BASICS OF PYTHON PROGRAMMING:

A QUICK GUIDE FOR BEGINNERS

Krishna Kumar Mohbey
Malika Acharya

Bentham Books

Basics of Python Programming: A Quick Guide for Beginners

Authored by

Krishna Kumar Mohbey
Department of Computer Science
Central University of Rajasthan
Rajasthan, India

&

Malika Acharya
Department of Computer Science
Central University of Rajasthan
Rajasthan, India

Basics of Python Programming: A Quick Guide for Beginners

Authors: Krishna Kumar Mohbey and Malika Acharya

ISBN (Online): 978-981-5179-63-7

ISBN (Print): 978-981-5179-64-4

ISBN (Paperback): 978-981-5179-65-1

© 2023, Bentham Books imprint.

Published by Bentham Science Publishers Pte. Ltd. Singapore. All Rights Reserved.

First published in 2023.

BENTHAM SCIENCE PUBLISHERS LTD.

End User License Agreement (for non-institutional, personal use)

This is an agreement between you and Bentham Science Publishers Ltd. Please read this License Agreement carefully before using the ebook/echapter/ejournal (**"Work"**). Your use of the Work constitutes your agreement to the terms and conditions set forth in this License Agreement. If you do not agree to these terms and conditions then you should not use the Work.

Bentham Science Publishers agrees to grant you a non-exclusive, non-transferable limited license to use the Work subject to and in accordance with the following terms and conditions. This License Agreement is for non-library, personal use only. For a library / institutional / multi user license in respect of the Work, please contact: permission@benthamscience.net.

Usage Rules:

1. All rights reserved: The Work is the subject of copyright and Bentham Science Publishers either owns the Work (and the copyright in it) or is licensed to distribute the Work. You shall not copy, reproduce, modify, remove, delete, augment, add to, publish, transmit, sell, resell, create derivative works from, or in any way exploit the Work or make the Work available for others to do any of the same, in any form or by any means, in whole or in part, in each case without the prior written permission of Bentham Science Publishers, unless stated otherwise in this License Agreement.
2. You may download a copy of the Work on one occasion to one personal computer (including tablet, laptop, desktop, or other such devices). You may make one back-up copy of the Work to avoid losing it.
3. The unauthorised use or distribution of copyrighted or other proprietary content is illegal and could subject you to liability for substantial money damages. You will be liable for any damage resulting from your misuse of the Work or any violation of this License Agreement, including any infringement by you of copyrights or proprietary rights.

Disclaimer:

Bentham Science Publishers does not guarantee that the information in the Work is error-free, or warrant that it will meet your requirements or that access to the Work will be uninterrupted or error-free. The Work is provided "as is" without warranty of any kind, either express or implied or statutory, including, without limitation, implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the results and performance of the Work is assumed by you. No responsibility is assumed by Bentham Science Publishers, its staff, editors and/or authors for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products instruction, advertisements or ideas contained in the Work.

Limitation of Liability:

In no event will Bentham Science Publishers, its staff, editors and/or authors, be liable for any damages, including, without limitation, special, incidental and/or consequential damages and/or damages for lost data and/or profits arising out of (whether directly or indirectly) the use or inability to use the Work. The entire liability of Bentham Science Publishers shall be limited to the amount actually paid by you for the Work.

General:

1. Any dispute or claim arising out of or in connection with this License Agreement or the Work (including non-contractual disputes or claims) will be governed by and construed in accordance with the laws of Singapore. Each party agrees that the courts of the state of Singapore shall have exclusive jurisdiction to settle any dispute or claim arising out of or in connection with this License Agreement or the Work (including non-contractual disputes or claims).
2. Your rights under this License Agreement will automatically terminate without notice and without the

need for a court order if at any point you breach any terms of this License Agreement. In no event will any delay or failure by Bentham Science Publishers in enforcing your compliance with this License Agreement constitute a waiver of any of its rights.

3. You acknowledge that you have read this License Agreement, and agree to be bound by its terms and conditions. To the extent that any other terms and conditions presented on any website of Bentham Science Publishers conflict with, or are inconsistent with, the terms and conditions set out in this License Agreement, you acknowledge that the terms and conditions set out in this License Agreement shall prevail.

Bentham Science Publishers Pte. Ltd.
80 Robinson Road #02-00
Singapore 068898
Singapore
Email: subscriptions@benthamscience.net

CONTENTS

FOREWORD i

PREFACE ii

CHAPTER 1 INTRODUCTION TO PYTHON 1
- **INTRODUCTION** 1
- **TECHNICAL STRENGTHS OF PYTHON** 3
  - Portability 3
  - Object-Oriented 3
  - Community Support 3
  - Advanced Features 3
  - Memory Management 4
  - Ease of Use 4
  - Installing Python 4
  - Windows Installer 5
  - Ubuntu 6
  - Linux Mint 6
  - Python IDLE 7
  - Anaconda Open-source Distribution 7
  - Installing Anaconda on Windows 7
  - Installing Anaconda on Linux 9
  - First Python Program 9
  - Python Keywords 10
  - Identifiers 10
  - Statements 11
  - Indentation 12
  - Comments 13
  - Coding Styles 14
  - Functional 14
  - Imperative 14
  - Procedural 15
  - Object-oriented 15
- **CONCLUDING REMARKS** 16

CHAPTER 2 DATA TYPES, OPERATORS, AND EXPRESSIONS 17
- **INTRODUCTION** 17
- **NUMBER** 18
  - Sequence 19
- **DICTIONARY** 21
- **BOOLEAN** 22
- **SET** 23
- **TYPE CONVERSION** 23
  - Implicit Type Conversion 23
  - Explicit Type Conversion 24
- **OPERATORS** 25
  - Arithmetic Operators 25
    - *Addition (+) Operator* 25
    - *Subtraction (-) Operator* 26
    - *Multiplication (*) Operator* 26
    - *Division (/) Operator* 26
    - *Modulus (%) Operator* 26

Exponent (**) Operator 26
Floor Division (//) Operator 26
Relational Operators 27
Greater than (>) Operator 27
Greater than or equal to (>=) Operator 27
Less than (<) Operator 27
Less than or equal to (<=) Operator 27
Equal to (==) Operator 28
Not equal to (! =) Operator 28
Assignment Operator 28
Assignment (=) Operator 28
Add AND (+=) Operator 28
Subtract AND (-=) Operator 28
Multiply AND (=) Operator* 28
Divide AND (/=) Operator 28
Modulus AND (%=) Operator 28
*Exponent AND (**=) Operator* 29
Floor Division (//=) Operator 29
Unary Operators 30
Bitwise Operators 30
Binary AND (&) Operator 30
Binary OR (|) Operator 30
Binary XOR (^) Operator 30
Binary One's Complement (~) Operator 30
Binary Left Shift (<<) Operator 30
Binary Right Shift (>>) Operator 31
Logical Operators 31
Logical AND Operator 31
Logical OR Operator 32
Logical NOT Operator 32
Membership Operators 32
Operator (in) 32
Operator (not) 32
Identity Operators 33
Operator (is) 33
Operator (is not) 33
Operators' Precedence and Associativity 34
EXPRESSIONS 35
Identifiers 35
Literals 35
Operators 35
STRING OPERATIONS 36
Accessing Values in Strings 36
Triple Quotes 37
CONCLUDING REMARKS 42

CHAPTER 3 CONTROL FLOW 43
INTRODUCTION 43
SEQUENTIAL STATEMENTS 43
DECISION CONTROL STRUCTURES 44
If Statements 44

If-else Statements 46
Nested if 46
if-elif-else 49
LOOPING STATEMENTS 50
For Loop 50
While Loop 53
Nested Loop 54
Break Statement 56
Continue Statement 57
The Pass Statement 59
CONCLUDING REMARKS 60

CHAPTER 4 FUNCTIONS 61
INTRODUCTION 61
Definition 61
PREDEFINED FUNCTIONS 62
USER-DEFINED FUNCTIONS 64
Function Call 65
Function Parameters and Arguments 66
Default Arguments 67
Variable Scope and Lifetime 68
Scope of the Variable 68
Local Scope 69
Global Scope 69
The Lifetime of the Variable 69
Local and Global Variables 70
Global Variables 70
Global Statement 71
Return Statement 72
Lambda Functions 72
Recursive Functions 73
Function Redefinition 75
CONCLUDING REMARKS 76

CHAPTER 5 SEQUENCE-STRING AND LIST 77
INTRODUCTION 77
STRING 77
String Concatenation 80
Using + Operator 80
Using join() Method 80
Using % Operator 81
Using Format() Function 81
Appending Strings 82
Using += Operator 82
Using Join() 83
Multiplying Strings 84
Immutable Strings 84
String Formatting Operator 85
Built-in String Functions 86
Slice Operation 88
The ord() Function 90
The chr() Function 90

String Comparison 91
Using Relational Operators 91
Using is and is not 92
Using User-defined Function 92
Iterating String 93
split() Function 94
re.findall() Function 96
The String Module 97
String Module Constants 97
string.ascii_letters 97
string.ascii_lowercase 97
string.ascii_uppercase 97
string.digits 97
string.hexdigits 97
string.punctuation 97
String capwords() Function 98
String Module Classes 99
Formatter 99
Template 99
Regular Expression 100
LISTS 102
Using List Index 103
List Operations 104
Updating Values in Lists 106
Nested List 106
Aliasing 108
Cloning Lists 109
Cloning List using Slicing 109
Cloning List using Extend() 110
Cloning List using List() 110
Cloning List using List Comprehension 111
Cloning List using Append() 111
Cloning List using Copy() 112
Built-in Functions 112
List Built-in List Methods 113
List as Arrays 115
Loops in Lists 116
For Loop in the List 116
While Looping in the List 117
List Comprehension 118
Enumerate() Method 118
Numpy in Loop 119
List in Functions 119
CONCLUDING REMARKS 120

CHAPTER 6 TUPLE AND DICTIONARIES 121
INTRODUCTION 121
TUPLE 121
TheTuple ()Function 123
Accessing Tuple 124
Tuple Indexes 124

Tuple Slicing 125
Updating Tuple 125
Deleting Elements from a Tuple 126
Tuple Operations 127
Tuple Basic Functions 128
Tuple Assignment 130
Tuples and Functions 132
Nesting of Tuples 133
Tuples from String and List 134
zip() Function 135
Inverse zip(*) Function 136
Tuple Sorting 137
DICTIONARY 138
Dictionary Creation 140
Accessing Dictionary Values 142
Modifying Dictionary 143
Delete an Element from the Dictionary 144
The Clear () Function 146
Sorting Dictionaries 147
Sorting Dictionary by Value 148
Sorting Dictionary in Reverse Order 150
Iterating Over a Dictionary 150
Nested Dictionaries 153
Updating Nested Dictionary 155
Deleting Elements from a Nested Dictionary 155
Deleting a Dictionary from the Nested Dictionary 156
Iterating Through a Nested Dictionary 157
Built-in Dictionary Functions 158
The Copy () Method 160
Formatting Dictionaries 161
CONCLUDING REMARKS 162

CHAPTER 7 FILE HANDLING 163
INTRODUCTION 163
FILE 163
Open() Function 164
Close() Function 166
Write() Function 166
Writelines() Method 167
Writing Numbers to a File 168
Read() Method 169
Readline() Method 170
Readlines() Method() 171
Reading Contents with a Loop 171
Creating a New File 172
File Object Attributes 173
File Positions 174
File Rename 176
Deleting a File 177
Binary Files 177
Directory Operations 178

Current Working Directory 178
List of Directories 179
Creating a Directory 179
Change Directory 180
Renaming a Directory 180
Delete a Directory 180
CONCLUDING REMARKS 181

CHAPTER 8 EXCEPTION HANDLING 182
INTRODUCTION 182
HANDLING EXCEPTIONS 184
EXCEPT BLOCK 186
ELSE AND FINALLY, KEYWORD 188
RAISING EXCEPTIONS 189
Built-in Exceptions 191
User-defined Exceptions 193
CONCLUDING REMARK 194

CHAPTER 9 MODULES AND PACKAGES 195
INTRODUCTION 195
MODULES 195
Import Statement 196
From…Import Statement 197
Renaming A Module 198
Dir() Built-In Function 199
Reload () Function 200
Built-In Modules 200
PACKAGE 202
Creating Package 202
Sub Packages 206
CONCLUDING REMARK 207

CHAPTER 10 OBJECT-ORIENTED PROGRAMMING 208
INTRODUCTION 208
CLASS AND OBJECT 209
DATA ABSTRACTION 209
ENCAPSULATION 209
INHERITANCE 209
POLYMORPHISM 210
Defining a Class 210
Creating an Object 211
Deleting Properties or Object 212
Data Abstraction Example 215
CONSTRUCTORS 216
Parameterized Constructor 218
Default Constructor 218
INHERITANCE IN PYTHON 219
TYPES OF INHERITANCE 220
Single Inheritance 220
Multi-level Inheritance 222
Multiple Inheritance 224
Hierarchical Inheritance 226

