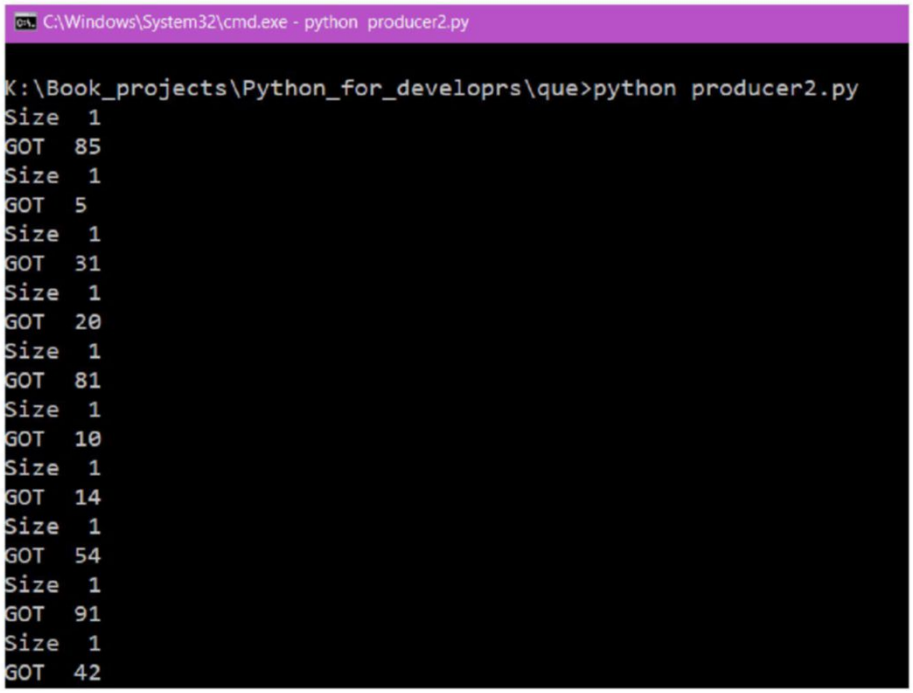

Figure 18.8

Now the `producer` and the `consumer` are in synchronization.

Let us see one more problem.

```
import queue
from threading import Thread
import time
q1 = queue.Queue()

def fun1():
while not q1.empty():
print (q1.get())
for each in range(5):
q1.put(each)
th1 = Thread(target = fun1)
th1.start()
```

The preceding program is verystraightforward. The queue is getting filled by the for loop, and the thread is consuming the items of the queue.

Check the following screenshot for the output:

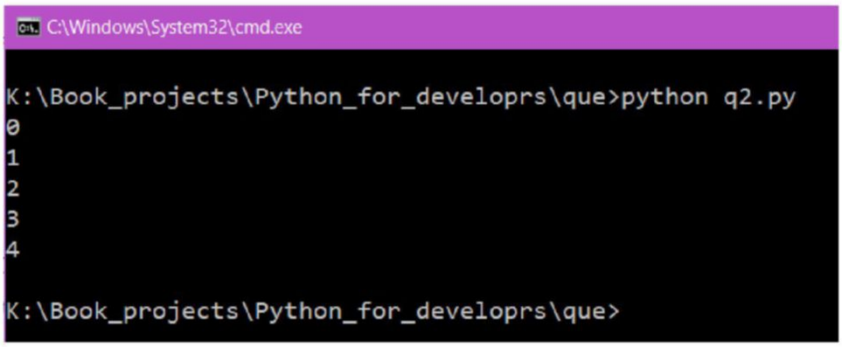

Figure 18.9

In the preceding code, the queue is getting filled before the thread begins.

Let us fill the queue after the creation of a thread:

```
import queue
from threading import Thread
import time
q1 = queue.Queue()
def fun1():
while not q1.empty():
print (q1.get())

th1 = Thread(target = fun1)
th1.start()
for each in range(5):
q1.put(each)
```

The following screenshot is showcasing the output:

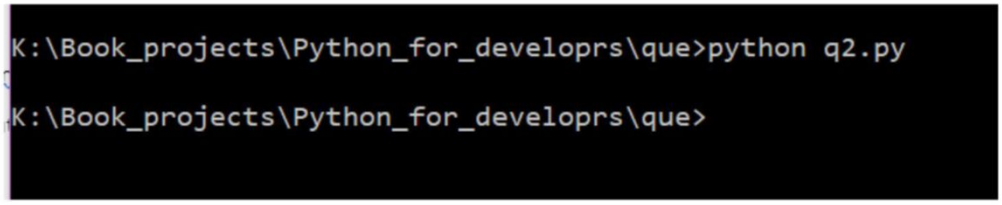

Figure 18.10

We are not getting anything, because the condition, `q1.empty()`, returns `true`, a thread is created before filling the queue. The following are the possible solutions with problems:

- If we use the `while True` loop, then we would get every element, but the program would not get terminated.
- If we use the `while True` loop with the daemon thread, then the main thread exists before the daemon thread finishes its task.
- If we use `while True` loop with daemon thread and join, then the Program will not get terminated.

We need something that will block the main thread until all the items of the queue have been popped.

Here we will use join with the queue, not with thread. See the following code:

```
import queue
from threading import Thread
import time
q1 = queue.Queue()
def fun1():
while True:
print (q1.get())
q1.task_done()
th1 = Thread(target = fun1)
th1.setDaemon(True)
th1.start()

for each in range(5):
q1.put(each)
q1.join()
```

Check the following screenshot for the output:

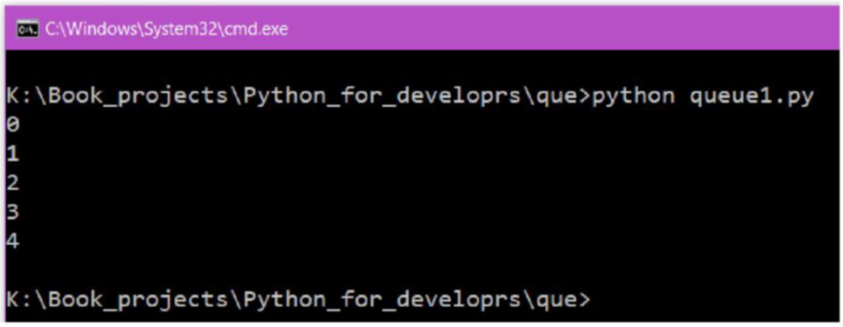

Figure 18.11

The `join()` method blocks until all items in the Python queue have been poppedand processed. Whenever an item is added to the Python queue, the count of unfinished tasks goes up. Whenever a consumer thread calls `q1.task_done()`, the count goes down. It is indicating that the item was retrieved, and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Conclusion

In this chapter, we have studied the queue. In Python 3, the queue is amodule, and the queue is the FIFO queue. The FIFO is the default case of a queue. The LIFO queue is the implementation of a stack. The third type is the priority queue,in which a tuple with two items (Priority number, data) is pushed. The item, in the form of a tuple, is retrieved according to the priority number. Threads can take the help of the queue to communicate with each other, as we have seen in the example of `producer` and `consumer`. At the end of the chapter, we have learned about the `join` method and `task_done`. In the next chapter, we will learn about multiprocessing and subprocess.

Questions

1. What is significance of the task_done method?
2. Which queue policy supports the stack?

Chapter 19
Multiprocessing and Subprocess

Multiprocessing refers to the simultaneous execution of multiple processes on a distinct CPU or core in a computer system. It is the ability of a system to support more than one processor within a single computer system. Such multi-processors often share the machine bus as well as the clock, storage, and peripheral devices. The multi-processor system's main advantage is to get more jobs done in a shorter time frame. Such types of devices are used when storing a considerable volume of data that requires extremely high speed. Compared to single-processor systems, multi-processing systems can save costs as processors and can share peripherals and power supplies. In this chapter, you will learn how to create multiprocessing with Python.

Structure

- Python multi-processing
    - The Process class
    - The Daemon process
- The communication between the processes
    - File
    - Shared memory
    - Communication channel

- Subprocess
  - Call
  - popen

Objective

This chapter will help us create parallel programs—where thread fails, we use multi-processing. You will learn how creation, execution, and communication between the processes works, and you will understand why communication between processes is expensive than threads. You will also learn about the subprocess module to replace `os.system` and `os.popen`.

Python multi-processing

The multi-processing of Python is similar to multithreading; Python multi-processing creates subprocesses in the place of threads. By using subprocess, multi-processing evades the GIL (Global Interpreter Lock).Additionally, it runs on both Unix and Windows and can take advantage of multicore processors.

The Process class

The processes are generated by creating an object of the `Process` class(defined in the multi-processing module),and they are started by calling the `start()` method. Let's gain a deeper understanding of it through examples of how the process class works. Consider the following code:

```
from multiprocessing import Process
def call_name(name):
    print ('hello ', name)

if __name__ == '__main__':
    p = Process(target=call_name, args=('mohit',))
    p.start()
```

Now, check out itsoutput in the following screenshot:

```
K:\Book_projects\Python_for_developrs\multiprocessing>python multi_example1.py
hello  mohit

K:\Book_projects\Python_for_developrs\multiprocessing>
```

Figure 19.1

We just created one process in the preceding example. The syntax of `multiprocessing` is very similar to multithreading. Let's create multiple processes using the following code:

```
from multiprocessing import Process
def worker(n):
    print ('process:', n)


if __name__ == '__main__':
    for i in range(5):
        p = Process(target=worker, args=(i,))
        p.start()
```

Now let's check its output in the following screenshot:

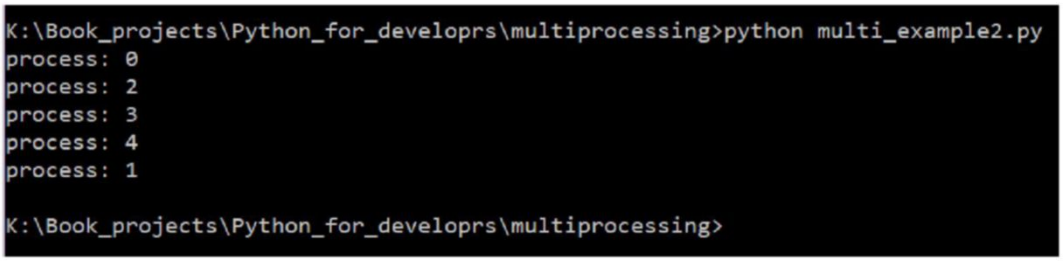

Figure 19.2

In the preceding example, five processes are created.

If you want to know about the current process and the process ID, then use the `current_process` class.

Let's give each process a name and get its process id. In order to get a name and process id, we will use `current_process()`.

Let's look at an example:

```
from multiprocessing import Process, current_process
def worker(n):
print ('process:', n)
print (current_process().name)
print (current_process().pid)


if __name__ == '__main__':
```

```
p1 = Process(name ='worker1', target=worker, args=(1,))
p2 = Process(name ='service', target=worker, args=(2,))
p1.start()
p2.start()
```

Here's its output:

```
K:\Book_projects\Python_for_developrs\multiprocessing>python current_process1.py
process: 1
worker1
13088
process: 2
service
27412

K:\Book_projects\Python_for_developrs\multiprocessing>
```

Figure 19.3

By using `current_process()`, we are printing the process IDand the process name.