The issubclass(sub,sup) method() 227
The isinstance (obj, class) method() 228
POLYMORPHISM IN PYTHON 229
Polymorphism in Operators 229
Function Polymorphism 229
Polymorphism in Class Methods 230
Method Overriding 231
Built-in Class Functions 232
Built-in Class Attributes 233
Static Variables 234
CONCLUDING REMARKS 236

CHAPTER 11 PYTHON FOR MACHINE LEARNING 237
INTRODUCTION 237
IMPORTANT PYTHON LIBRARIES 239
• NUMPY 239
• PANDAS 240
• SCIKIT-LEARN 241
• MATPLOTLIB 243
• TENSORFLOW 244
• KERAS 245
• PYTORCH 245
• NLTK 246
Installing NLTK on Mac/ Unix 247
Installing NLTK on Windows 247
CONCLUDING REMARKS 250

CHAPTER 12 PROGRAMMING WITH PYTHON 251
INTRODUCTION 251
BASIC PYTHON PROGRAM 251
Program to Solve a Quadratic Equation 251
Program to Swap Two Numbers 252
Program to Find the Factorial of Two Numbers 252
Program for Fibonacci Series Using Recursion 253
Sorting Elements of Array in Ascending Order 254
Program to Print the Sum of Elements of Array 254
Program to Transpose a Matrix 255
Program to Reverse a String 255
Program to Concatenate Two Strings 256
Program to Append Elements to the List 256
Program to Delete an Element of List 256
Program for Linear Search 257
Program for Binary Search 257
Program to Perform Time Series Analysis 258
CONCLUDING REMARKS 259

BIBLIOGRAPHY 260

SUBJECT INDEX 261

FOREWORD

The fact that Python programming is employed across most fields contributes to its growing popularity in recent years. It is possible to generate software at any level, from the simplest of programs to a fully functional application. Even every company, institution, organization, or researcher uses Python programming in their work, albeit they may do so in various methods. Python makes it possible to develop algorithms that are both effective and efficient as well as to manage more significant amounts of data in today's world. The proposed book covers various Python programming topics, ranging from the fundamentals to more advanced techniques. This book, which serves as a helpful guide to becoming a programmer, will significantly benefit the community.

D. S. Rajput

School of Information Technology and Engineering
Vellore Institute of Technology
Vellore, India

PREFACE

It gives immense pleasure to bring the book "**Basics of Python Programming: A Quick Guide for Beginners**." The most extensively used programming language today is Python, which also happens to be the most popular programming language. Students enrolled in various classes who can efficiently use this high-level programming language as a problem-solving tool are this Book's target audience. Python is not only employed in the field of computer science; instead, it is used in the development of applications for all areas.

Python programming can be swiftly learned by anybody with a fundamental understanding of computers and the ability to reason logically. Because of this inspiration, we could write this Book clearly and concisely. After reading the Book, you will have a fundamental understanding of how to do programming in Python. We have attempted to present the intricacies of Python in a very colloquial language such that the potential readers require no special expertise to refer to the book. It is apt for beginners as the concepts are explained in simple language with suitable demonstrative examples to facilitate both theory and practical learning.

Our primary goal in writing this book is to provide an approachable resource for beginners new to programming or with limited coding experience. We understand that learning a programming language can be intimidating, especially for those starting from scratch. With "Basics of Python Programming: A Quick Guide for Beginners", we have consciously designed the content to be beginner-friendly, focusing on simplicity and clarity of explanations. We believe that our book's accessible style will empower beginners to grasp the fundamental concepts of Python programming swiftly and confidently.

We recognize that many aspiring programmers are looking for a resource that allows them to learn Python quickly and efficiently. While there are extensive books available that provide in-depth coverage of Python, our book takes a different approach. We have distilled the core concepts and essential components of Python programming into a concise guide that can be absorbed quickly. By focusing on the basics, we aim to provide beginners with a solid foundation in Python programming without overwhelming them with excessive information.

We firmly believe that practical application is critical to mastering Python programming. Therefore, our book emphasizes hands-on learning and incorporates numerous practical examples throughout the chapters. By engaging in coding exercises and mini-projects, readers can actively apply the concepts they learn, solidifying their understanding of Python. Our approach encourages learners to gain practical experience alongside theoretical knowledge, enabling them to build their coding skills from the very beginning.

We firmly believe that "Basics of Python Programming: A Quick Guide for Beginners" offers a unique value proposition to individuals looking to kick-start their journey in Python programming. Its simplicity, efficiency, hands-on approach, clear progression, and supplementary resources set it apart from other books. We hope this book is a valuable tool for learning Python programming and unlocks the door to exciting possibilities.

Krishna Kumar Mohbey
Department of Computer Science
Central University of Rajasthan
Rajasthan, India

&

Malika Acharya
Department of Computer Science
Central University of Rajasthan
Rajasthan, India

CHAPTER 1

Introduction to Python

Abstract: Python is an object-oriented programming language that can support a wide range of applications like web development, desktop applications, *etc*. Its general-purpose programming features make coding easy, comprehensive and readable even for beginners.

Keywords: Development environments, Object-oriented programming, Operating system, Shuffle exchange network, Virtual machine.

INTRODUCTION

Python has grown to be a mainstream language. Its simple syntax and comprehensible code make it a popular general-purpose programming language among developers, engineers, and amateurs with limited programming skills. Its open-source features with a wide variety of libraries facilitate efficient programming. The first chapter of this book provides a brief introduction to Python along with a stepwise guide to installation on Windows and Linux. It will familiarize you with the Python IDE and editors, thus paving your journey to Python programming.

Guido van Rossum, Python's Benevolent Dictator for Life, developed Python in the 1990s. After that, several versions of Python were released. With the end of life of version 2.7, currently, versions 3.6 and 3.9 are widely used. Python is an open-source project maintained by the Python Software Foundation. We can take a tour of the Python universe at www.python.org.

Python is an object-oriented language that, due to its interoperability with existing codes in C and Fortran, gears up to the developer's demands and enhances the programmer's productivity while cutting down the time consumed. Python has become the developer's choice in various fields. For example, some of the prominent areas where Python is used are NASA's Jet Propulsion Laboratory, the Lawrence Livermore National Laboratory, Shell Research Boeing, Industrial Light and Magic, Sony Entertainment, and Procter & Gamble. Python is a high-level language; thus, the programs coded in it are easy and comprehensible. Python codes execute on a virtual machine; hence, a layer of abstraction exists

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

between the code and the executing platform. Thus, Python is an interpreted language and produces portable codes that are cross-platform executable. Fig. (**1**) depicts the execution of a simple Python program. Unlike Fortran, Python is a dynamically typed language that uses an interpreter to interpret the representative types at the run time. The layer of abstraction abstracts the underlying optimization from the code. Python binds Fortran and C libraries using an interpreter to perform intensive computation.

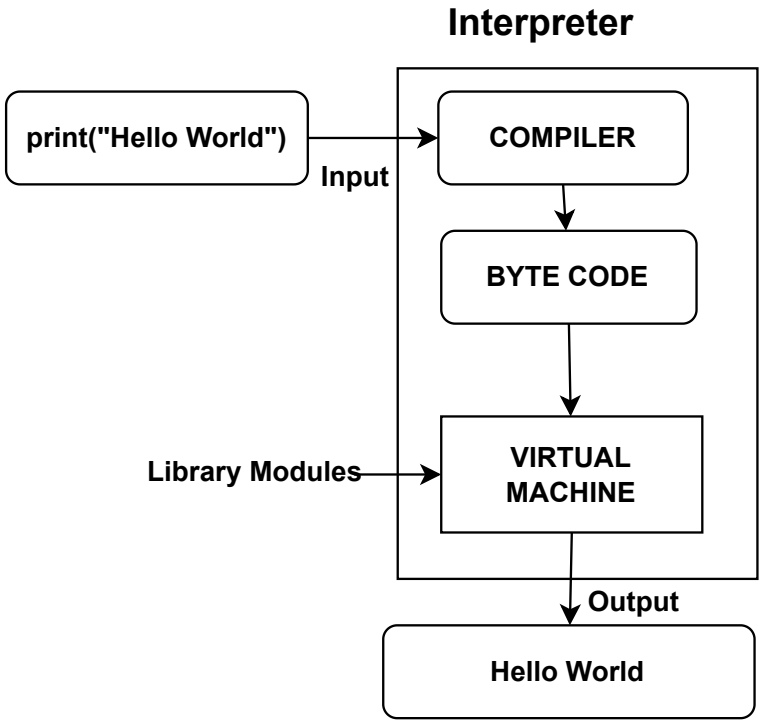


Fig. (1). Execution schematic of Python code.

You must have previously understood various languages like C++, Java, Perl, Scheme, or BASIC. All these languages are high-level languages. Nevertheless, computer hardware understands only low-level language called machine language. So, high-level language needs to be converted into machine language. For this, we have two translators: a *compiler* and an *interpreter*. A compiler is a program that translates the entire program into machine language. The high-level program is known as the source code, and the output is machine code. An interpreter is also a program that takes high-level language and converts it into machine code instruction by instruction. Compiled code can be run any number of times without repeated compiling or the source code.

In contrast, interpreted code requires an interpreter and source code every time. Thus, interpreted code makes the programming environment more user-friendly and enhances portability. Every CPU has its machine code, but we can run a high-level language program with the interpreter.

TECHNICAL STRENGTHS OF PYTHON

Portability

As already discussed, Python is a portable language that requires Byte Code for execution on any platform. An executable byte code converts the source code to platform-independent code. Python consists of a Tkinter toolkit to support the Tk GUI interface so the graphical user interface can run on all GUI's supported platforms without program changes. Python's original, standard implementation was given in ANSI C, making it executable on all major platforms ranging from PDAs to supercomputers.

Object-Oriented

Python is an object-oriented programming language. The language is extensible, *i.e*., the programs can be extended to C, C++, and Java. It uses a class model to support polymorphism, operator overloading, and other notions. It is a powerful scripting tool for other object-oriented system languages, like C++, Java, and C#. The recent versions of Python also support functional programming. This includes generators, comprehensions, closures, maps, *etc*.

Community Support

Python community support is quite active and responsive to the user's queries. The community consists of Python creator Guido van Rossum, the Benevolent Dictator for Life (BDFL), and a crew of thousands of workers. Python is more conservative than other languages in terms of changes, *i.e*., the changes need to be approved by the community, especially BDFL.

Advanced Features

Python provides full support for features of scripting languages like PERL. The scheme facilitates the use of software development tools that are easily found in compiled languages. It is a product of the FLOSS community, *i.e*., Python is a Free, Libre, and Open-Source Software that assists in knowledge sharing. Some of the salient features of Python are:

Memory Management

Memory allocation and reclamation are no longer the programmer's headache. It adopts automatic memory management allocation and garbage collection so that memory growth and shrink are as per demand. Python considers memory as the private heap. There are two types of allocators for memory management: raw memory allocator and object-specific allocator:

- **Large system development**: Python, with its vast repository of tools, allows faster and easier development of extensive systems. Python tools can be classified into two categories: general-purpose and functional tools. The generic tools include modules, classes, and exceptions that help to disintegrate large applications into small modules, reuse customized codes, and handle errors and exceptions.
- **Dynamic typing**: The programmer can easily write code without worrying about the program's syntax and structure. There is no need for the type and variable declaration beforehand.
- **Vast Library and Tools**: Python has full-fledged support for various built-in objects types like lists, dictionaries, *etc*. Python's standard operations, like concatenation, slicing, sorting, *etc*., allow sequential processing of these object types. Further, Python has some precoded library tools to support application-level specific tasks like networking, regular expressions, *etc*. It also has an extensive library to handle threading, GUI, *etc*. The easy integration of Python allows customization and communication with other frameworks. For example, integration of Java and .NET components allows easy communication with frameworks like COM and Silverlight.

Ease of Use

Python has become the developer's first choice due to its interactive user interface and rapid turnaround time. Python is quite easy to learn with its simple and flexible programming paradigm. Python majorly emphasizes on quality and comprehensibility of code rather than its structure. The absence of lengthy compilation time and immediate execution boosts the developing time.

Installing Python

Visit the official website www.python.org and get the latest version of Python. Fig. (**2**) shows the webpage. If the operating system is Mac or Linux, Python is already installed, although it might be an older version.

Fig. (2). Selection of Python for the operating system.

Windows Installer

1. Once you visit the Python official website, it will prompt you to download the latest version of Python for the respective operating system. You can select the respective operating system and it will take you to the available versions of Python for that operating system.

2. Select a Python version depending on your configuration (32-bit or 64-bit) from the list of options and it will start the automatic download of the executable file.

3. Once the download is complete, go to the downloading site and run the executable file.

4. A window will pop up that shows the default installation destination and also provides the option to customize the installation destination by browsing the same. Further, it prompts two checkboxes to *Add a Python Version to PATH* and to use the administrator privileges. This will confirm that the interpreter is added to the execution path.

5. Click ***Install*** Now. You will be prompted to *Optional Features* . Select the features as required.

6. Click ***Next*** . You will be prompted to the ***Advanced options*** features. Select the features as required. Also, here you will be required to choose the installation location.

7. Click ***Install*** . Your setup progress bar will show the progress of the installation. After the installation is complete a window will prompt the successful installation.

8. Once the installation is complete, you can open it by searching in the taskbar.

Ubuntu

With different versions of the Ubuntu distribution, Python is preinstalled. You might require updating the version of Python. The steps for Python installation on Ubuntu are as follows:

1. Check the version of Python installed on your Ubuntu system.

```
python -version
```

2. Update and refresh the list of the repository.

```
sudo apt update
```

3. Installation of Supporting Software.

```
sudo apt install software-properties-common
```

4. Add Deadsnakes PPA ((Personal Package Archive)).

```
sudo add-apt-repository ppa:deadsnakes/ppa
```

5. You will be prompted to press enter for installation continuation. Press Enter. Refresh the package lists again.

```
sudo apt update
```

6. Install Python 3 using the following command.

```
sudo apt install python3.9
```

7. Verify the Python version after successful installation.

Linux Mint

Installation of Mint is like Ubuntu, and installation instructions for Ubuntu can also be used in Mint. A Deadsnakes/PPA working well with Mint.

Python IDLE

IDLE is the acronym for Interactive Development and Learning Environment. This IDLE helps to run your programs. Python has an IDLE for Windows that provides an interactive platform for program development. IDLE for Python comes in two modes: interactive mode and script mode. IDLE starts in the shell; you can type your code and see the output on the same window. Fig. (**3**). depicts the Python shell. You can search the Python shell in the Windows search bar.

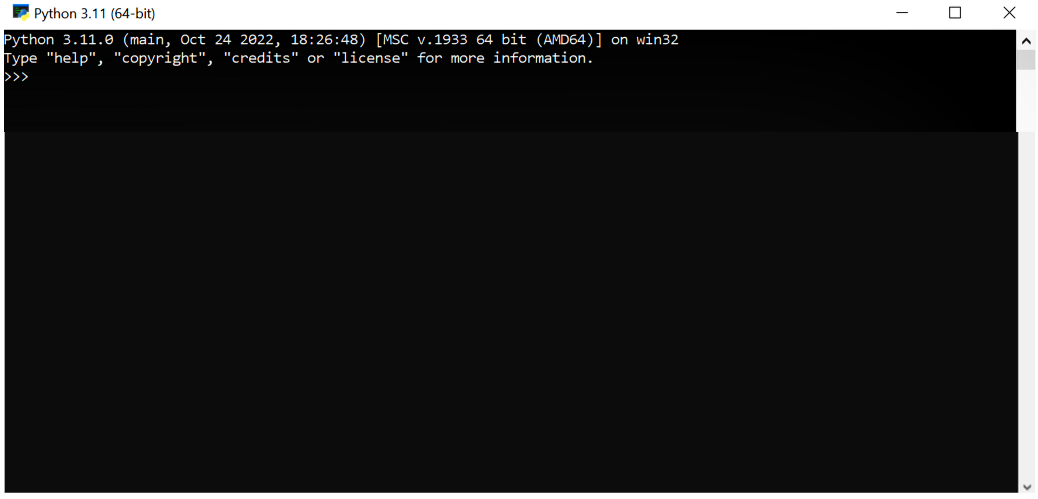


Fig. (3). Python Shell.

Anaconda Open-source Distribution

Anaconda, a Python package manager, provides applications suitable for large data processing, scientific computing, *etc*. It works for both Python and R programming languages. It provides various development platforms like JupyterLab, Jupyter, Notebook, Spyder, Glue, Orange, RStudio, and Visual Studio Code.

Installing Anaconda on Windows

1. Visit the official website of Anaconda: https://www.anaconda.com/distribution/#download . Fig. (**4**) depicts the site page. This link will prompt you to select the operating system on which you require Anaconda to be installed. Selecting the operating system would lead to the automatic downloading of the executable file.

2. Download the executable file. Run the installer on your computer.

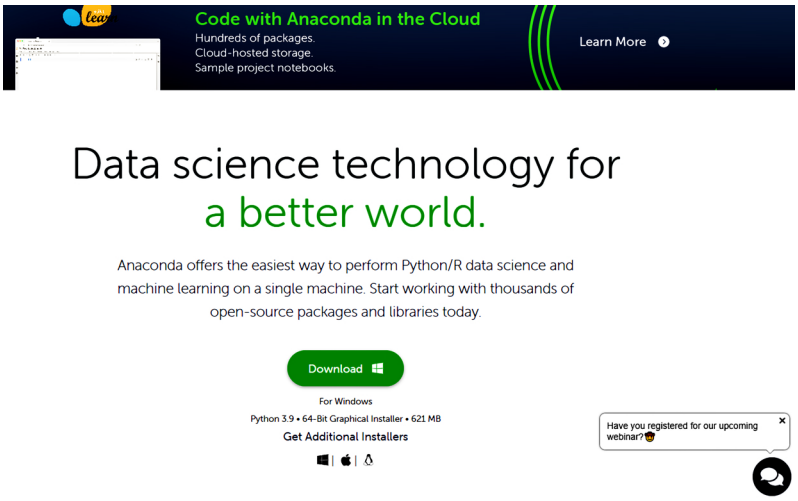


Fig. (4). Anaconda site page for Windows.

3. Once you select to download Anaconda, you will be prompted to agree to the license. Select the ***I Agree*** button to proceed.

4. You will be prompted to select ***Add path*** to the environment. Further, it will also prompt you to select where you want to install the Anaconda. Browse the installation destination and click ***Next*** .

5. In the next step select the installation type. In this, you get to select the user type for which the installation is to be done. It prompts two options, *Just me* or *All Users* . The former is used for the current user account and the latter is used for all users in the system, but it requires the users to consider administrator privileges. After selecting the type of installation, click on ***Next.***

6. Installation begins with the progress bar displaying the package being installed. In this window, there is a ***Show Details*** button from where you can check the progress of the installation along with the packages installed so far. After the installation is complete the window will display the " *Completed"* message.

7. Once the installation is complete, Anaconda provides two checkboxes one for starting the distribution tutorial and another to start with the Anaconda. Click on the ***Finish*** button to start the Anaconda shell.

8. You can open the Anaconda Navigator after the successful installation. In that Navigator, there will be several platforms available to start the coding. Over here we will focus on **Jupyter** and **Spyder** platforms.

Installing Anaconda on Linux

1. You can also download the file from the *curl* and *wget* retrieval commands.

2. Run the shell script using the bash command.

3. You will be prompted to select the destination folder and other settings during the installation. For Linux systems, keep the standard settings.

4. After installation, create an environment to install packages. You can also build an individual environment for each project.

First Python Program

In this section, we will write the first program using **Spyder**. Open Anaconda Navigator and select Spyder. Click on the "Launch" button present below the Spyder logo. Spyder IDE will open. Spyder IDE consists of three sections. The left pane is the workspace where you write code. The right pane is divided into two sections. The upper section is used for variables and object analysis. The lower pane is the console where the output occurs after successfully compiling the code. In IDE, you can write the following code in the left pane and execute it to see the result on the console.

Another very common application used for Python programming is **Jupyter**. Click on the launch button below the icon of Jupyter. It will prompt to the list of folders as depicted in Fig. (**5**). You can click on the "New" button to start the new Jupyter Notebook. The Jupyter Notebook is depicted in Fig. (**6**). You can type your code in the workspace and run each cell to get the output.

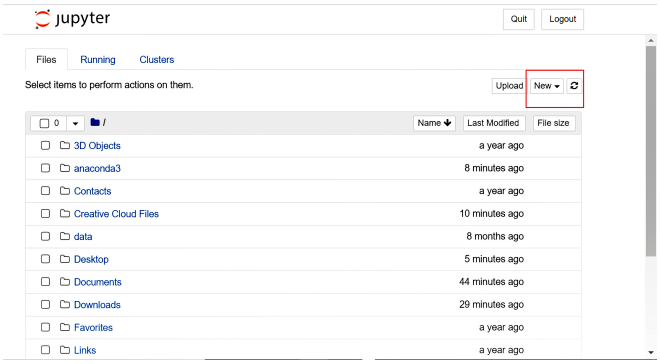


Fig. (5). Jupyter Home.

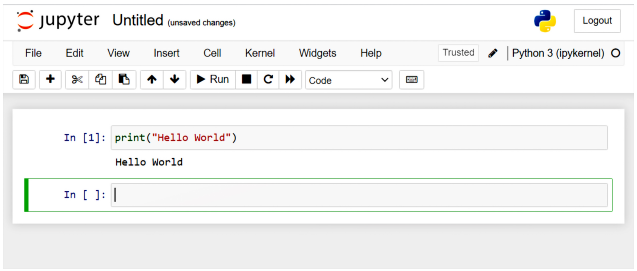


Fig. (6). Jupyter Notebook.

Example:

```
1. # Display a Name
2. print ("Hello World")
3. #Function displaying a name
```

Output:

```
Hello World
```

Python Keywords

Python keywords are words reserved for specific purposes and cannot be used for anything else. In total, Python has 35 keywords. The list of keywords has been revised many times. In Python 2.7, *print* and *exec* were keywords, but in Python 3+, they were eliminated, and in Python 3.7, *await* and *async* have been added as the new keywords. Below is the list of keywords for Python 3.9. You can use the help command to get the keywords in Python.

False	Def	if	Try	assert	except	nonlocal
Class	True	raise	In	else	lambda	yield
from	Pass	and	Elif	is	with	break
Or	global	del	As	while	await	for
None	continue	import	Return	async	finally	not

Identifiers

An identifier is a name that identifies a variable, function, class, object, *etc*. Some rules for identifier conventions are:

1. Do not use special characters like @, !, &, *, *etc*, and punctations in the identifier names.

2. An identifier should not start with a digit.

3. Identifiers are highly case-sensitive. Hence, ***atoms*** and ***Atoms*** are different identifiers in Python.

4. An identifier can start with alphabets *A to Z* and *a to z* or an underscore (_) that may or may not be followed by letters, underscores, or digits (0-9).

5. Do not use the keywords as the identifiers.

6. A class name must always begin with an uppercase alphabet.

7. A private identifier begins with a single underscore. A strongly private identifier begins with two underscores. Moreover, a special name ends with two trailing underscores.

y, dept, course_in_cs, course1, course_2, _grade, grade_2 ➔ **valid identifiers**
4, dep in college, cs-course, 1 st, 1_exam ➔ **Invalid identifiers**

Statements

In Python, a statement can be defined as an instruction or a command that the interpreter can execute. There are several kinds of statements in Python that offer a variety of purposes. We can compose a single line of multiple statements separated by semicolons. The various types of statements are:

1. Expression statement: They are used to compute, write a value or call a procedure.

2. Assignment statements: They assign the values and modify the items.

3. Assert statements: They provide debugging assertions.

4. Pass statements: They are much like null operations; no operation takes place when they are executed.

5. Del statements: As the name suggests, they specify the deletion targets.

6. Return statements: They are specified after the function block and defined when the program leaves the current function block.

7. Yield statements: They are used in the *generator* function only.

8. Raise statements: They raise the exceptions. If there exist no new exceptions, then they re-raise the exception that is currently being handled.

9. Break statements: They are usually used with loops (*for, while*) and with try statements. They end the execution sequence and move the program out of the current block in the loop.

10. Continue statements: They stop the current iteration in the loop and begin with the next iteration.

11. Import statements: They are used to import the modules that are required for easy pre-processing.

12. Global statements: They are used to define the identifier that remains valid for the entire code block.

13. Nonlocal statements: They bind the identifier to the nearest block so that searching for the identifier in the local namespace is very easy.