In the multithreading chapter, we saw the use of the `join()` method. In multiprocessing, the `join()` method works in the same way. The `join` method will pause the main process until the child process completes its task. Let's look at another example:

```
from multiprocessing import Process, current_process
from multiprocessing import Process, current_process
def worker(n):
print ('process:', n)
print (current_process().name)
print (current_process().pid)

if __name__ == '__main__':
p1 = Process(name ='worker1', target=worker, args=(11,))
p2 = Process(name ='service', target=worker, args=(12,))
p1.start()
p2.start()
p1.join()
p2.join()
print ("Task finished")
```

Here's what its output looks like:

```
K:\Book_projects\Python_for_developrs\multiprocessing>python multiprocess_join1.py
process: 11
worker1
process: 12
22060
service
13160
Task finished

K:\Book_projects\Python_for_developrs\multiprocessing>
```

Figure 19.4

The `join()` method blocks the main process. If we don't give an argument to the `join()` method then `join()` waits for an indefinite time. If you specify the time in seconds to the join method, then the join method pauses the main process for the specified time.

Killing a Process

In order to kill a process, the `terminate()` method is used, and in order to check if a process is live or not, the `is_alive()` method is used. Let's look at an example:

```
from multiprocessing import Process, current_process
import time
def worker1(num):
time.sleep(3)
print (current_process().name)


if __name__ == '__main__':
p1 = Process(name ='worker1', target=worker1, args=(1,))
p1.start()
print ("p1 process After start ", p1.is_alive())
p1.terminate()
time.sleep(1)
print ("p1 process after termination ", p1.is_alive())
print ("Task Done")
```

Let's see the output in the following screenshot:

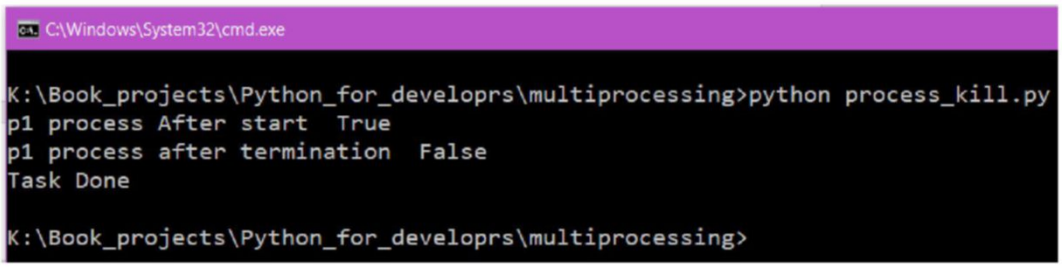

Figure 19.5

As you can see in the preceding screenshot, after the process was completed, the background machinery was given 1 second to update the object's status to reflect the termination (you can also use the `join()` method instead of `time.sleep()`).

The Daemon process

The concept of the daemon process has already been explained in *Chapter 17, Multithreading*. Please refer to the Daemon thread concept to understand the Daemon process.

The communication between the processes

The Python processes do not share the usual lists and dictionary; however, the list and dictionary created by the process manager is shared by the processes.

Let's now look at a process with a regular list:

```
from multiprocessing import Process, Manager,current_process
list1 = []
def worker(list1, num):
list1.append(num)
print ("list1 by ",current_process().name, list1 )

if __name__ == '__main__':

p1 = Process(name ='worker1', target=worker, args=(list1,1))
p2 = Process(name ='service', target=worker, args=(list1,2,))
p1.start()
p2.start()
```

```
p1.join()
p2.join()
print ("list is ", list1)
print ("All finished")
```

Now let's check out the output for the preceding code:

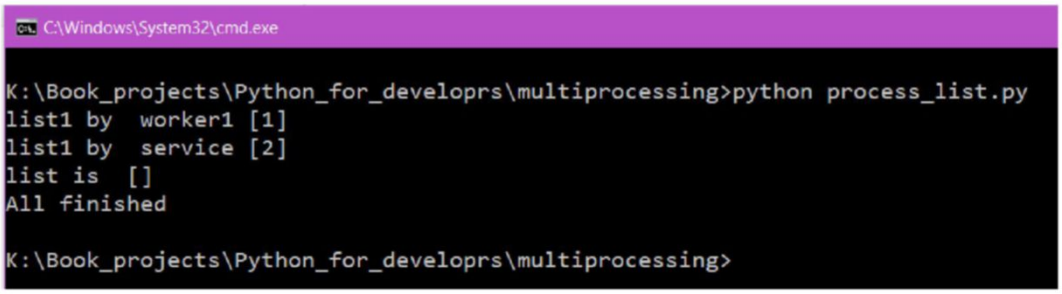

Figure 19.6

As you can see, every list is different. The lists used by `worker1`, `service`, and `main` processes are separate. That's why each list contains different values. Each process has its own address space; thus, the processes do not share the variables. So, the **interprocess communication (IPC)** will be used to share the data among the processes. For more clarification, take a look at the following diagram:

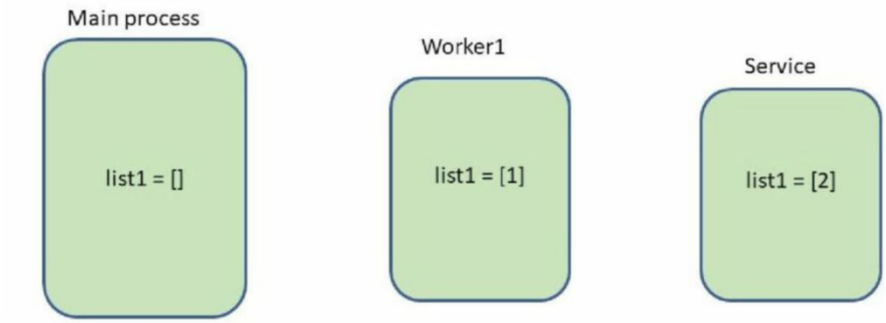

Figure 19.7

The question is: how can two processes share the data? They can share the data via three options, which are:

- File
- Shared memory
- Communication channel

The file option involves the harddisk operation, where one process writes the data to file and the other process reads the data from the file. Let's understand the concept of shared memory and communication channel with the help of the following figure:

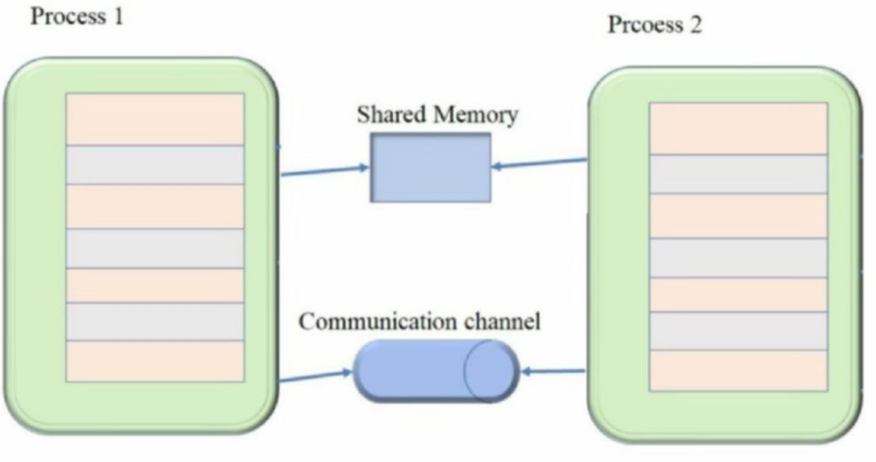

Figure 19.8

Shared memory

In the multiprocessing communication,`Value`, `array`, and `manager` are used to share the state. First,you will learn about `Value`.

Value

We use Value to store a single number using the following syntax:

```
Value(typecode_or_type, *args, lock=True)
```

The `Value` returns a shared memory object. By default, it returns a synchronized wrapper for the object, and the object itself can be obtained using the `value` attribute of `Value`, that is, `Value.value`.

The `typecode_or_type` argumentdefines the return type of the object, for example, if we are expecting an integer, then use `i`, and use `d` (double) for float.

Take a look at the following table for the details:

Type code	C Type	Minimum size in bytes
'b'	Signed integer	1
'B'	Unsigned integer	1
'u'	Unicode character	2

'h'	Signed integer	2
'I'	Unsigned integer	2
'l'	Signed integer	4
'L'	Unsigned integer	4
'q'	Signed integer	8
'Q'	Unsigned integer	8
'f'	Floating point	4
'd'	Floating point	8

Table 18.1

The `*args` argument specifies the size or the initial value.

If the lock is `True` (default), a new recursive lock object will be generated to synchronize the value. The purpose of the lock (RLock or Lock) is to synchronize the access of the object. If the lock is `False`, then access to the returned object will not be automatically defended by a lock.

Let's understand this with the help of the following examples:

```
import multiprocessing
import time
def shared(v):
v.value = v.value+1
print ("Sum of square ", v.value)

if __name__ =="__main__":
val = multiprocessing.Value('i', 0)
list1 = []
for each in range(5):
p = multiprocessing.Process(target=shared, args=(val,))
p.start()
list1.append(p)

for p in list1:
p.join()
print("In the end",val.value)
```

Let's take a look at its output as shown in the following screenshot:

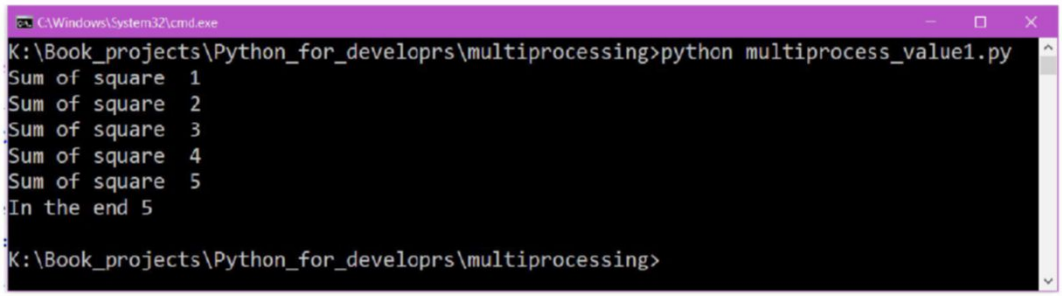

Figure 19.9

As you can see, each process is updating the value; the last process and the main process have the same value, which is 5.

Let's add a one second delay before printing the print ("Sum of square ", v.value) line. Add the following line:

```
print ("Sum of square ", v.value).
time.sleep(1)
```

Don't forget to import the time module. Let's look at the following screenshot to see the output of the code again:

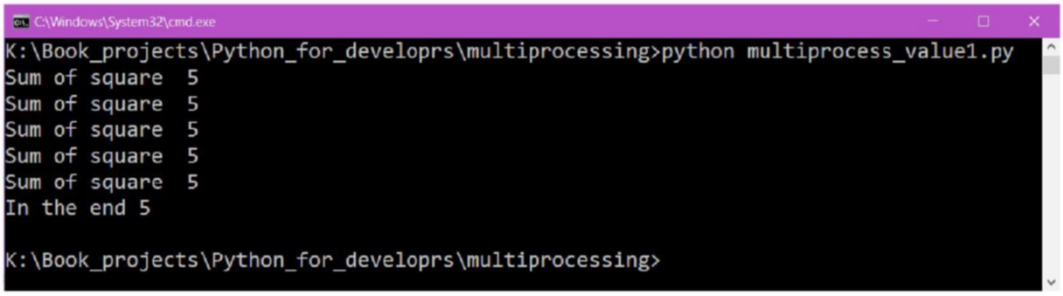

Figure 19.10

During the waiting time of 1 second, the last process updated the value, and all the process accessed the same value, which is 5, after the one second delay.

The main process and the child processes are sharing the same state.

Array

We use `Array` to store multiple numbers, as you can see in the following syntax:

```
multiprocessing.Array(typecode_or_type, size_or_initializer, *,
lock=True)
```

TheArray can take three arguments; let's discuss them one by one:

- `typecode_or_type:` This argument defines the type of elements of the Array. We use i for integer and d for double.
- `size_or_initializer:` We can give an integer ora sequence. If we provide an integer, then the integer specifies the length of the `Array`, and all the elements of the `Array` will be initialized with 0. If we give any sequence, then the `Array` will contain the element of the sequence.

The working of the lock will be the same as we saw in `Value`.

Let's take a look at the following example:

```
import multiprocessing
import time

def shared(a,i):
a[i] = 10+i
print (a[:], type(a))

if __name__ =="__main__":
arr = multiprocessing.Array('d', 5)
print ("In start", arr[:])
list1 = []
for each in range(5):
p = multiprocessing.Process(target=shared, args=(arr,each))
p.start()
list1.append(p)

for p in list1:
p.join()
print("In the end",arr[:])
```

The `arr = multiprocessing.Array('d', 5)` statement signifies that all the elements of the `Array` will contain double types, and the size of the `Array` will be 5. Let's discuss the output, which is shown in the following screenshot:

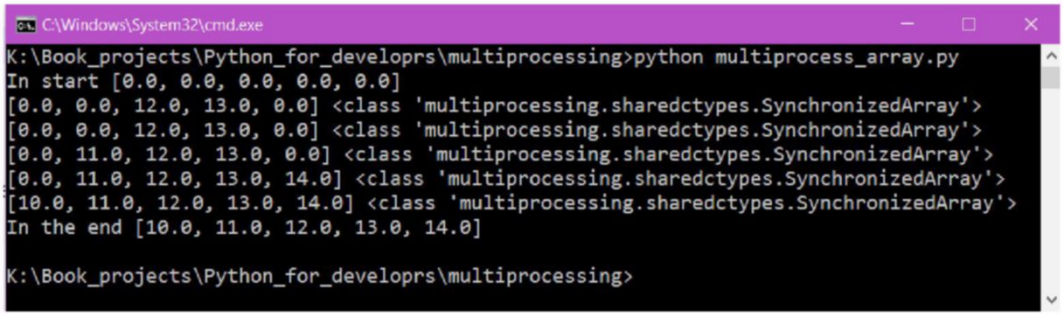

Figure 19.11

All the elements of the `Array` are initialized with 0.0; however, if we had used `i` instead of d, then it would be 0. Every process is filling its value one by one. In the end, the Array has been filled by all the processes. Let's perform an experiment. Replace the `arr = multiprocessing.Array('d', 5)` statement with `arr = multiprocessing.Array('d', range(5))`, and then run the program.

The following screenshot shows the result:

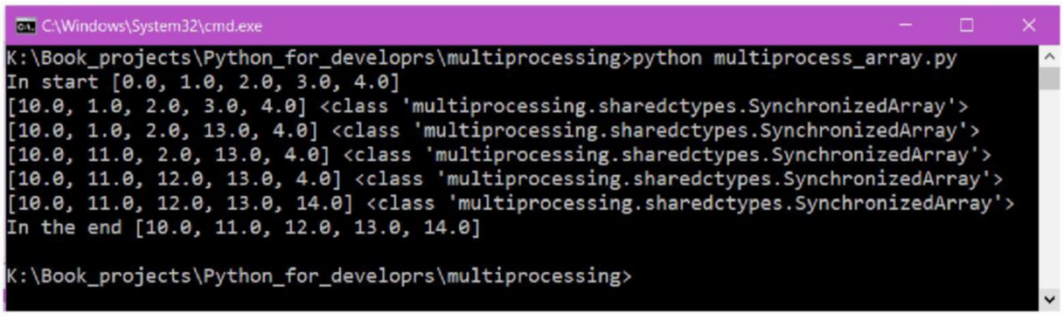

Figure 19.12

In the preceding screenshot, you can see that the `Array` is initialized with the values specified in the sequence.

The Manager class

Through the `Manager` class object, the list, dictionary, and queue can be shared. A `Manager` class' object controls a server process that holds Python objects and allows other processes to manipulate them using proxies.

Take a look at the following program; here, two processes are appending the values to the list:

```
from multiprocessing import Process, Manager,current_process
def worker(list1, num):
list1.append(num)
print ("list1 by ",current_process().name, list1 )

if __name__ == '__main__':
mgr = Manager()
list1 = mgr.list()
p1 = Process(name ='worker1', target=worker, args=(list1,1))
p2 = Process(name ='service', target=worker, args=(list1,2,))
p1.start()
p2.start()
p1.join()
p2.join()
print ("list is ", list1)
print ("All finished")
```

The Python list, `list1`, has been created by the `Manager` class object.

Now let's look at the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\multiprocessing>python process_list_manger.py
list1 by  worker1 [1]
list1 by  service [1, 2]
list is  [1, 2]
All finished

K:\Book projects\Python for developrs\multiprocessing>
```

Figure 19.13

Exchanging object through the communication channel

Multiprocessing facilitates two types of interprocess communication channels:

- **Queue:** Python multiprocessing uses the queue defined in the multiprocessing module. This isn't the normal queue; it resides in shared memory. This queue is similar to the FIFO queue (`queue.Queue`).
- **Pipe:** The `pipe()` function returns two objects to make connections, which is duplex in nature.

Let's explore the queue. The following program represents the producer and consumer problem:

```
from multiprocessing import Process, Queue
import time,random
Q1 = Queue()
def produce1():
while True:
num1 = random.randint(1,100)
Q1.put(num1)
time.sleep(1)
print ("size is ", Q1.qsize())

def consumer():
while True:
print (Q1.get())
time.sleep(1)

p1 = Process(target = produce1)
p1.start()

p2 = Process(target = consumer)
p2.start()
```

Let's look at the output for the preceding program here:

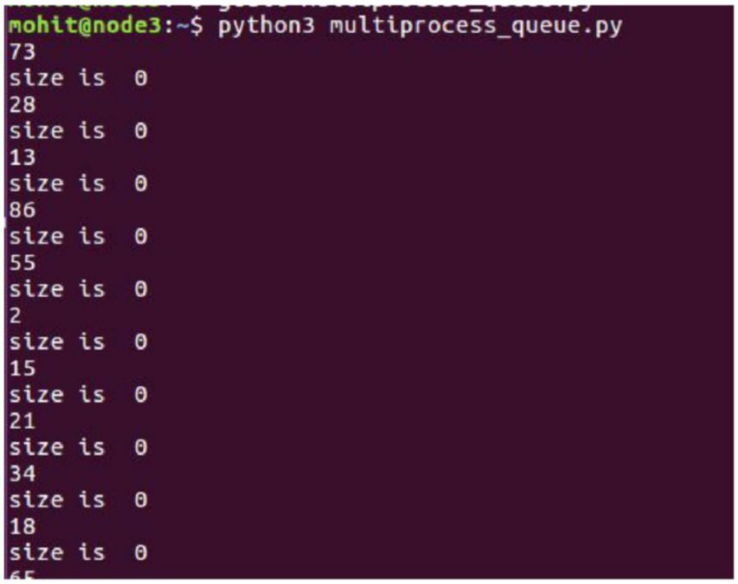

Figure 19.14

The above result shows the synchronization of two processes. Now let's see the communication with the help of pipe.

Pipe

A two-way communication channel can be created using pipe. The `pipe()` function returns two connection objects; each object can send and receive messages by using `send()` and `recv()`, as shown here:

```
from multiprocessing import Process, Pipe

def shared(conn):
    conn.send(["mohit", 99])
    conn.close()


if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=shared, args=(child_conn,))
    p.start()
    p.join()
    print(parent_conn.recv())
```

In the preceding program, the child process is sending the message and the main process is receiving the message. The following screenshot shows the output of the program:

```
K:\Book_projects\Python_for_developrs\multiprocessing>python multi_process_shared_pipe.py
['mohit', 99]

K:\Book_projects\Python_for_developrs\multiprocessing>
```

Figure 19.15

Subprocess

The subprocess module allows you to create new processes, connect to their input/output/error pipes, and obtain their return codes. The subprocess module is designed to replace several older modules and functions such as `os.system,os.spawn*,os.popen*, andpopen2.*.`

Difference between subprocess and multi-processing

The aim of multi-processing is to run the function or task of the Python program. Multi-processing allows you to divide the tasks and execute them in parallel; whereas, subprocess is designed to run the external command.

The call() function

Whenever you run the command in the terminal or the command prompt, it takes a standard input and displays its output. The `call()` method is used to run the external commands; this is the replacement of `os.system()`.

Take a look at the following syntax:

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)
```

Now we will discuss its options with the help of examples.

Let's see the following example on Windows:

```
>>> import subprocess
>>> subprocess.call("ping -n 1 127.0.0.1")

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
0
>>>
```

Figure 19.16

If the return code is 0, it means the command was executed successfully.

You can also give the command in the list, as follows:

```
>>> subprocess.call(['ping', '-n', '1', '127.0.0.1'])

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
0
>>>
```

Figure 19.17

In Linux, multiple commands in the form of string may give an error, as shown in the following screenshot:

```
>>> import subprocess
>>> subprocess.call("ls -l")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.7/subprocess.py", line 323, in call
    with Popen(*popenargs, **kwargs) as p:
  File "/usr/local/lib/python3.7/subprocess.py", line 775, in __init__
    restore_signals, start_new_session)
  File "/usr/local/lib/python3.7/subprocess.py", line 1522, in _execute_child
    raise child_exception_type(errno_num, err_msg, err_filename)
FileNotFoundError: [Errno 2] No such file or directory: 'ls -l': 'ls -l'
>>>
>>>
```