Indentation

In Python, indentation plays a very important role. Indentation can be defined as the blank spaces before a few statements. Besides enhancing the code readability, indentation also defines the scope of the function, much like the curly braces ({ }) in C and C++. Fig. (**7**) depicts the indentation for the loop. The indentation rules are as follows:

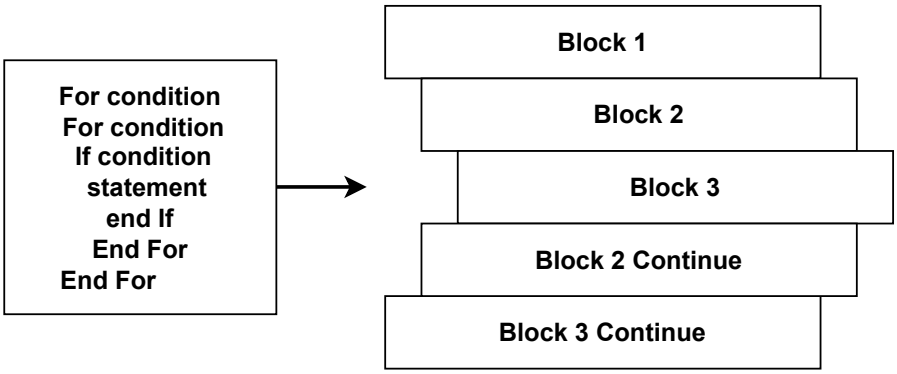


Fig. (7). Indentation in Python.

1. “Four spaces” are the default indentation spaces.

2. There cannot be an indentation in the first line of the code.

3. Maintain the uniformity of code, *i.e.*, the number of spaces must be uniform.

4. Prefer the use of whitespaces rather than tabs for indentation.

Comments

Comments in Python are statements added in the code by the programmers that indicate the purpose of the code block. They increase the readability of the code. They are not executed at the run time. The various purposes of comments are:

1. Enhanced readability.

2. Inclusion of resources.

3. Reuse of the existing codes.

4. Easy comprehension of the code and handy explanation.

Comments start with a # symbol and can be placed at the end of a line. There are two types of comments in Python. These are:

Single-line comments and *Multiline comments*.

• Single line comments

It starts with a # symbol which means you have to write the # symbol in every line.

• Multiline comments

A multiline comment in Python is a block of text enclosed by a delimiter (“”””) at the beginning and end of the comment. There should be no white space in delimiters (“”””).

Let us take an example for understanding comments.

Example:

```
print ("Python Programming") # Single line comment
"""
This is a comment
written in
more lines
"""
print ("Book for Computer Science.")
```

Output:

```
Python Programming

Book for Computer Science.
```

Coding Styles

Python offers four different coding styles, as discussed below.

Functional

In this style, the problem is divided into different functions, and each statement is considered a Mathematical equation. It is widely used in parallel processing tasks and lambda calculus. It is the first choice for large data processing jobs.

Example:

```
1. my_list = [2,3,6,7,8,10]
2. new_list = list(map(lambda x: x + 2, my_list))
3. print(new_list)
```

Output:

```
[4, 5, 8, 9, 10, 12]
```

Imperative

This style is used explicitly for various operations on data structures. It requires a pre-definition of the order of operations.

Example:

```
1. my_list = [2,3,6,7,8,10]
2. sum = 0
3. for x in my_list:
4. sum += 2*x
5. print(sum)
```

Output:

```
72
```

Procedural

It is the most adopted style by nonprogrammers. It primarily focuses on the code. The entire code is divided into a list of instructions, like breaking a problem into simpler tasks. It uses sequencing, selection, modularization, and iteration to reduce code redundancy.

Example:

```
1. my_list = [2,3,6,9,18,20]
2. def add(list):
3. sum = 0
4. for x in the list:
5. sum += x
6. return sum
7. print(add(my_list))
```

Output:

```
36
```

Object-oriented

In the procedural style, the focus was on the given code, but in this style, the focus is on the data. The merit of this style is the formation of a class paradigm and adherence to object-oriented principles like abstraction, inheritance, polymorphism, and encapsulation.

Example:

```
1. class A:
2. def test(self,a,b):
3. return(a+b)
4. b=A()
5. print("res=", b.test(45,77))
```

Output:

```
res=122
```

CONCLUDING REMARKS

In this chapter, we furnished a general introduction to Python. This chapter also guided the readers through the installation of Python and Anaconda over different operating systems and for different configurations. We also familiarised the readers with some basic concepts of Python programming. In the next chapter, we will look deeply into Python data types and operators.

CHAPTER 2

Data Types, Operators, and Expressions

Abstract: Now that you have been familiarised with the installation and basics of Python programming, it's time to dig in a little more and understand the different data types that are available along with the operators and expressions that make the programming more user-friendly. In this chapter, we will learn about data type categorization and the operators present. Here you will learn the following nuances:

1. Data types supported by the Python.
2. Use of variables to store and access the data.
3. Operators do mathematical works and logical functions.
4. Variety of expressions to serve the range of applications.

Keywords: Associativity, Literals, Operators, Precedence.

INTRODUCTION

In computer programming, a data type is a classification of the many kinds of information that may be saved in a variable. Since Python is a dynamically typed language, we do not need to specify the variable type when we declare it.

The value is bound to its type in a way that the interpreter does not explicitly specify—for example, **X=100**. We did not declare the type of the variable **X**, yet it now holds the integer value 100. In Python, the integers are by default treated to be of integer data type. To check the type of the variable, we use **type()** function available to us in Python, and it returns the type of the variable that was handed in. The following example illustrates how to specify the values for a variable and to check their types.

Example:

```
1.    x=100
2.    y="Python Programming"
3.    z= 12.59
4.    print (type(x))
5.    print (type(y))
6.    print (type(z))
```

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

Output:

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

Python has a wide variety of standard data types, each of which can have a unique storage mechanism defined for it. The data types defined in Python are listed below:

1. Numbers
2. Sequence
3. Dictionary
4. Boolean
5. Set

NUMBER

The number is responsible for storing numerical values. A Python Numbers data type is responsible for storing all integer, float, and complex values. To check the data type of a variable, Python has a function called **type()**. It also has the **isinstance()** function that checks whether or not an object belongs to a specific class. When a number is assigned to a variable in Python, Number objects are created automatically. A number data type can store int, float, and complex type values.

- The integer value can be any length. There is no limitation on the length of an integer in Python.
- Float is used to store floating-point numbers which is accurate up to 15 decimal points.
- A complex number is specified in the form of **a + ib**, where **a** and **b** stand for the real and imaginary components of the number, respectively.

Example:

```
1.  a = 10
2.  print ("The type of a is", type(a))
3.
4.  b = 10.15
5.  print ("The type of b is", type(b))
6.
7.  c = 2+4j
8.  print ("The type of c is", type(c))
9.  print ("c is a complex number", isinstance(2+4j,complex))
```

Output:

```
The type of a is <class 'int'>

The type of b is <class 'float'>

The type of c is <class 'complex'>

c is a complex number True
```

Sequence

The language treats Python's string, list, and tuple datatypes as sequence types. The character sequence enclosed in quote marks is an example of the string, which may be defined as that sequence. When defining a string, Python allows us to use single quotes, double quotes, or triple quotations. As Python includes predefined functions and operators that may be used to execute various actions on strings, managing strings in Python is a simple process. The *"Hello World!"* is the result of the operation *"Hello"* + *"World!",* which uses the plus (+) operator to concatenate the two strings. Similarly to repeat the strings we can use the repetition operator (*). For example "India" *7 outputs IndiaIndiaIndiaIndiaIndiaIndiaIndia '.

Example:

```
1.      S1= "Hello World"
2.      print (S1)
```

Output:

```
Hello World
```

Lists in Python function is similar to arrays in C. The list, however, may include information of varying formats. The list's contents are surrounded in square brackets [] and separated by commas (,). We may selectively get the list's contents by using slice [:] operators. The list may be manipulated using the same concatenation (+) and repetition (*) operators as strings.

Example:

```
1.  L1 = [10, "Hello", "World!", 20, 30]
2.  L2 = [1, "Hello", 2.0]
3.  #Checking the type of given list
4.  print (type(L1))
5.  print (type(L2))
6.  #Printing the list
7.  print (L1)
8.  print (L1[3:])
9.  print (L2[0:1])
10.
11.  # List Concatenation using + operator
12.  print (L1 + L2)
13.
14.  # List repetition using * operator
15.  print (L2 * 4)
```

Output:

```
<class 'list'>

<class 'list'>

[10, 'Hello', 'World!', 20, 30]

[20, 30]

[1]

[10, 'Hello', 'World!', 20, 30, 1, 'Hello', 2.0]

[1, 'Hello', 2.0, 1, 'Hello', 2.0, 1, 'Hello', 2.0, 1, 'Hello', 2.0]
```

A tuple is very much like a list in many ways. Tuples, much like lists, are collections of things that can be of any data type. We use parenthesis to specify the components of a tuple and different elements of the tuple are separated by commas (,). Because the size and value of the elements included in a tuple cannot be changed, the tuple data structure is said to be read-only.

Example:

```
1.  T1 = [10, "Hello", "World!", 20, 30]
2.  T2 = [1, "Hello", 2.0]
3.
4.  # Checking type of tuples
5.  print (type(T1))
6.  print (type(T2))
7.  #Printing the tuple
8.  print (T1)
9.
10. # Tuple slicing
11. print (T1[2:])
12. print (T1[0:1])
13.
14. # Tuple concatenation using + operator
15. print (T1 + T2)
16.
17. # Tuple repetition using * operator
18. print (T1 * 2)
```

Output:

```
<class 'list'>
<class 'list'>
[10, 'Hello', 'World!', 20, 30]
['World!', 20, 30]
[10]
[10, 'Hello', 'World!', 20, 30, 1, 'Hello', 2.0]
[10, 'Hello', 'World!', 20, 30, 10, 'Hello', 'World!', 20, 30]
```

DICTIONARY

A dictionary is a collection of key-value pairs that are not arranged in any order. It functions similarly to an associative array or a hash table, in which the individual keys each contain a distinct value. The value of a key can be any arbitrary Python object, although the key itself can store any primitive data type. The entries in the dictionary are delineated from one another using commas (,), and they are encapsulated in curly braces ().

Example:

```
Dic = {101:'Amit', 102:'Sumit', 103:'Rahul', 104:'James'}
 # Printing dictionary
print (Dic)
print("Your first record is : "+Dic[101])
print("Your fourth record is:  "+ Dic[104])
# Printing keys and values
print (Dic.keys())
print (Dic.values())
```

Output:

```
{101: 'Amit', 102: 'Sumit', 103: 'Rahul', 104: 'James'}
Your first record is: Amit
Your fourth record is: James
dict_keys([101, 102, 103, 104])
dict_values(['Amit', 'Sumit', 'Rahul', 'James'])
```

BOOLEAN

The **True** and **False** values are predefined using the Boolean data type. The supplied values are compared to these to check whether the assertion is true or false. The bool class is used to indicate what it is. The value '**T**' or any other value that is not zero can represent true, whereas the value **'0'** or the letter **'F'** can be used to represent false.

Example:

```
x = bool(1)
#display x:
print (x)
#display the data type of x:
print (type(x))
```

Output:

```
True
<class 'bool'>
```

SET

The data type known as Set in Python is a collection that is not sorted. It is possible to edit it after it has been created, it may be repeated, and it has components that are not found anywhere else. The order in which the items are presented in a set is indeterminate, hence the return sequence may be in an alternative order. The built-in method **set()** is used to specify the set data type**;** alternatively, a series of items may be given between the curly braces and delimited by commas to build the set. It is capable of holding a wide variety of values.

Example:

```
1.  emp_set = set()
2.  set_data = {'Walter', 74, 'Book'}
3.  #Display values of the set
4.  print (set  data)
5.  #Inserting data into the  set
6.  set_data.add(1282)
7.  print (set_data)
8.  #Delete a record  from the set
9.  set_data.remove(74)
10. print (set_data)
```

Output:

```
{'Book', 'Walter', 74}

{'Book', 'Walter', 1282, 74}

{'Book',  'Walter', 1282}
```

TYPE CONVERSION

It converts the value from one data type to another. The data type may be an integer, string, float, *etc*. There are two types of conversion: *Implicit Type Conversion* and *Explicit Type Conversion.*

Implicit Type Conversion

In this conversion, Python automatically converts one data type to another data type. A user is not involved in this type of conversion. Let us take an example of implicit type conversion.

Example:

```
num_int = 234
num_float = 23.4
num_new= num_int + num_float
print ("datatype of num_int:",type(num_int))
print ("datatype of num_flo:",type(num_float))
print ("Value of num_new:",num_new)
print ("datatype of num_new:",type(num_new))
```

Output:

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>
Value of num_new: 257.4
datatype of num_new: <class 'float'>
```

Explicit Type Conversion

In this conversion, users convert the data type of an object to the required data type. It uses some predefined functions like *int(), float(), str()*, *etc*., to perform explicit type conversion. Moreover, it is also called typecasting because the user casts (changes) the objects' data type. For typecasting, the variable for which the type conversion is required is specified along with the requisite data type. There is the following syntax of explicit type casting.