Figure 19.18

It is better to give the command in the list, as follows:

```
>>> subprocess.call(["ls","-l"])
total 81000
-rw-rw-r--  1 mohit mohit     1329 May 17 11:17 config-genesis.batch
-rw-------  1 mohit mohit  4710400 Jun 22  2018 core
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Desktop
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Documents
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Downloads
-rw-r--r--  1 mohit mohit     8980 May 17  2018 examples.desktop
-rw-r--r--  1 root  root         0 May 19 17:12 game.py
-rw-rw-r--  1 mohit mohit        0 May  3 15:12 hello.log
-rw-rw-r--  1 mohit mohit       67 Apr  5  2019 key1
-rw-rw-r--  1 mohit mohit        0 May  3 15:13 log
-rw-rw-r--  1 mohit mohit     1522 Jun 20  2018 main.ui
-rw-rw-r--  1 mohit mohit     3111 Sep  8 22:19 mohit.csv
-rw-rw-r--  1 mohit mohit     1573 Jun 22  2018 mohit_exp.ui
-rw-r--r--  1 root  root      2605 Jun 22  2018 mohit.py
-rw-rw-r--  1 mohit mohit      356 Apr 14 22:28 multiprocess_queue.py
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Music
drwxrwxrwx 11 mohit mohit     4096 Sep 25 10:57 my_code
drwxrwxrwx  3 mohit mohit     4096 Mar 27  2019 my_code1
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Pictures
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Public
-rw-rw-r--  1 mohit mohit      134 Jun 20  2018 py1.py
-rw-r--r--  1 root  root      2605 Jun 22  2018 pyde.py
-rwxrw-rw-  1 mohit mohit 21748006 Sep  9 16:57 stacer_1.1.0_amd64.deb
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Templates
-rwxrwxrwx  1 root  root         0 Jul  3 07:30 text1.txt
drwxr-xr-x  2 mohit mohit     4096 May 17  2018 Videos
-rw-rw-r--  1 mohit mohit      135 Sep 17 15:53 vlun1.c
drwxrwxr-x  3 mohit mohit     4096 May 17  2018 vm
-rwxrwxrwx  1 mohit mohit 56375699 Sep 14  2017 VMwareTools-10.1.15-6627299.tar.gz
drwxr-xr-x  9 mohit mohit     4096 Sep 14  2017 vmware-tools-distrib
-rw-rw-r--  1 mohit mohit      135 Sep 17 15:54 vuln.c
0
>>>
```

Figure 19.19

Note: Do not use stdout=PIPE or stderr=PIPE with this method; the child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Let's see another case of Windows, as shown in the following screenshot:

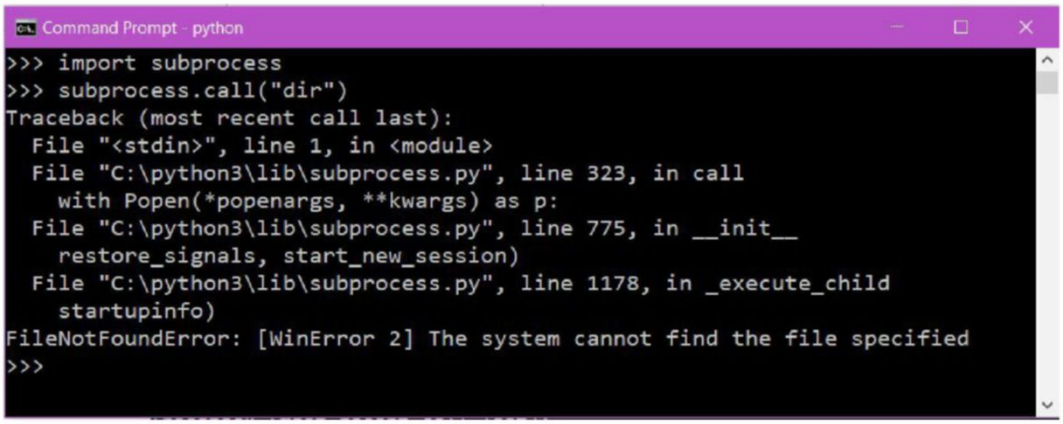

Figure 19.20

The `dir` command in Windows displays the information of the file and folder present in the current directory. In the preceding figure, the interpreter is throwing an error. Because `dir` is not an external command, it is built-in to shell.

In order to run a built-in shell or command prompt command, we need to `shell=True` argument, as shown in the following screenshot:

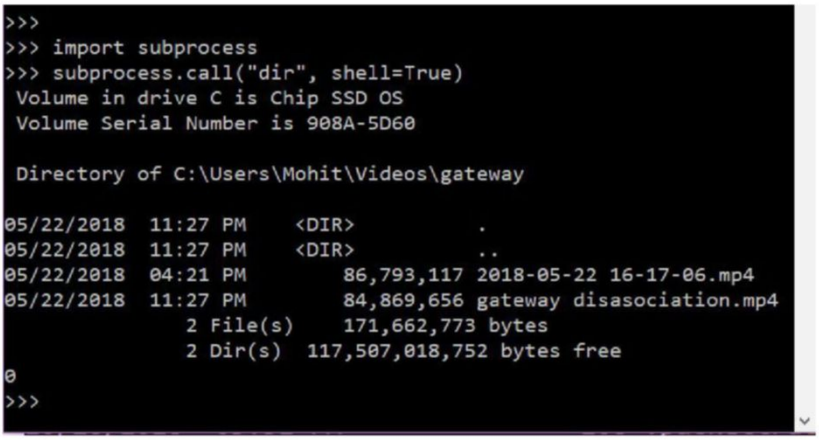

Figure 19.21

If you want to store the output in a variable, then you can use the `check_output()` method, as shown in the following example:

```
>>>
>>> import subprocess as sp
>>> p=sp.check_output(['ls', '-ltr'])
>>> print (p)
total 59748
drwxr-xr-x 9 mohit mohit     4096 Sep 14  2017 vmware-tools-distrib
-rwxrwxrwx 1 mohit mohit 56375699 Sep 14  2017 VMwareTools-10.1.15-6627299.tar.gz
-rw-r--r-- 1 mohit mohit     8980 May 17  2018 examples.desktop
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Desktop
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Videos
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Templates
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Public
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Pictures
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Music
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Downloads
drwxr-xr-x 2 mohit mohit     4096 May 17  2018 Documents
drwxrwxr-x 3 mohit mohit     4096 May 17  2018 vm
-rw-rw-r-- 1 mohit mohit      134 Jun 20  2018 py1.py
-rw-rw-r-- 1 mohit mohit     1522 Jun 20  2018 main.ui
-rw------- 1 mohit mohit  4710400 Jun 22  2018 core
-rw-rw-r-- 1 mohit mohit     1573 Jun 22  2018 mohit_exp.ui
-rw-r--r-- 1 root  root      2605 Jun 22  2018 pyde.py
-rw-r--r-- 1 root  root      2605 Jun 22  2018 mohit.py
drwxrwxrwx 3 mohit mohit     4096 Mar 27 14:44 my_code1
-rw-rw-r-- 1 mohit mohit       67 Apr  5 14:37 key1
-rw-rw-r-- 1 mohit mohit     1330 Apr 12 11:41 config-genesis.batch
-rw-rw-r-- 1 mohit mohit      356 Apr 14 22:28 multiprocess_queue.py
drwxrwxrwx 5 mohit mohit     4096 May  3 15:09 my_code
-rw-rw-r-- 1 mohit mohit        0 May  3 15:12 hello.log
-rw-rw-r-- 1 mohit mohit        0 May  3 15:13 log

>>>
```

Figure 19.22

In the preceding figure, you can see that the output is stored in the variable. The next method is `popen()`.

Popen()

The `subprocess.popen` is the replacement of `os.popen`. Let's discuss the `popen()` method with the help of the following example:

```
>>>
>>> proc = sp.Popen(['ping', '-c', '1', '127.0.0.1'])
>>> PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.016 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.016/0.016/0.016/0.000 ms

>>> proc.communicate()
(None, None)
>>>
```

Figure 19.23

In the preceding example, `Popen` is working as `os.system()`. Now take a look at the following example:

```
>>> proc = sp.Popen(['ping', '-c', '1', '127.0.0.1'], stdout= sp.PIPE)
>>> print (proc.communicate()[0])
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.014 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.014/0.014/0.014/0.000 ms

>>>
```

Figure 19.24

The `stdout=subprocess.PIPE` argument is a file object that provides output from the child process. Otherwise, it is `None`.

The `Popen.communicate(input=None)` argument reads the data from `stdout` and `stderr` until the end-of-file is reached. Wait for the process to terminate; the `communicate()` attribute returns a tuple `(stdoutdata, stderrdata)`.

Conclusion

In this chapter, you learned about multiprocessing, and with its help, we can create multiple processes. Where multi-threading does not work, we use multi-processing. The multi-processes create their memory block and don't share their stack; therefore, communication between the processes is not an easy task. However, shared memory and communication channels make it possible. In shared memory, the `Value, Array,` and `Manager` classes are included; and in the communication channel,the queue and pipe classes are included. At the end of the chapter, you also saw the subprocesses, which help in execution of OS commands. A lot of subprocess methods can be used as the replacement of the OS functions. In the next chapter, you will learn about `configparser, argparse, logger`, and `debugger`; these modules will be very helpful from the industry point of view.

Questions

1. What is the difference between multithreading and multi-processing?
2. Which takes less time to create: a thread or a process?
3. Can we use a queue with multi-processing?
4. When to use multithreading and multi-processing?

Chapter 20
Useful Modules

You may have seen the Linux directory structure—the /bin directory contains a lot of binary executable files and the `/etc` directory contains a lot of configuration files. The binary files cannot be updated, but they take some input from its configuration file placed in the `/etc` directory. A user can change the configuration file, which contains some parameter that are used by the binary file. In Python, we can make this type of functionality with the help of `configparser` so that we can avoid hardcoding of the Python program. You may have seen some popular applications like apache, where they take their parameters like IP address and port number from a configuration file (`httpd.conf or apache.conf`) and write their log event in a specified log file. By viewing the log file, we can file the events, errors, and so on. In Python, you will learn about the logger to create log files. Some commands like ping, which can be used with options like ping `-n 1` is used to send one ICMP packet, similarly the `ls` command, which can be used with a lot of options such as `ls -l`, `ls -l -t -r`. In Python, we can achieve the same thing by using a module called `argparse`, which you will learn about in this chapter.

Structure

- Configparser
- Loggers
- Argparse
- Debugging

Objective

In this chapter, you will learn the modules, which will be very helpful when starting a project. You will learn how to avoid hardcoding, how to create efficient log files to track events, and how to generate command-line arguments. By the end of the chapter, you will learn how to debug a code.

Configparser

In this section, we will see the `configparser` module. The `configparser` module is used to prevent the program from being hardcoded. In professional code, we don't hardcode any variable. If you are acquainted with Linux, in the `etc`' directory, there are many conf files that are used to configure the executable files. Let's consider a scenario where the production environment has five servers, and every server has the same set of codes. The developer wants to change some parameters in the code. It will be very tedious to change the parameter in the code of all the servers. To tackle this problem, we will write all the parameters required by the codes to a conf file. In the future, if some setting needs changes, then we will deploy a new conf file to all servers with the new modified setting, as shown in *Figure 20.1:*

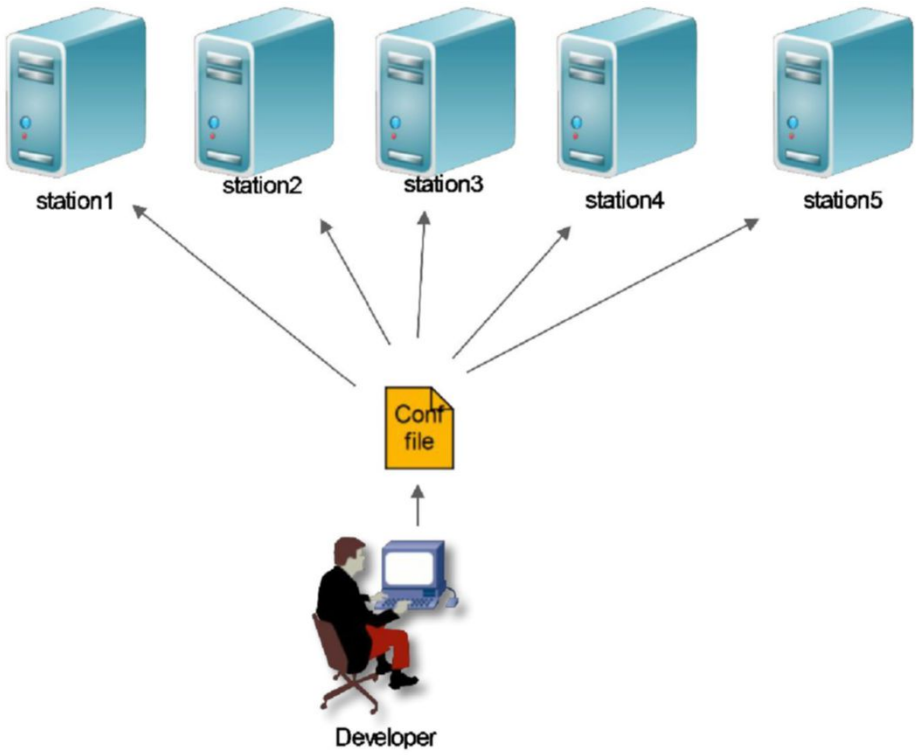

Figure 20.1

Let's see how we can use `configparser`.

Let's consider we want to use the DNS and HTTP server's IP address and port numbers in our program. Now, we will look at how we can create the configuration file and avoid the hardcoding.

First, create a config file named `config.ini` (you can use any name and extension). Take a look at the following syntaxes of config file:

```
[DNS]
Server_ip= 127.0.0.1
Port = 53


[HTTP]
Server_ip= 127.0.0.7
Port = 80
```

In the preceding config file, we have fixed the different server IPs and ports. By using `[]`, we have created the sections like DNS and HTTP in the config file. Each section contains a key with the value type structure, as you can see in the following code:

```
import configparser
config = configparser.ConfigParser()
config.read("config1.ini")
http_ip = config.get("HTTP", "IP")
print (http_ip)

http_port  = config.get("HTTP", "Port")
print (http_port)

dns_ip = config.get("DNS", "IP")
print (dns_ip)

dns_port  = config.get("DNS", "Port")
print (dns_port)
```

By using the `config = ConfigParser.ConfigParser()` syntax, the config object gets created. In the `config.read("config.ini")` syntax, the read method is used to read the config file. The `http_ip = config.get("HTTP", "IP")` syntax returns the value of `IP of HTTP` server.

Let's look at the following screenshot for the output:

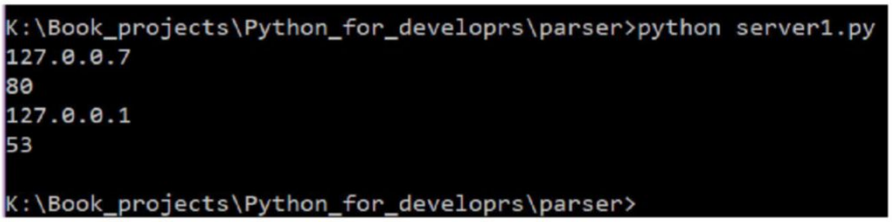

Figure 20.2

Now that we have an idea of how to parse the config file, let's consider we don't know any section name, and any key and value (even though we can see the config file, for one moment, let's consider we don't know anything).

We will parse the file without knowing any section or keys and values.

Let's look at the full code as shown here:

```
import configparser
config = configparser.ConfigParser()
config.read("config.ini")

for section in config.sections():
  print ("For ", section)
  for k,v in  (config.items(section)):
    print (k , ": ", v )
```

The first three lines are the same. The config.section() command returns a list of all the sections:

```
>>> config.sections()
['DNS', 'HTTP']
>>>
```

If we take one section from the list, then config.items() returns the list of tuples with the key and values:

```
>>> config.items('DNS')
[('ip', '127.0.0.1'), ('port', '53')]
>>>
```

Now, let's see the preceding code's output in the screenshot shown here:

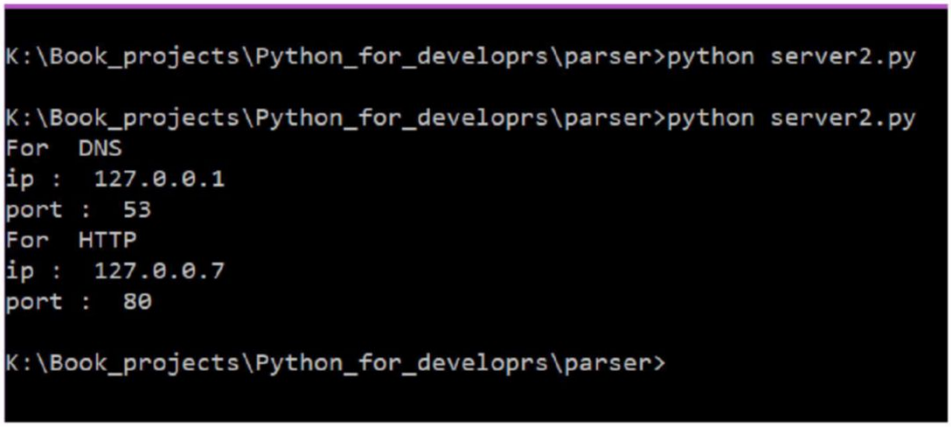

Figure 20.3

Now that you've got an idea of `configparser`, try to avoid the hard coding and use the configuration file.

Loggers

The logging module provides a way of tracking events that occur when a specific code is operating. Logging offers a collection of convenience methods that act as a level for logging. These levels are `debug()`, `info()`, `warning()`, `error()` and `critical()`. Each level has its priority level; `debug()` has a low priority level and `critical()` has a high priority level.

Although a program can write the output on a text file, opening and closing file operations is very time consuming. In the case of multithreading, it is a very complicated task to take the output on the file. Here, we will discuss how you can make your logger.

Let's create a Simple Logger with the help of `basicConfig`, as shown here:

```
import logging
logging.basicConfig(filename="Mylive.log",filemode="w", level=logging.DEBUG)
logging.debug("A debug message")
logging.info("A Info message")
logging.warning("A warning Here")
logging.error("Ohh Error!")
logging.critical("It is Critical, do something !!!!")
```

Let's look at the output of the preceding code:

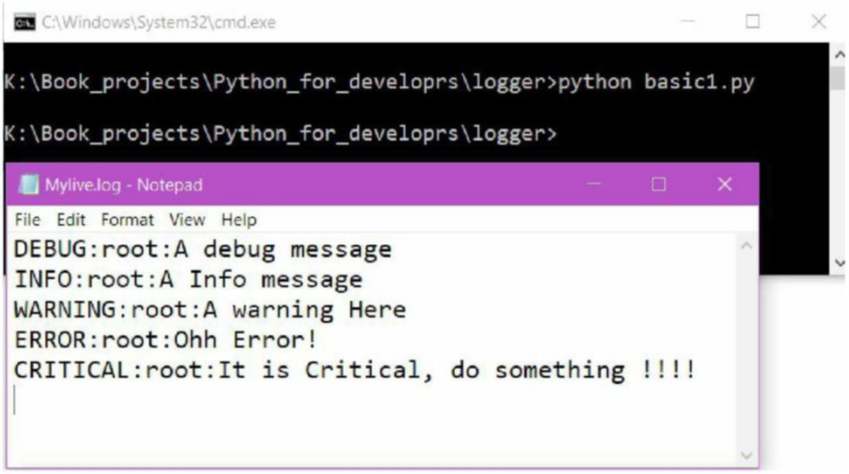

Figure 20.4

In the preceding screenshot, you can see messages of all levels because the second line of the program specified `level=logging.DEBUG`, and we know there are five levels of logging (in ascending order): **`DEBUG`**, **`INFO`**, **`WARNING`**, **`ERROR`**, and **`CRITICAL`**, where Debug is the lowest level. However, if we set `level=logging.ERROR`, then we will get all the messages higher than the level of Error. Let's now see the output of the code after changing the level.

Let's run the program with the `ERROR` level, as shown here:

```
import logging
logging.basicConfig(filename="Mylive.log",filemode="w", level=logging.ERROR)
logging.debug("A debug message")
logging.info("A Info message")
logging.warning("A warning Here")
logging.error("Ohh Error!")
logging.critical("It is Critical, do something !!!!")
```

The following screenshot shows the code's output:

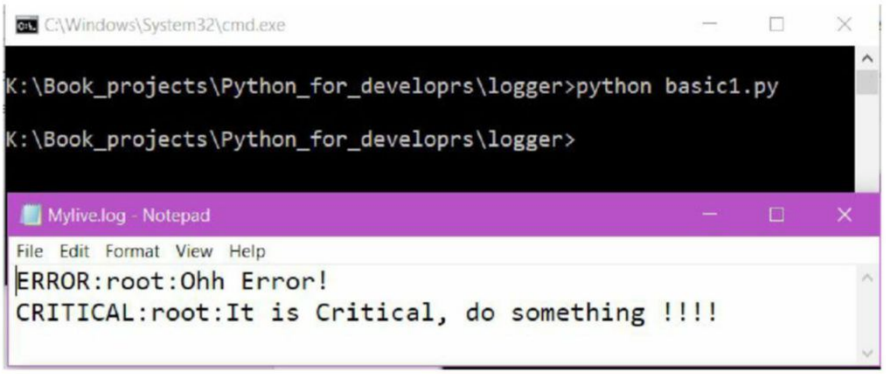

Figure 20.5

You can see the only error and critical message is due to its ascending order. The root part signifies that this logging message is coming from the root logger or the main logger. To make it more descriptive, don't use `basicConfig`. We can make our own template or format to log the events; the following program shows us how:

```
import logging

import time

logger = logging.getLogger("MOHIT")

logger.setLevel(logging.INFO)

fh = logging.FileHandler("mohit_live.log")

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')

fh.setFormatter(formatter)

logger.addHandler(fh)
```

Here, we created a logger instance named "`MOHIT`". The level of the logger is set at line number 4. The line number 5 specifies the log file name. The line number six, `logging.Formatter()`, specifies the format of log file, which you will understand after viewing the events on the log file.