```
<required_datatype>(expression)
```

Example:

```
x = 345
y = "867"
print ("x has the type :",type(x))
print ("y has the type :",type(y))

y = int(y)
print ("After Type Casting y is :",type(y))
sum = x + y
print ("Sum :", sum)
print ("Sum has the data type:",type(sum))
```

Output:

```
x has the type : <class 'int'>
y has the type : <class 'str'>
After Type Casting y is : <class 'int'>
Sum :1212
Sum has the data type : <class 'int'>
```

OPERATORS

In Python, the idea of carrying out anything from a computational perspective is called an operator. In this context, we use two different terminology forms: operators and operands. An operand is a value being operated on by an operator, and an operator is a symbol being executed with the assistance of the operands. It is a special kind of symbol that carries out an operation on its operands and returns the outcome of that action. Take statements **10 + 20** as an example to help you better comprehend the concept. The numbers 10 and 20 make up the operands, while the plus **(+)** sign is the operator. Therefore, the evaluation of this example will consist of a straightforward arithmetic expression, and the outcome will be the addition of two numbers. The various operators available in Python are enlisted below:

1. Arithmetic Operators
2. Relational Operators
3. Assignment Operator
4. Unary Operators
5. Bitwise Operators
6. Logical Operators
7. Membership Operators
8. Identity Operators

Arithmetic Operators

It performs the basic mathematical operation on the numeric operands. The arithmetic operators return the result, and the outcome depends on the type of operands you use in programming. Arithmetic operators are listed below:

Addition (+) Operator

The symbol + is used while using this operator. It uses two operands and adds the two values. For example, a=1, b=2, then a + b returns 3.

Subtraction (-) Operator

The symbol – (minus) is used while using this operator. It uses two operands, and this operator subtracts the value and returns the result. For example, a=4, b=2, then a - b returns 2.

Multiplication (*) Operator

The symbol * is used while using this operator. It operates between two operands, and this operator multiplies these values and returns the result. For example, a=4, b=2, then a*b returns 8.

Division (/) Operator

The symbol (/) uses while using this operator. It uses two operands; this operator divides the value and returns the result. For example, a=4, b=2, then a / b returns 2.

Modulus (%) Operator

The symbol % is used while using this operator. It uses two operands, and this operator returns the remainder. For example, a=4, b=2, then a % b returns 0.

Exponent (**) Operator

The symbol ** is used while using this operator. It operates between two operands. It performs exponential (power) calculations on operators.

For example, a=4, b=2, then a ** b means $(4)^2$. The result will be 8.

Floor Division (//) Operator

Floor division is like regular division but returns the highest possible integer. This integer is less than or equal to the outcome of normal division.

For example, 7//2 = 3 and 7.0//2.0 = 3.0, -11//3 = -4, -11.0//3 = -4.0

Example:

```
1. a=4
2. b=2
3. f=a/b        #Division (/)
4. print (f)
5. g=a%b        # Modulus (%)
6. print (g)
7. h=a**b       #Exponent (**)
8. print (h)
9. i=a//b       #Floor Division (//)
10.  print (i)
```

Output:

```
2.0

0

16

2
```

Relational Operators

They are used to compare the variables and output the relation. It is also called a comparison operator. This type of operator returns a Boolean value (*True/False*). There are various types of relational operators.

Greater than (>) Operator

It returns true if the left operand is greater than the right. Example, (a>b).

Greater than or equal to (>=) Operator

It returns true if the left operand is greater than or equal to the right. Example, (a>=b).

Less than (<) Operator

It returns true if the left operand is less than the right. Example, (a<b).

Less than or equal to (<=) Operator

It returns true if the left operand is less than or equal to the right. Example, (a<=b).

Equal to (==) Operator

It returns true if both operands are equal. Example, (a==b).

Not equal to (! =) Operator

It returns true if operands are not equal. Example, (a != b).

Assignment Operator

It assigns the values for a variable. For example, to assign a value of 50 to the variable say **a** simple command is: **a = 50**. The assignment operator used in conjunction with other operators serves different purposes, such as **x += 10**, adds 10 to the variable, and later assigs the same. It is equivalent to **x = x + 10**. It is also called a shortcut operator. There are various types of assignment and shortcut operators.

Assignment (=) Operator

This operator assigns values from right-side operands to left-side operands. For example, **z = x + y** assigns **x + y** into **z**.

Add AND (+=) Operator

This operator adds the right operand to the left operand and assigns the result to the left operand. For example, **z += x** is equivalent to **z = z + x**.

Subtract AND (-=) Operator

This operator subtracts the right operand from the left operand and assigns the result to the left operand. For example, **z -= x** equals **z = z – x**.

Multiply AND (=) Operator*

This operator multiplies the right operand with the left operand and assigns the result to the left operand. For example, **z *= x** is equivalent to **z = z * x**.

Divide AND (/=) Operator

This operator divides the left operand with the right operand and assigns the result to the left operand. For example, **z /= x** is equivalent to **z = z / x**.

Modulus AND (%=) Operator

This operator takes modulus using two operands and assigns the result to the left operand. For example, **z %= x** is equivalent to **z = z % x**.

*Exponent AND (**=) Operator*

This operator performs exponential (power) calculation on operators and assigns value to the left operand. For example, **z **= x** is equivalent to **z = z ** x**.

Floor Division (//=) Operator

This operator performs floor division on operators and assigns value to the left operand. For example, **z //= x** is equivalent to **z = z // x**.

Example:

```
1.  x = 10
2.  y = 20
3.  #value modification
4.  val= x + y
5.  print ( " val : ", z)
6.  val2 += x
7.  print ( " val2 : ", z)
8.  val3 *= x
9.  print ( "val3: ", z)
10. val4/= x
11. print ( "val4 : ", z)
12. num= 2
13. num %= x
14. print ( "num:  ", z)
15. num **= x
16. print ( "num:  ", z)
17. num //= x
18. print ( "num: ", z)
```

Output:

```
val: 30

val2:40

val3:400

val4:40.0

num:2

num:1024

num:02
```

Unary Operators

It is used in Python and performed on a single operand. The two unary operators are: **unary (+)** and **unary (-).** Let us take an example for understanding unary operators.

Example:

```
1.    x = 1
2.    print ("Unary positive operator", +x)
3.    print ("Unary Negative operator", -x)
```

Output:

```
Unary positive operator 1

Unary Negative operator -1
```

Bitwise Operators

The Bitwise operator operates bit by bit. Suppose **x = 26** and **y = 18**; then the binary conversion of x and y is 11010, 10010. With their results, let us understand the various bitwise operations on **x** and **y**.

There are the following Bitwise operators:

Binary AND (&) Operator

This operator copies a bit of the result if it exists in both operands.

Binary OR (|) Operator

This operator copies a bit if it exists in either operand.

Binary XOR (^) Operator

This operator copies the bit if it is set in one operand but not both.

Binary One's Complement (~) Operator

It is unary and has the effect of 'flipping' bits.

Binary Left Shift (<<) Operator

In this operator, the left operand's value is moved left by the number of bits specified by the right operand.

Binary Right Shift (>>) Operator

In this operator, the value of the left operand is moved right by the number of bits specified by the right operand. Let us take an example for understanding all bitwise operators.

Example:

```
x = 26
y = 18
z = 0
z = x & y,
print ( "Ans for '&': ", z)
z = x | y,
print ( " Ans for '|':", z)
z = x ^ y,
print ( " Ans for '^':", z)
z = ~x,
print ( " Ans for '~':", z)
z = x << 2,
print ( "Ans for '<<':", z)
z = x >> 2,
print ( " Ans for '>>':", z)
```

Output:

```
Ans for '&': 18
Ans for '|': 26
Ans for '^': 8
Ans for '~': -27
Ans for '<<": 104
Ans for '>>':, 6
```

Logical Operators

Logical operators are used to perform arithmetic and logical computations. These are the special symbols **(and, or, not)** for accomplishing the operations. These operators return True/False as a result.

Logical AND Operator

This operator returns true when both conditions become true.

Logical OR Operator

This operator returns true when two operands are non-zero.

Logical NOT Operator

This operator returns the reverse result, *i.e* the true condition becomes false and the false condition becomes true.

Example:

```
1.  x = 70
2.  print (x > 30 and x < 100)
3.  print (x > 30 or x < 40)
4.  print (not(x > 30 and x < 100))  # returns False because not is used to reverse the result
```

Output:

```
True

True

False
```

Membership Operators

A membership operator tests if a sequence is presented in an object. Here an object may be strings, lists, or tuples. There are two types of membership operators ***in*** and ***not***. Let us take an example of using these operators.

Operator (in)

The operator returns *True* if the variable sought is present in the sequence and *False* otherwise.

Operator (not)

The operator returns *True* if the variable sought is ***NOT*** present in the sequence and *False* otherwise.

Example:

```
x= 10
y = 20
list = [10, 12, 23, 44, 35 ],

if ( x in list ):
   print ( " x is available in the given list")
else:
   print ( " x is not available in the given list")

if ( y not in list ):
   print ( "y is not available in the given list")
else:
   print ( " y is available in the given list")

x = 2
if ( x in list ):
   print ( " x is available in the given list")
else:
    print ( " x is not available in the given list")
```

Output:

```
x is available in the given list

y is not available in the given list

x is not available in the given list
```

Identity Operators

This operator is used to compare the objects, not if they are equal, but if they are the same object, with the exact memory location. There are two types of identity operators, namely, ***is*** and ***is not***. Let us take an example of using these operators.

Operator (is)

This operator returns true if both variables are the same object.

Operator (is not)

This operator returns true if both variables are not the same object.

Example:

```
x = ["James", "Sam"]
y = ["James", "Sam"]
z = x
print (x is z) # returns True because z is the same object as x
print (x is y) # returns False because x is not the same object as y, even if they have the same content
print (x != y) # This comparison returns False because x is equal to y
```

Output:

```
True
False
False
```

Operators' Precedence and Associativity

Operator precedence means which operator should be executed first. It is based on the priority of an operator. If two or more operators have the same precedence, it will check an operator's associativity. Associativity means operating the value from left to right and right to left. Suppose an expression is **2*4+6**; first, **2*4** will be executed because * has higher precedence between the * and + operators.

Suppose an expression is **5/2*3**, then the operator's priority is the same, then it will check the associativity of an operator. Here associativity of / and * is left to right, thus 5/2 is evaluated first. Table **1** enumerates the various operators precedence-wise along with their associativity.

Table 1. Operators description and its associativity.

Operator	Description	Associativity
()	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right

(Table 1) cont.....

Operator	Description	Associativity
is, is not **in, not in**	Identity Membership operators	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
not	Logical NOT	right-to-left
and	Logical AND	left-to-right
or	Logical OR	left-to-right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

EXPRESSIONS

Expression is a representation of value. Moreover, it combines values, variables, operators, and calls to functions. Expressions need to be evaluated. If you ask Python to print an expression, the interpreter evaluates it and displays the desired result. It is very different from the statements. A simple statement in Python is used for value assignment but it is not evaluated, whereas expressions are more logical content that is used in the evaluation. Python expressions only contain three things, Identifiers, Literals, and Operators.

Identifiers

It can be any name that is used to define a class, function, variable module, or object.

Literals

It can be string literals, byte literals, integer literals, floating-point literals, and imaginary literals.

Operators

Based on the token or symbol, it can evaluate an operation. There are various operators, as discussed above, and their corresponding tokens can be used to implement any expression.

STRING OPERATIONS

A string is an essential concept in Python because it can be used in coding in many places. The string can be created with either single or double quotation marks. Let us take an example of string creation.

S1= 'Python Programming.'

S2= "Python Programming."

Both are the way to create a string.

Accessing Values in Strings

An assignment is much easier than another programming language because character types are NOT supported in Python. Characters are considered strings of length one. If we want to take a subpart of a string, then we have to take a substring. The substring is determined by specifying the slice required using index/indices within the square brackets. A string can also update and reassign the value to another string. Let us take an example.

Example:

```
1.  wr1= 'Book for Beginners
2. wr2= 'Requires"
3. print ("wr1[0]: ", wr1[0])
4. print ("wr2[0:6]: ", wr2[0:6])
5. print ("Updated :- ", str2[:9] + 'Book')
```

Output:

```
wr1[0]:  B

wr2[0:5]:  Requir

Updated  :-  Requires Book
```

There are various types of special symbols used in the string:

- **Concatenation (+)** – It adds values on either side of the operator.
- **Repetition (*)**- It creates a new string, concatenating multiple copies of the same string.
- **Slice ([])** – It gives the character from the given index.
- **Range Slice ([:])**- It gives the characters from the given range.

Various types of formatted string operators decide the formatting of string. The different format specifiers in Python are specified by adding the '%' operator symbol before the required format. % symbol denotes the placeholder to insert the variable in the specified format.

%c character, %s string conversion, %d signed decimal integer, %u unsigned decimal integer, %o octal integer, %x hexadecimal integer (lowercase letters), %e exponential notation (with lowercase 'e'), %f floating point real number, %g short for %f and %e, %G the short for %f and %E.

Example:

```
1. print ("My name is %s and my student ID is %d." % (Bateron, 025))
```

Output:

```
My name is Bateron and my student ID  is 025.
```

Triple Quotes

It is used for writing a string in multiple lines. It can be written as triple quotes to three consecutive single or double quotes.

Example:

```
1.  Longstring = """ This is a long string example and that is made up of
2.  several lines and non-printable characters such as
3.  TAB ( \t ) and they will show up that way when displayed.
4.  NEWLINEs within the string, whether explicitly given like
5.  this within the brackets [ \n ], or just a NEWLINE within
6.  the variable assignment will also show up.
7.  """
8.  print Longstring
```

Output:

```
This is a long string example and that is made up of

several lines and non-printable characters such as

TAB (	) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [

], or just a NEWLINE within

the variable assignment will also show up.
```

Various types of built-in functions manipulate strings:

- *capitalize()*- It is used to put the first letter of the string in upper case.
- *center(width, fill char)*- It is used to pad the original string cantered to the total column width.
- *count(str, beg= 0,end=len(string))*-To count the number of times a specific word/sequence of words occurs within a string we use this function. It requires us to specify the beginning and the ending index of the string we need to search in.
- *decode(encoding='UTF-8',errors='strict')*- To return the decoded string we use this function.
- *encode(encoding='UTF-8',errors='strict')*- It returns the encoded string. By default, it raises a Value Error, but we can specify 'ignore' or 'replace'.
- *endswith(suffix, beg=0, end=len(string))*- It is used to check whether the given string/ substring ends with the specified suffix. It returns True if the condition is satisfied otherwise False.
- *expandtabs (tabsize=8)*: It is used to expand the scope of tabs to the specified tab size. By default, the tab size is 8.
- *find (str, beg=0 end=len(string))*- It is used to check if the string/ substring is present in the given string within the specified beginning and ending indices.
- *isalnum()* - It is used to check whether the string is composed of all alphanumeric characters or not. It outputs True if the string has one character and other alphanumeric characters, otherwise it returns False.
- *isalpha()*- It is used to determine whether the sting is composed of characters and alphabets.
- *isdigit()* – It returns True if the string contains only digits and False otherwise.
- *istitle()*- It returns True if the string is properly "title case" and False otherwise.
- *maketrans()*- It returns a translation table for the use in translate function.

- *max(str)*- It returns the max alphabetical character from the string.
- *min(str)*- It returns the min alphabetical character from the string str.
- *replace(old, new [, max])*- It is used to replace old string contents with the new or at most max occurrences specified within max.
- *rfind(str, beg=0,end=len(string))* – To search the string from backward we use this function.
- *rstrip()*- It strips off the trailing whitespaces.
- *split(str=", num=string.count(str))*- It is used to split the string based on the delimiter and return the substring that is split into at most *num* substrings.
- *splitlines(num=string.count('\n'))*- It splits a string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
- *startswith(str, beg=0,end=len(string))*-It determines if a string or a substring of string (if starting index beg and ending index end are given) starts with substring str, returns true if so and false otherwise.
- *strip([chars])*-It is used to perform both lstrip() and rstrip() on string.
- *swapcase()*- It converts the case of all letters.
- *string.title()*- It returns the "titlecased" version of the string, that is, all words begin with uppercase, and the rest are lowercase.
- *upper()*- It is used to convert lowercase letters to uppercase.
- Let us take an example to understand all the above built-in functions.

Example:

```
1.  #!/usr/bin/python
2.  text = "This is a long string example and that is made up of several lines and character
        s"
3.  print ("text.capitalize() : ", text.capitalize())
4.  print ("text.center(40, 'a') : ", text.center(40, 'a'))
5.  sub = "i";
6.  print ("text.count(sub, 4, 40) : ", text.count(sub, 4, 40))
7.  sub = "wow";
8.  print ("text.count(sub) : ", text.count(sub))
9.  text1 = "An operator, is a concept in Python ";
10. text2 = "Python";
11. print (text1.find(text2))
12. print (text1.find(text2, 10))
13. print (text1.find(text2, 40))
14. print (text1.index(text2))
15. text = "this2009";  # No space in this texting
16. print (text.isalnum())
17. text = "An operator, is a concept in Python";
18. print (text.isalnum())
19. text = "this";  # No space & digit in this texting
20. print (text.isalpha())
21. text = "An operator, is a concept in Python";
22. print (text.isalpha())
23. text = "123456";  # Only digit in this texting
24. print (text.isdigit())
25. text = "An operator, is a concept in Python";
26. print (text.isdigit())
27. text = "An operator, is a concept in Python";
28. print (text.islower())
29. text = "An operator, is a concept in Python";
30. print (text.islower())
31. text = u"this2009";
32. print (text.isnumeric())
33. text = u"23443434";
34. print (text.isnumeric())
35. text = "      ";
36. print (text.isspace())
37. text = "An operator, is a concept in Python";
38. print (text.isspace())
39.
40. text = "Hello, welcome to the world of Python";
41. print (text.istitle())
42. print (text.isupper())
43.
44. print (text.isupper())
45. s = "-";
46. seq = ("a", "b", "c"); # This is sequence of textings.
47. print (s.join( seq ))
```

Output:

```
text.capitalize() :  This is a long string example and that is made up of several lines and characters

text.center(40, 'a') :  This is a long string example and that is made up of several lines and characters

text.count(sub, 4, 40) :  3

text.count(sub) :  0

29

29

-1

29

True

False

True

False

True

False

False

False

False

True

True

False

False

False

False

a-b-c
```

CONCLUDING REMARKS

In this chapter, we covered a variety of different data types as well as operations. In addition, we spoke about the precedence of operators and their associativity when evaluating expressions—an explanation of the many principles involved in producing string operations, lists, and dictionaries. The next topic that will be covered is control flow statements, which are utilized in Python. This will be covered in the following chapter.

CHAPTER 3

Control Flow

Abstract: It's now the time for the readers to acquaint themselves with the control flow in the programming. So far the users have seen the linear path of execution, where the execution commences from the top and moves sequentially to the bottom. Further, our world is filled with tasks required to run in loops, for example in banking applications, unless the user enters the correct password the system prompts the dialogue "Please enter the correct password". In this chapter, we introduce the concept of control flow through which the users can decide the execution path for the program and the looping constructs to iterate through the tasks. Our key takeaways from this chapter are listed below:

1. Understanding the decision control statements.
2. Programming with *if-else* ladder and its variants.
3. A hand on *for*, and *while* loop statements.
4. Understanding the control flow alterations the *jump* statements like *break*, *continue* and *pass*.

Keywords: Control flow, Conditional processing, Looping constructs.

INTRODUCTION

The control flow of a program is illustrated by a statement known as a control flow statement. In addition, it determines the sequence in which the code of the program is executed. Conditional statements, loops, and function calls are the primary mechanisms that direct how a Python program executes its instructions. Python programming utilizes three distinct types of control structures in its many applications.

1. Sequential statements
2. Decision control statements
3. Looping statements

SEQUENTIAL STATEMENTS

It uses a default mode because the control will move line by line in a program. Moreover, it is a series of statements that are executed in a sequence.

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

Example:

```
# Python Program to find the area of a triangle
a = 5
b = 6
c = 7
# calculate the semi-perimeter
s = (a + b + c) / 2
# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print ('The area of the triangle is %0.2f' %area)
```

Output:

```
The area of the triangle is 14.70
```

DECISION CONTROL STRUCTURES

There are a few other names for a decision control statement, including a selection control statement and a branching statement. The condition serves as the foundation for the selection statements. If a condition is met, then the statement will be carried out. In addition to that, we utilize it as a checking and testing mechanism. There are wide distinct varieties of decision control statements, including the following:

- *if*
- *if-else*
- *nested if*
- *if-elif-else*

If Statements

If statements are utilized in programming to determine whether or not a particular section of code should be executed. If a condition is met, the action will be successful; otherwise, it will not.

Syntax:

```
if condition:
# Statements to execute if
# Condition is true
```

The flow diagram for the if statement is shown in Fig. (**1**).

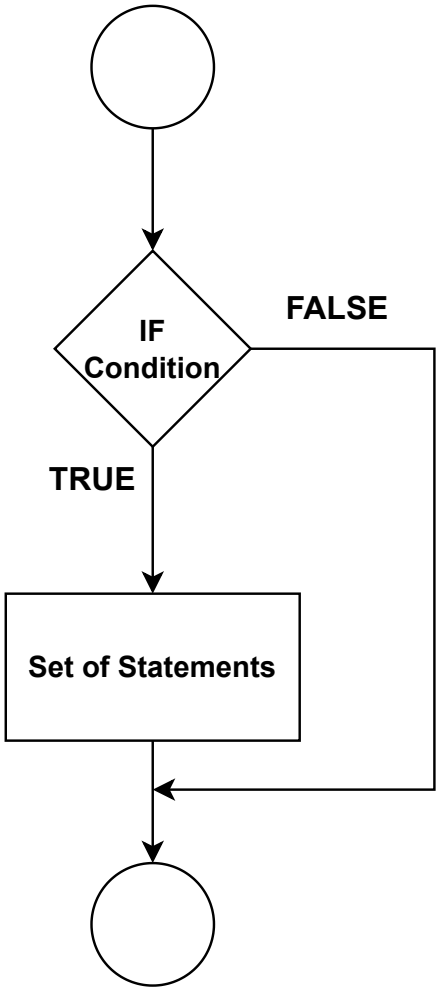


Fig. (1). Flow diagram of if statement.

Example:

```
1.  # Program to check whether a number is positive or Negative
2.  n = 10
3.  if n > 0:
4.      print (n, "is a positive number.")
5.  n1 = -2
6.  if n1<0:
7.      print (n1, "is a Negative number.")
```

Output:

```
10 is a positive number.
-2 is a Negative number.
```

If-else Statements

The **if-else** statement is used to execute a specific code based on the results of previous statements known as if and else. If a condition is met, the code included within an if block will be carried out. If a condition is not met, the code contained within an else block will be carried out.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

The flow diagram for the **if-else** expression may be found in Fig. (**2**).

Example:

```
1.  # Program to check whether the number is positive or negative
2.  n = 10
3.  if n >= 0:
4.      print ("Positive or Zero")
5.  else:
6.      print ("Negative number")
```

Output:

```
Positive or Zero
less than 1000
```

Nested if

When you have an if statement inside of another if statement, you have a nested if

statement. The flow pattern of nested if statements may be seen in Fig. (**3**).

Syntax:

```
if (condition1):
  # Executes when condition1 is true
  if (condition2):
    # Executes when condition2 is true
  # if block is end here
# if block is end here
```

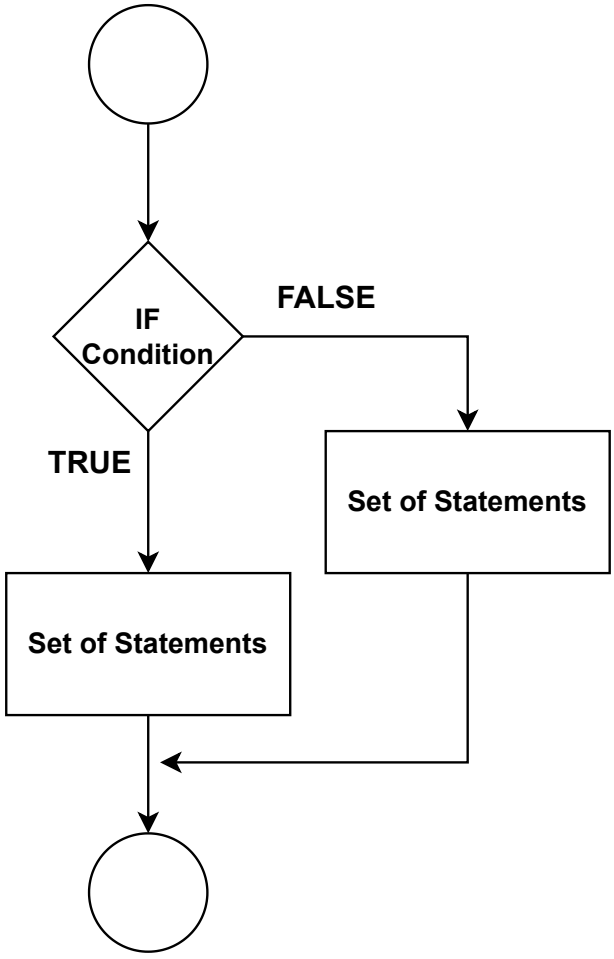


Fig. (2). Flow diagram of *if-else* statement.

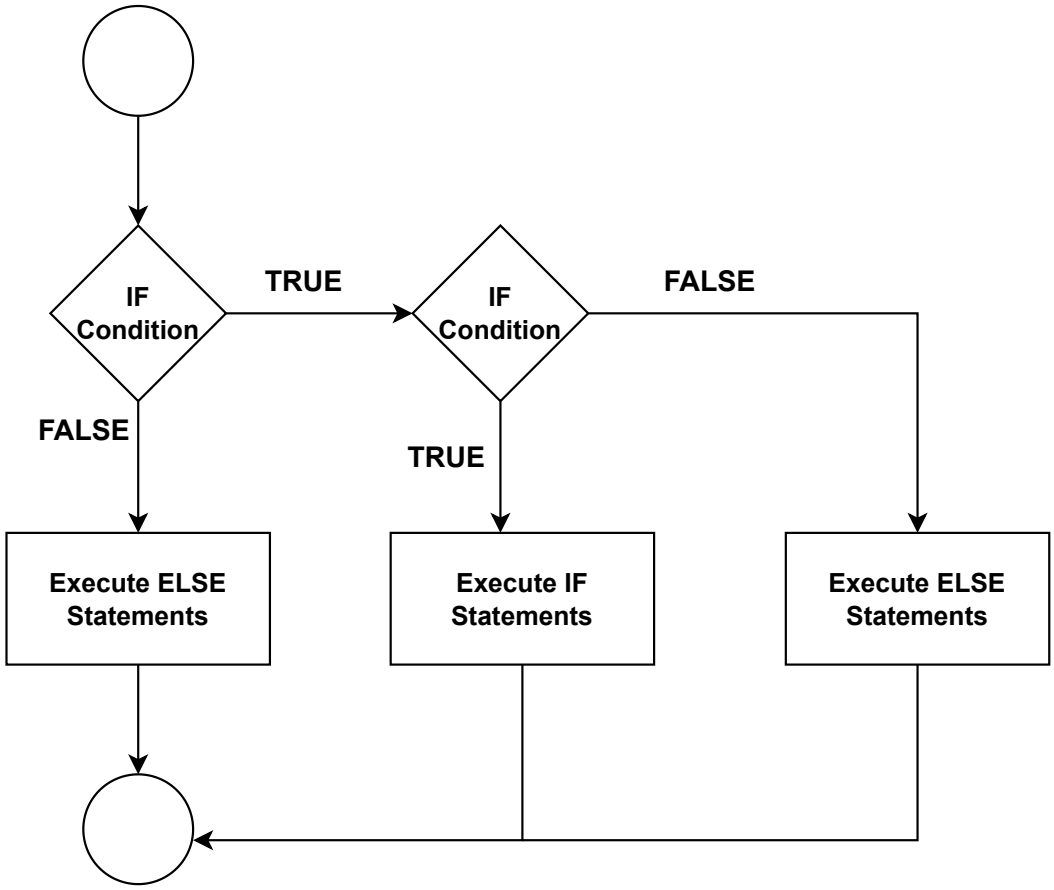


Fig. (3). Flow diagram of nested if statement.

Example:

```
n = 10
if n >= 0:
    if n == 0:
        print("Zero")
    else:
        print ("Positive number")
else:
    print ("Negative number")
```

Output:

```
Positive number
```

if-elif-else

The if-elif-else statement is used to execute a statement or a block of statements conditionally. Fig. (**4**) shows the flow diagram of the if-elif-else statement.

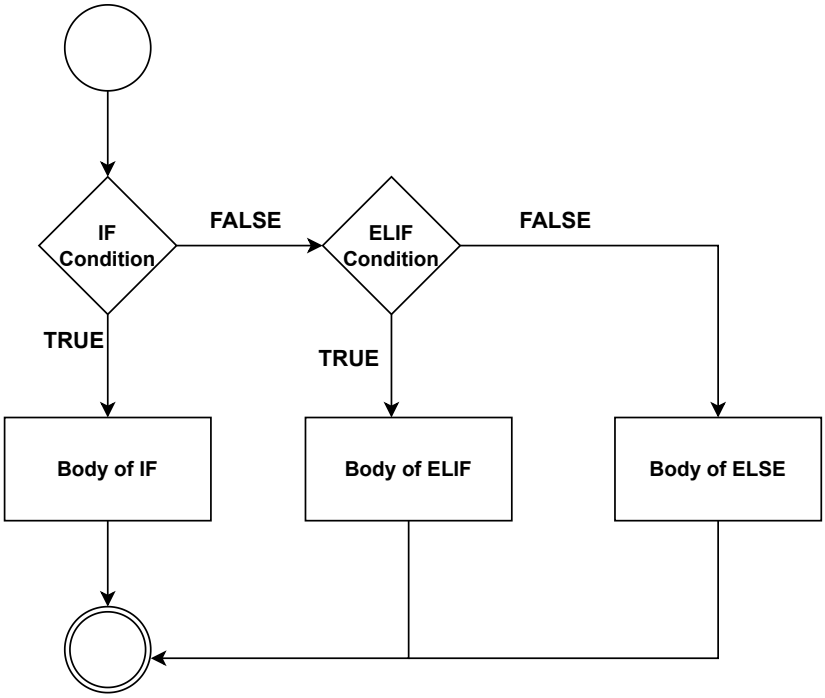


Fig. (4). Flow diagram of if-elif-else statement.

Syntax:

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

Example:

```
1.  n = 200
2.  if n > 0:
3.      print ("Positive number")
4.  elif n == 0:
5.      print ("Zero")
6.  else:
7.      print ("Negative number")
```

Output:

```
Positive number
```

LOOPING STATEMENTS

Looping, which is just repeating the same code section many times, requires its utilization. This control structure will be utilized whenever there is a need to repeat a set of instructions based on the presence of a particular condition. Python supports two distinct iteration styles for its loops.

- *for loop*
- *while loop*

For Loop

Repeating over a sequence that is either a list, tuple, dictionary, or set may be accomplished with the help of a for loop. It is very important to remember that the ***in*** keyword is a component of the syntax for the for statement, even though it has no practical connection to the in operator used for membership checking. The flow diagram of the loop is shown in Fig. (**5**).

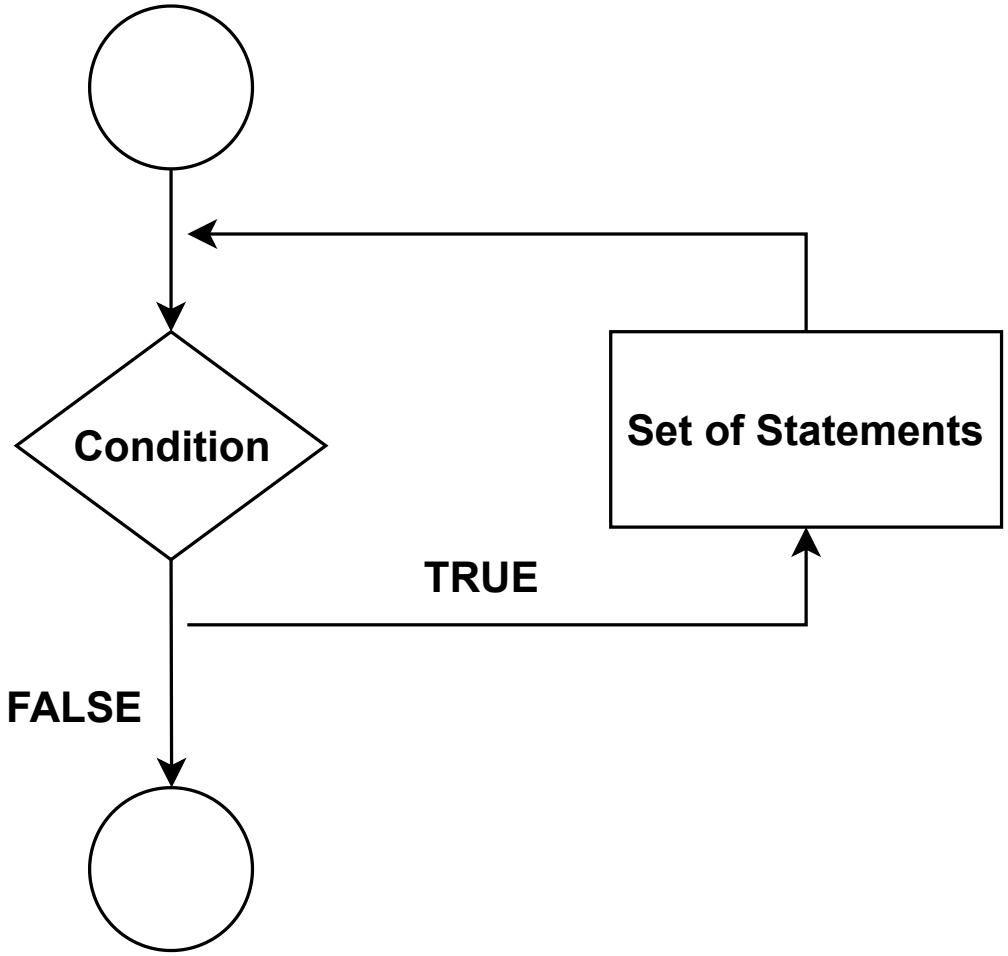


Fig. (5). Flow diagram of for loop.

Syntax:

```
for var in sequence:
    statements(s)
```

Example 1:

```
1.  # Python program to repeat over range 0 to n-1
2.   n = 10
3.  for i in range(0, n):
4.      print (i)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Example 2:

```
1.  list = [1,2,3,4,5,6,7,8,9,10]
2.  n = 5
3.  for i in list:
4.      c = n*i
5.      print (c)
```

Output:

```
5
10
15
20
25
30
35
40
45
50
```

Example 3:

```
1.  for x in range(5):
2.      print(x)
3.
4.  for x in range(3, 6):
5.      print(x)
6.
7.  for x in range(3, 8, 2):
8.      print(x)
```

Output:

```
0
1
2
3
4
3
4
5
3
5
7
```

While Loop

It repeats the statement until a given condition is satisfied. If the condition is true, the while loop will work; otherwise, it will not. The flow diagram of the while loop is shown in Fig. (**6**).

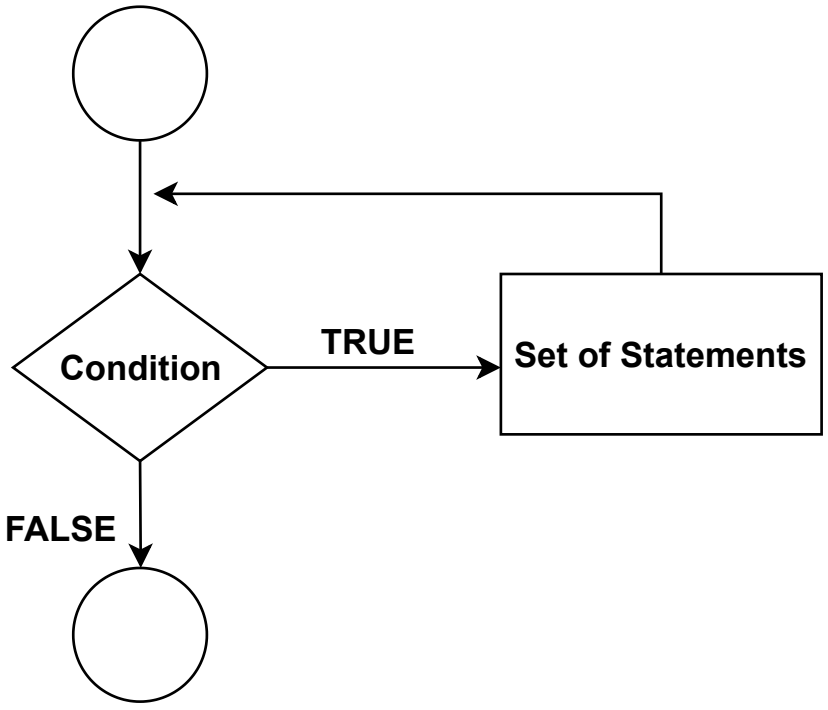


Fig. (6). Flow diagram of while loop.