The logger has been created. Now save the program as `logger1.py.`

Let's make another program, `prime1.py`, which will import `logger1.py`. The following program will find the prime number:

```
import math

import logger1 as lg
```

```
try:
  num1 = int(input("Enter the number "))
  loop_number =int(math.sqrt(num1))+1  # 1/2
  flag = 0
  lg.logger.info("Value is %d"%(num1))
except Exception as e :
  lg.logger.error("Error in first and Error is %s"%(e))

try:
  for each in range(2,loop_number):
    #print (each)
    c=num1%each
    lg.logger.info("Result is %d"%(c))
    if c ==0:
      flag = 1
      break
except Exception as e :
  lg.logger.error("Error in second and Error is %s"%(e))

try:
  if flag ==0:
    print ("Number is prime ")
  else :
    print ("Number is not prime")
except Exception as e :
  lg.logger.error("Error in flag and Error is %s"%(e))
```

We just used two levels that are **`INFO`** and **`ERROR`**. Let's generate its output and check the `INFO` and `ERROR` events. See the following screenshot for the output:

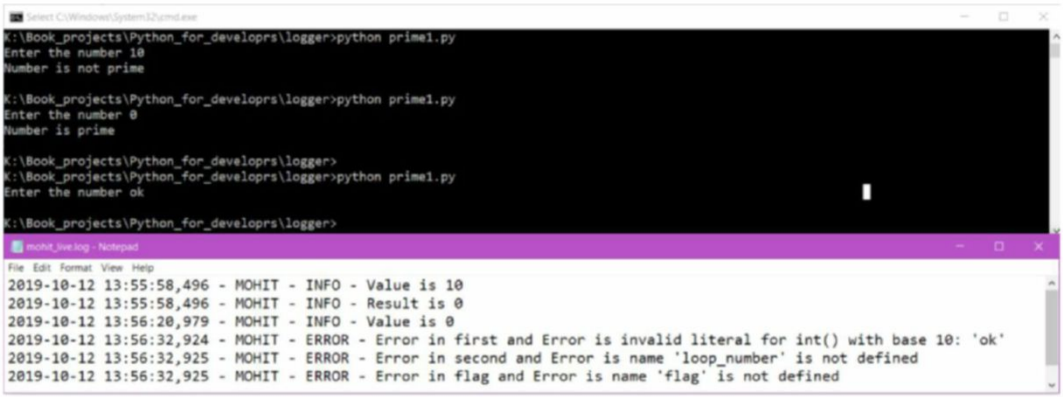

Figure 20.6

We set the format as `logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')` in the `logger1.py` program.

Therefore, we got the output in the same format. First, we got the time of event, then the logger name, level name, and the messages crafted in `prime1.py`. When there is no error, an Info message is being printed and if an error occurs, then an error message is getting printed. The following table describes the formats that you can use:

%(name)s	Logger's name
%(levelno)s	Level number of DEBUG (10), INFO (20), WARNING (30), ERROR (40), and CRITICAL (50).
%(levelname)s	Level name of "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"
%(pathname)s	The path from where the source file has been executed.
%(filename)s	Name of the source file
%(module)s	Module (name portion of the filename)
%(lineno)d	The line number of Python source file from where the logging call was published
%(funcName)s	Function name from logging call was published (if the function is defined)
%(created)f	Time in Epoch Unix time when the log event was created
%(asctime)s	Human-readable time, when the logging call was recorded
%(msecs)d	The millisecond part of the time of the creation
%(relativeCreated)d	Time in milliseconds when the logging call was recorded, relative to the time the logging module was loaded (typically at application startup time)

`%(thread)d`	The ID of Thread (if available)
`%(threadName)s`	Name of Thread (if available)
`%(process)d`	Id of Process (if available)
`%(message)s`	The logging message crafted by the user

Table 20.1

Argparse

In this section, we will understand how to use the command-line argument. The `argparse` module allows writing user-friendly command-line interfaces. It also automatically generates help and usage messages, and issues errors when users give the program invalid arguments. There are different types of arguments associated with `argparse`.

Let's start with the basic code to understand it:

```
import argparse

parser = argparse.ArgumentParser()

parser.parse_args()
```

The following screenshot displays its output:

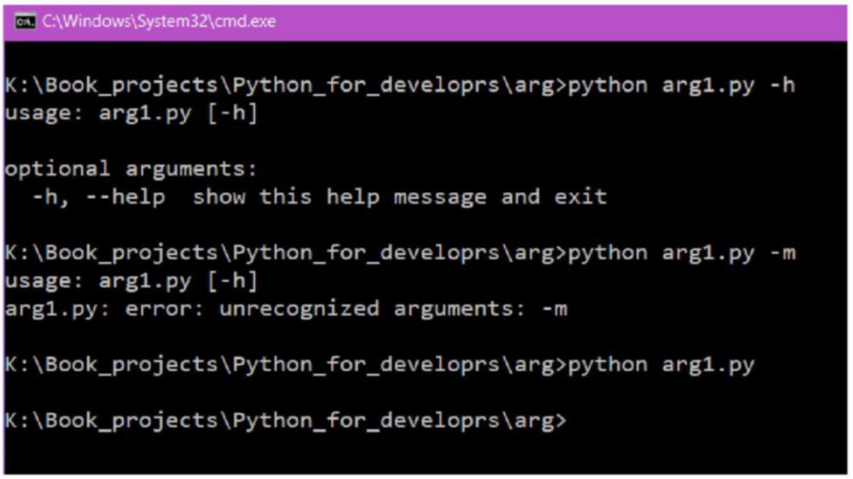

Figure 20.7

From the preceding output, we can conclude three things:

- With the `-h` option, the help message is getting printed
- With an unknown option, the argument program is giving an error
- Without any option, nothing is printed

Let's study the different types of arguments offered by the `argparse` module.

The positional argument

When you provide an argument without any option, it is the positional argument.

Let's look at the following code for better understanding:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("mohit")
arg =parser.parse_args()
print (arg.mohit)
```

In the preceding program, a couple of things are new – the add `argument()` method has been added to accept the command-line argument. In this case, "`mohit`" is used.

Positional arguments with Help message and Type

Let's add the help message and the type of argument, which is shown as follows:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("Square", help="Find Square ", type = int)
arg =parser.parse_args()
if arg.Square:
  print (arg.Square**2)
```

The following screenshot shows its output:

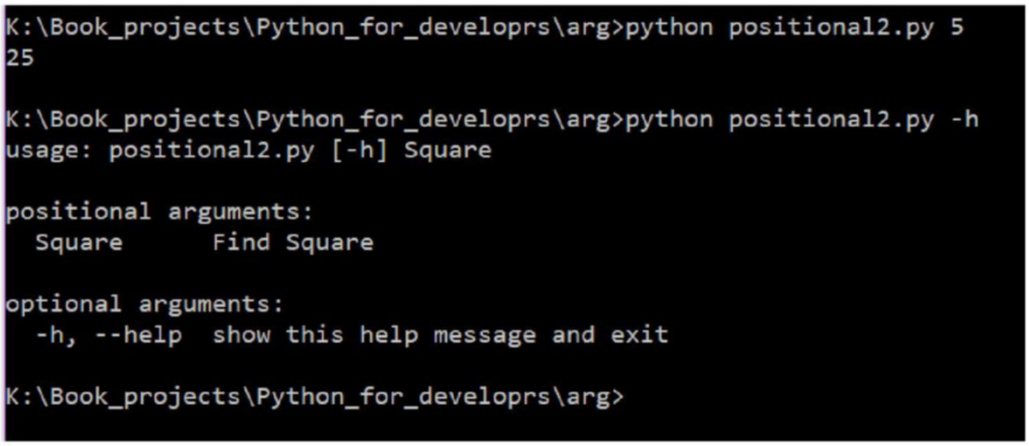

Figure 20.8

The Argparse optional argument

You have seen the positional argument; the following syntax will show you how to add custom arguments:

```
ArgumentParser.add_argument(flags or name, action, nargs, const, default,
type, choices, required, help, metavar, dest)
```

Here's a description of the above-mentioned options:

- `flags or name:` Either a name or a list of option strings, for example, foo or -f, --foo
- `action:` If this argument were made at the command line, an action such as `store_const` or `store_true` would be taken
- `nargs:` This is the number of command-line arguments that needs to be absorbed with name or flags
- `const:` This argument requires a constant value when an action like `store_const` is used
- `default:` If the argument is missing from the command line, then the default value will be taken
- `type:` This is the type to which the command-line argument's value should be converted, such as int or float
- `choices:` This is a container that is like a list of the permissible values for the argument
- `required:` If `required=True`, then the command-line argument must be present
- `help:` This is a short explanation about the purpose of the argument
- `metavar:` This is a name for the argument in display messages
- `dest:` The dest option of the `add_argument()` gives a name to the argument and if not given, it is inferred from the option

When `parse_args()` is called, optional arguments or the name of the flag will be identified by the – prefix.

Let's discuss name or flag with an action, as shown here:

```
import argparse
list1 = [1,2,1,9,4,5]
parser = argparse.ArgumentParser()
parser.add_argument("-R", help="Print in reverse order", action='store_true')
parser.add_argument("-g", help="A constant number ", action='store_const',const=9.8)
arg =parser.parse_args()
if arg.R:
```

```
  print (list1[::-1])
if arg.g:
  print (arg.g  )
```

In the preceding program, `store_true` stores the Boolean value, `True`. Similarly, `store_false` can be used to store the Boolean value, `False`. The `store_const` command is used to store the constant value. The `store_true` command is the particular case of `store_const`. The following screenshot shows the output:

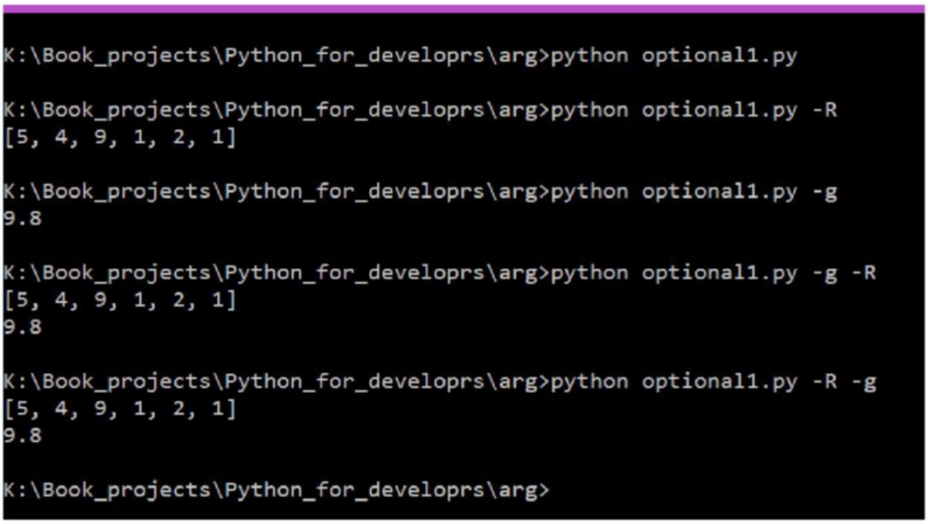

Figure 20.9

In the preceding figure, you can see that the `-R` and `-g` options are the flags. The presence of the `-g` and `-R` flag has made their respective `if block` executed.

If you want to give a value with options, then we use `nargs`.

nargs

Let's see, with the help of the following code, how to supply multiple values with one argument:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-Y", nargs=2, help= "Hours")
parser.add_argument("-N", nargs='?', help= "One argument")
parser.add_argument("-R", nargs='*', help= "Second argument")
parser.add_argument("-S", nargs='+', help= "Third argument")
arg =parser.parse_args()
if arg.Y:
```

```
  print (arg.Y)
if arg.N:
  print (arg.N)
if arg.R:
  print (arg.R)
if arg.S:
  print (arg.S)
```

See the following point and output to understand `nargs`:

- `nargs =2` means you will have to give two values with the `-Y` option.
- `nargs ='?'` means only one argument will be consumed from the command line if given.
- `nargs ='*'` means that with "*", Option R can accept 0 to many values.
- `nargs ='+'` means that with "+", Option S can accept 1 to many values, or at least one value must be supplied.

See the output in the following screenshot for more clarification:

```
K:\Book_projects\Python_for_developrs\arg>python optional2.py

K:\Book_projects\Python_for_developrs\arg>python optional2.py -Y
usage: optional2.py [-h] [-Y Y Y] [-N [N]] [-R [R [R ...]]] [-S S [S ...]]
optional2.py: error: argument -Y: expected 2 arguments

K:\Book_projects\Python_for_developrs\arg>python optional2.py -Y 2 4
['2', '4']

K:\Book_projects\Python_for_developrs\arg>python optional2.py -N

K:\Book_projects\Python_for_developrs\arg>python optional2.py -N 12
12

K:\Book_projects\Python_for_developrs\arg>python optional2.py -N 12 45
usage: optional2.py [-h] [-Y Y Y] [-N [N]] [-R [R [R ...]]] [-S S [S ...]]
optional2.py: error: unrecognized arguments: 45

K:\Book_projects\Python_for_developrs\arg>python optional2.py -R

K:\Book_projects\Python_for_developrs\arg>python optional2.py -R 12
['12']

K:\Book_projects\Python_for_developrs\arg>python optional2.py -R 12 5
['12', '5']

K:\Book_projects\Python_for_developrs\arg>python optional2.py -R 12 5 0
['12', '5', '0']

K:\Book_projects\Python_for_developrs\arg>python optional2.py -S
usage: optional2.py [-h] [-Y Y Y] [-N [N]] [-R [R [R ...]]] [-S S [S ...]]
optional2.py: error: argument -S: expected at least one argument

K:\Book_projects\Python_for_developrs\arg>python optional2.py -S 89
['89']

K:\Book_projects\Python_for_developrs\arg>
```

Figure 20.10

In the preceding screenshot, you can supply the values with arguments.

Subparser

A subparser means a parser within a parser or a nested parser. Sometimes, we need options within an option; that's when we use sub-parser. Let's look at the following code and understand it (this piece code is the same as the old code):

```
import argparse
import sys
import os
parser = argparse.ArgumentParser()
```

In the following line of code, a sub-parser object has been created. The `dest` keyword is normally supplied as the first argument to `add_argument()`. So, for the sub-parser, the first argument is command:

```
subparsers = parser.add_subparsers(help='Making commands',dest='command')
```

In the following piece of code, a new sub-parser named create has been created. The `-d` and `-f` options have been added:

```
create_parser = subparsers.add_parser('create', help='Create files or dir')
create_parser.add_argument('-d', nargs =1, help='New directory to create')
create_parser.add_argument('-f', nargs =1, help='create file')
```

Similarly, a delete sub-parser has been added:

```
delete_parser = subparsers.add_parser('delete', help='Remove a file or directory')
delete_parser.add_argument('-df', nargs =1, help='Delete directory or file')
```

The following line is very important as `sys.argv[1:]` contains all the arguments:

```
args = parser.parse_args(sys.argv[1:])
```

The following code is the logic part; if the sub-parser is created, then check the `-d` and `-f` options:

```
if  args.command== "create":
  if args.d:
    os.mkdir(args.d[0])
  elif args.f :
    f=open(args.f[0], "w")
    f.write("hello everyone")
    f.close()
```

The following lines show the logic of the delete command:

```
if  args.command== "delete":
  if args.df:
    os.remove(args.df[0])
```

Let's see the output in the following screenshot:

```
K:\Book_projects\Python_for_developrs\arg>python sub1.py

K:\Book_projects\Python_for_developrs\arg>python sub1.py -h
usage: sub1.py [-h] {create,delete} ...

positional arguments:
  {create,delete}  Making commands
    create         Create files or dir
    delete         Remove a file or directory

optional arguments:
  -h, --help       show this help message and exit

K:\Book_projects\Python_for_developrs\arg>python sub1.py create mohit
usage: sub1.py [-h] {create,delete} ...
sub1.py: error: unrecognized arguments: mohit

K:\Book_projects\Python_for_developrs\arg>python sub1.py create -d mohit

K:\Book_projects\Python_for_developrs\arg>python sub1.py create -f mohit.txt
```

Figure 20.11

From the above output, you can see that create and delete are the two sub-parsers. The create `-d` option creates a directory and create `-f` creates a file. Check the following figure for the output.

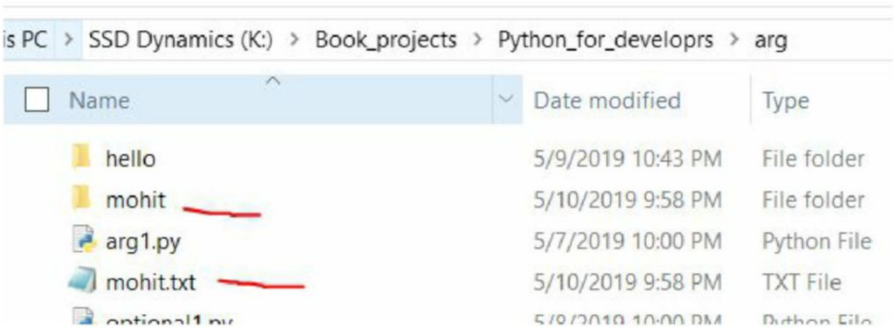

Figure 20.12

Similarly, you can use the delete option with `-df <file or folder>`. The `delete` option will then delete the file.

Debugging

In this section, we will see how to debug the Python code. You cannot be a good developer until you don't know how to debug the code. The Python exception will tell you which line is causing the problem, but to go deeper, which is, what is the actual reason for the error, we debug the code. We will take the help of the `pdb` debugger. Let's look at the following code as an example:

```
def calc():
  a = 10+20
  return a
def fun1():
  try:
    num1 = int(input("Enter the number "))
    c = calc()/num1
    a = c+40
    print (a)
  except Exception as e :
    print (e)
fun1()
```

Let's run the code to check the output (the following screenshot displays what it looks like):

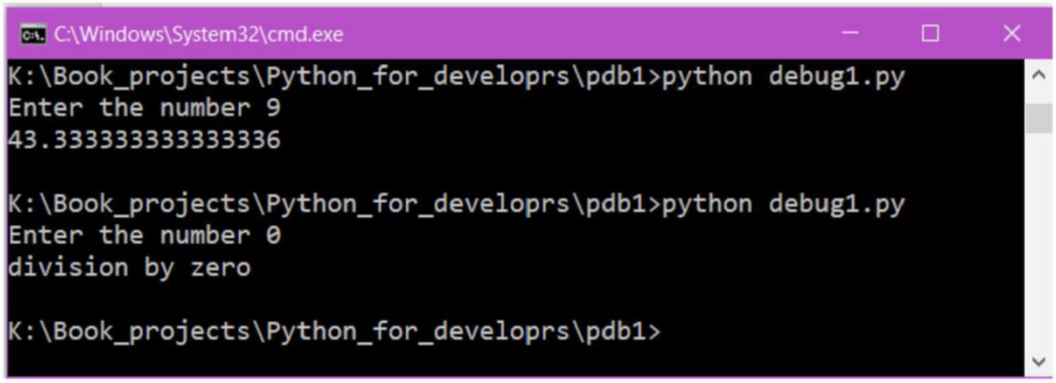

Figure 20.13

With the help of the traceback module, you will get the line number that is causing the problem. To check it line by line, and to get an output at each line, we will use the debugger. With the help of the following command, we can use `pdb`:

```
Python -m pdb <scrip_name>
```

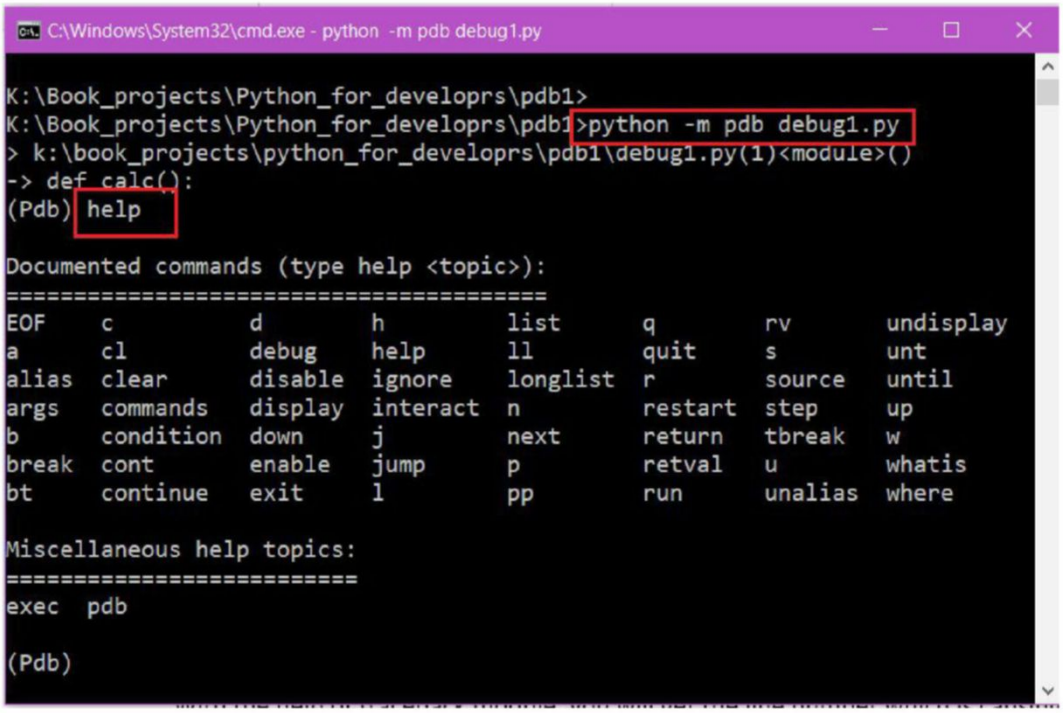

Figure 20.14

With the help option, all the offered options would be displayed. To check line by line, we will press `s` (step), and to stop after the execution of a function, we will use n (next). First, we will use the s option:

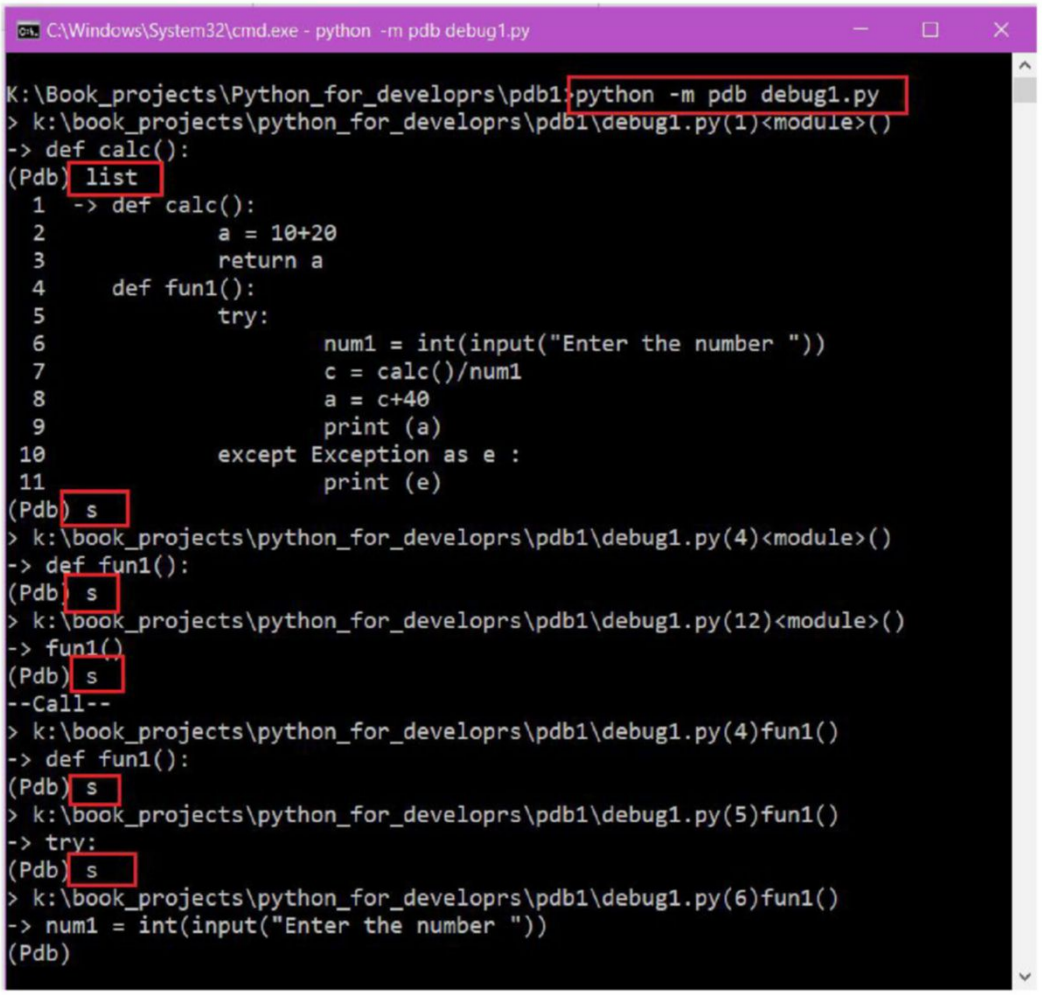

Figure 20.15

With the help of the list command, we can print the entire code; the a symbol on the left side of the line indicates the position of the debugger right now. When we press s, the debugger executes it line by line. See the following screenshot for the continuation:

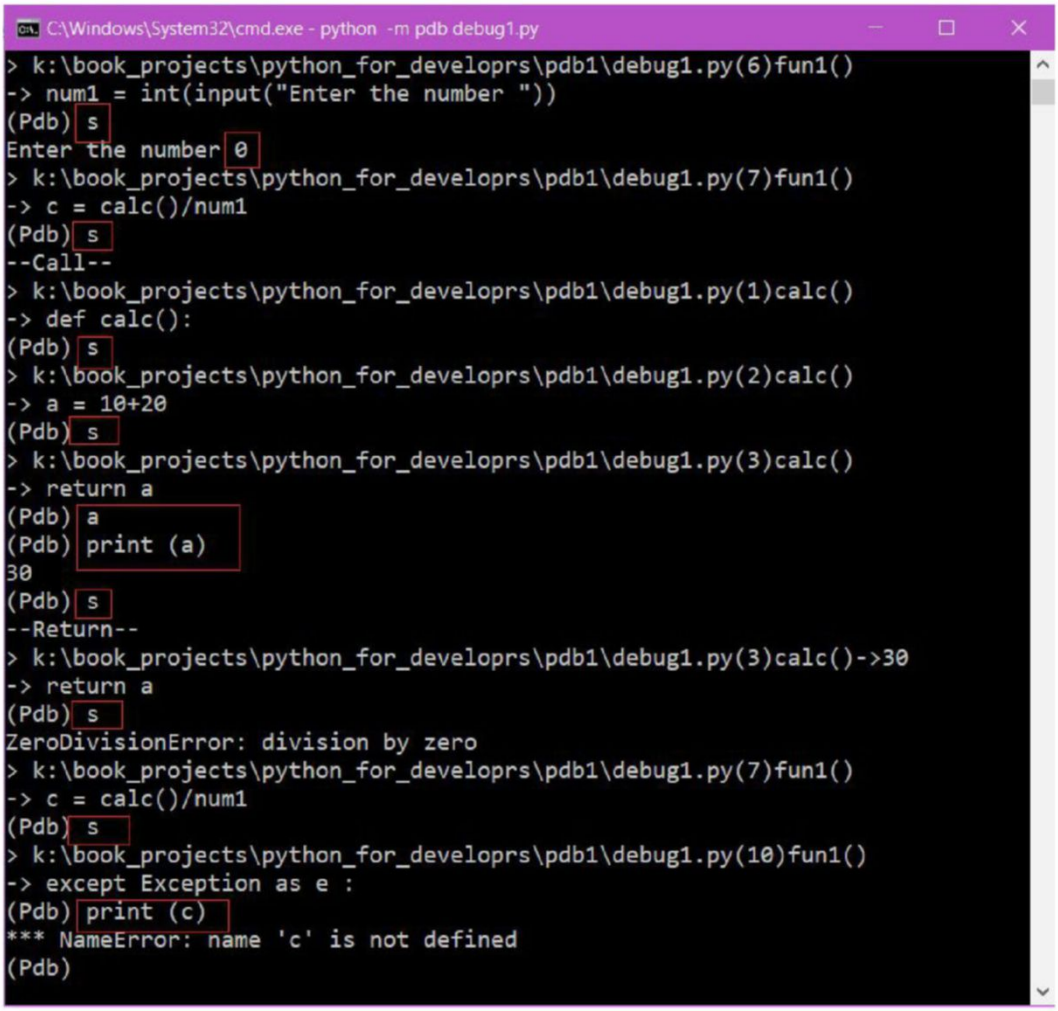

Figure 20.16

In the preceding screenshot, you can see that we can check any variable. We have printed a and c. We get an error that says c is not defined; it means the calculation to produce c has not been executed. This is the basic technique to find the error.

Let's use the n option now. When the n option is used, it stops after the execution of the entire function, as shown in the following screenshot:

Figure 20.17

Setting a breakpoint

Let's consider a program that contains 300 lines, and an exception module indicating the error at line 50. It's very time consuming to go line by line to reach line number

50. In that case, we use a breakpoint to jump directly to the specified line, as shown in the following example:

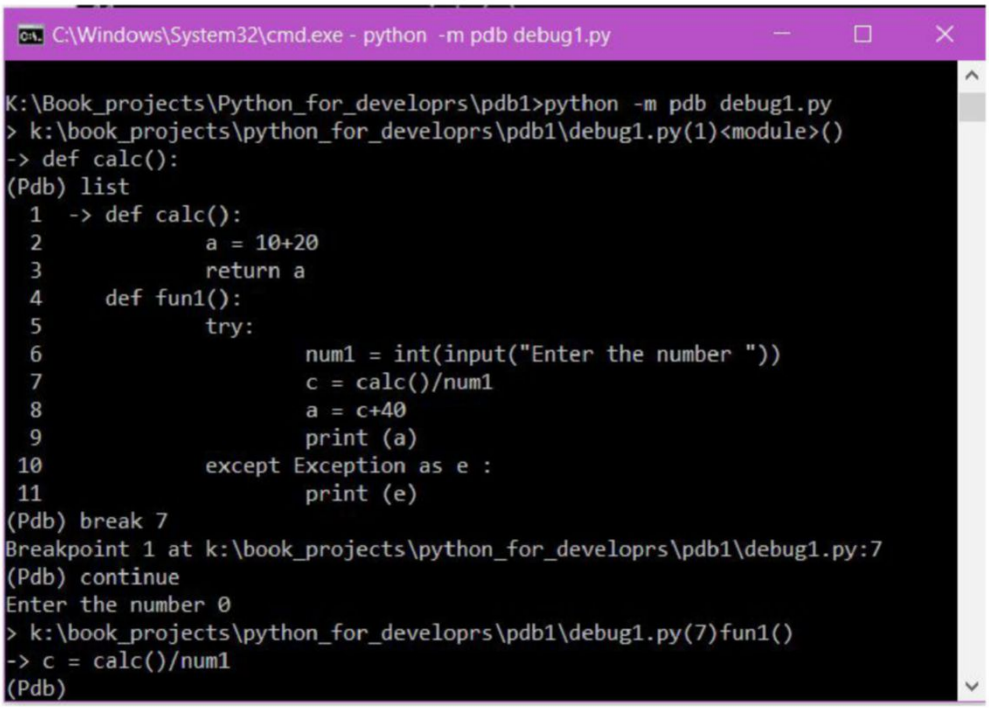

Figure 20.18

In the preceding screenshot, we set a breakpoint at line number 7. When the `continue` command is issued, the debugger directly jumps to the breakpoint. After that, you can check it line by line using the s option.

If you have the flexibility to edit the code, then you can write the following line to use `pdb`:

```
import pdb; pdb.set_trace()
```

Take a look at the following example:

```
def calc():
  a = 10+20
  return a
def fun1():
  try:
    num1 = int(input("Enter the number "))
    import pdb; pdb.set_trace()
    c = calc()/num1
```

```
    a = c+40
    print (a)
  except Exception as e :
    print (e)
fun1()
```

Running this code will give us the following output:

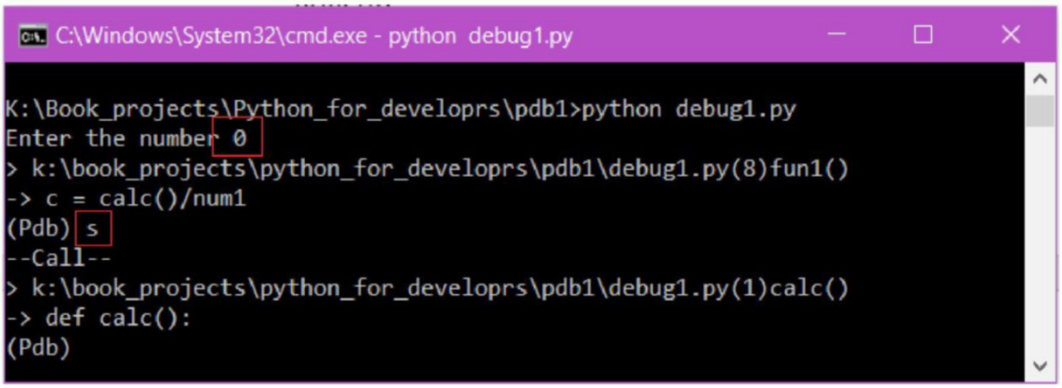

Figure 20.19

As you can see, the debugger directly jumped to the line where import pdb; pdb. set_trace() is written

Conclusion

In this chapter, you learned about the modules that are very useful for application development. With the help of the `configparse` module, we can avoid hardcoding. The configuration file allows a user to change the parameter of the executable code without touching it. The logging module enables a user to track the events, and it offers five levels of logging. The logging module also provides exciting information such as time, level name, error message, and line number, and so on. The `argparse` module allows writing command-line arguments. It also offers a positional argument, optional arguments, and sub-parser. The sub-parser refers to a parser within a parser. At the end, you learned about the Python debugger `pdb` – you cannot be a good developer until you don't know how to debug a code. The pdb facilitates you to check a code line by line, obtain the output at any line with the help of the `s` option.

Questions

1. Which logging level has the highest priority?
2. How to give the exact three values with the command-line argument?
3. What is the difference between "`s`" and "`n`" in `pdb`?