Syntax:

```
while expression:
   statement(s)
```

Example:

```
1.  cnt = 0
2.  while (cnt < 10):
3.    print ("The count is:", cnt)
4.    cnt = cnt + 1
```

Output:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
The count is: 9
```

Nested Loop

A nested loop is a loop that is contained within another loop. We have the option of nesting a for loop inside a while loop, a while loop within of a for loop, a for loop inside of a for loop, or a while loop inside a while loop. To better grasp the nested loop, let us look at an example.

Syntax:

```
for var in sequence:
   for var in sequence:
      statements(s)
      statements(s)
```

The following is an example of the Python programming language's syntax for the statement that creates a nested while loop:

```
while expression:
   while expression:
      statement(s)
      statement(s)
```

Example 1:

```
# This program uses a nested for loop
i = 2
while(i < 20):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) :
      print (i, " is prime")
   i = i + 1
```

Output:

```
2  is prime
3  is prime
5  is prime
7  is prime
11  is prime
13  is prime
17  is prime
19  is prime
```

Example 2:

```
#program to print a pattern
for i in range (1, 5):
  for j in range(i):
    print (j + 1, end = ' ')
  print ()
```

Output:

```
1
1 2
1 2 3
1 2 3 4
```

Example 3:

```
#program to print a pattern
for i in range (1, 5):
  for j in range(i):
    print (j + 1, end = ' ')
  print ()
```

Output:

```
1
1 2
1 2 3
1 2 3 4
```

Break Statement

With the break statement, you can end a loop. When the break statement is run, the control will be moved out of the loop. Fig. (**7**) shows the flow chart for the break statement. If the break statement is inside a nested loop, it will end the loop that is closest to it.

Example:

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple
num_sum = 0
count = 0
for x in numbers:
    num_sum = num_sum + x
    count = count + 1
    if count == 6:
        break
print ("Sum of first ",count,"integers is: ", num_sum)
```

Output:

```
Sum of first  6 integers is:  21
```

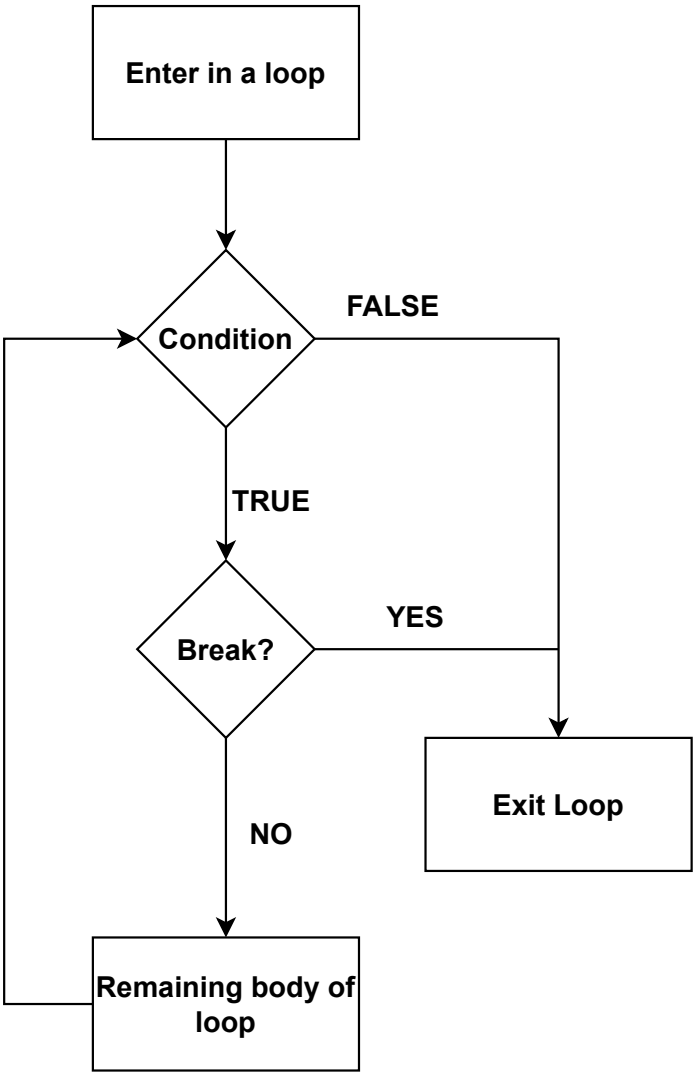


Fig. (7). Flow diagram of break statement.

Continue Statement

It is utilized to bypass the remainder of the code included within a loop for the sole purpose of the current iteration. When a continue statement is present in a loop, the loop does not exit but continues with the subsequent iteration. Fig. (**8**) depicts the flow diagram of the continue statement in its many iterations.

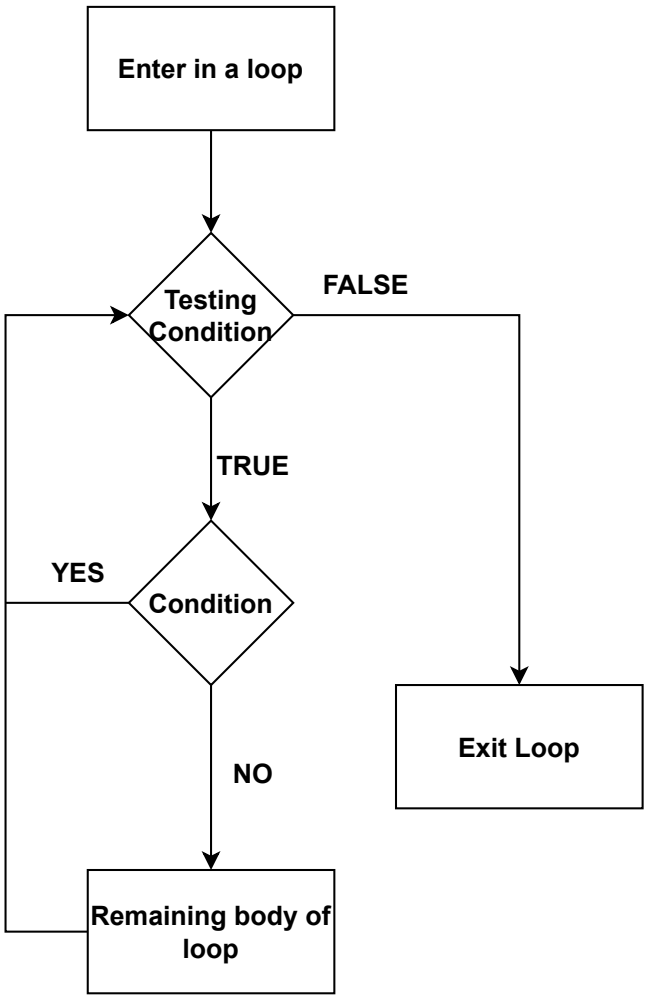


Fig. (8). Flow diagram of continue statement.

Example 1:

```
1.  for i in range(10):
2.      if i==5:
3.          continue
4.      print (i)
```

Output:

```
0
1
2
3
4
6
7
8
9
```

Example 2:

```
1.  for x in range(7):
2.      if (x == 4 or x==6):
3.          continue
4.      print (x)
```

Output:

```
0
1
2
3
5
```

The Pass Statement

In Python, the term "null statement" refers to the *pass* statement. The most significant distinction between a comment and a pass statement is that the interpreter disregards the former, while the latter is not thus treated. In addition, when the pass is put into action, absolutely nothing takes place. It does not affect the operation. To better grasp the pass statement, let us look at an example.

Example 1:

```
1.  a = 10
2.  b = 20
3.  if b > a:
4.    pass
```

Output:

```
No output
```

Example 2:

```
1. sequence = {'a', 'b', 'c', 'd', 'e', 'f'}
2. for x in sequence:
3.     pass
4.
```

Output:

```
No output
```

CONCLUDING REMARKS

This chapter covered several distinct categories of control statements, including sequential control structures, selection control structures, and repetition control structures. It provides an in-depth analysis of each control statement. We enunciate the functioning of the loops. As a result, this chapter features each control statement with an appropriate illustration and graphic. The next topic will be the usage of functions in Python, which will be covered in the following chapter.

CHAPTER 4

Functions

Abstract: Functions are one of the primary concepts in every programming language. They provide an easy way to package the programming logic and use it as and when required as many times at any place. Thus they help to reduce the redundancy in code and increase the reproducibility. With the increase in the length of code, it is often a good idea to divide the code into separate modules by splitting the code into different functions based on its utility. This is a much- sought practice to organise the lengthy code. They also help in unit testing the code as testing small units in isolation is quite an easy task. This deliberate use of functions thus supports language flexibility and a user-friendly interface. In this chapter the pertinent takeaways would be:

1. Understanding *functions* and *function calls*.
2. Comprehending the concept of *local* and *global* variables.
3. Programming with the recursive functions.

Keywords: Reproducibility, Redundancy, Reusability, Recurison.

INTRODUCTION

The code may be organized, made more understandable, reused, and repurposed with the use of functions, which are a helpful approach to separate the code into more manageable portions. In addition, a function is a piece of code that is put into action when the function itself is invoked. It can take in data in the form of a parameter or argument and then return the result.

Definition

A function is an area of code that comprises a block of statements that carry out a particular operation. When creating a function, there are a few fundamental guidelines to follow:

1. Function blocks start with the **def** keyword. After that, the **function name** and parentheses (**()**) are used.
2. An argument or parameters should be passed inside the parentheses.
3. Any function's code block begins with a colon (:) and is indented.

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

Syntax:

```
def function_name(parameters):
  Statement 1
  Statement 2
  Statement 3
        -
        -
        -
   Statement n
  return [expression]
```

There are two types of functions, as described below:

- Predefined functions
- User-defined functions

PREDEFINED FUNCTIONS

It is sometimes referred to as a built-in function because Python has its functionality established in the language. The Python interpreter has several built-in functions that are always available for usage. There are many kinds of built-in functions, and here is an example of some of those built-in functions (Table **1**):

Table 1. Predefined functions.

abs()	It returns the absolute value of a number
bin()	It returns the binary version of a number
float()	It returns a floating-point number
hex()	It converts a number into a hexadecimal value
int()	It returns an integer number
len()	It returns the length of an object
list()	It returns a list
max()	It returns the largest item in an iterable
min()	It returns the smallest item in an iterable
oct()	It converts a number into an octal
pow()	It returns the value of x to the power of y
print()	It prints to the standard output device
range()	It returns a sequence of numbers, starting from 0 and increments by 1 (by default)
round()	It rounds a numbers

Let's look at an example of each of the built-in functions discussed so far to better grasp how they should be used.

Example:

```
#Return the absolute value of a number
x1 = abs(-5.37)
print (x1)

# Return the binary version of 24
x2 = bin(24)
print (x2)

#Convert the number 7 into a floating-point number
x3 = float(7)
print (x3)

#Convert 321 into hexadecimal value
x4 = hex(321)
print (x4)

#Convert the number 7.2 into an integer
x5 = int(7.2)
print (x5)

#Return the number of items in a list
list1 = ["Malika", "Krishna", "William", "Sam"]
x6 = len(list1)
print (x6)

#Return the largest number
x8 = max(245, 433)
print (x8)

#Return the lowest number
x9 = min(343, 434)
print (x9)

#Convert the number 12 into an octal value
x10 = oct(32)
print (x10)

x11 = pow(5, 5) #Return the value of 5 to the power of 5 (same as 5 * 5 * 5 * 5 * 5)
print (x11)

#Print a message onto the screen
Print ("It is a message")

#Create a sequence of numbers from 0 to 7, and print each item in the sequence
x12 = range(8)
for n in x12:
  print (n)

#Round a number to only two decimals
x13 = round(6.66543, 2)
print (x13)
```

Output:

```
5.37
0b11000
7.0
0x141
7
4
433
343
0o40
3125
It is a message
0
1
2
3
4
5
6
7
6.67
```

USER-DEFINED FUNCTIONS

A user-defined function is what's known as a user-defined function when it's a function that's been written for a specific job. In addition, the user is the one who designed or wrote the function themselves. Let's look at an example of a user-defined function so that we can have a better grasp of how this process works.

Example 1:

```
1.  # Declaring a show function
2.  def show():
3.      print ("Inside function")
4.
5.  # Calling function
6.  show()
```

Output:

```
Inside function
```

Example 2:

```
1.  # Python function to check even and odd
2.  def evenOdd(var):
3.      if (var%2 == 0):
4.          print ("Number is Even")
5.      else:
6.          print ("Number is Odd")
7.
8.  # Calling function
9.  evenOdd(10)
10. evenOdd(11)
```

Output:

```
Number is Even

Number is Odd
```

Function Call

We may call a function to perform a task once it has been specified. For calling a function, we type the name of the function and the necessary parameters.

Syntax:

```
function_name (argument_1, argument_2)
```

Example 1:

```
1.  # Function Definition
2.  def my_function():
3.    print ("This function has been created")
4.
5.  my_function()  # Function Calling
```

Output:

```
This function has been created
```

Example 2:

```
# Function Definition
def my_name(name):
    print (name)

my_name('David')  # Function Calling with argument
```

Output:

```
David
```

Example 3:

```
# Function for the sum of three numbers
def sum_three_numbers(num1, num2, num3):
    return num1 + num2 + num3

# Function calling with three arguments
x= sum_three_numbers (1,2,3)
print (x)
```

Output:

```
6
```

Function Parameters and Arguments

A piece of information can be passed into a function based on the arguments and parameters that the function accepts. The precise values sent into a function are referred to as **arguments**. We may provide several arguments, but a comma must break each up. The phrases “parameter” and “argument” are synonymous and can be used in place of one another. In addition, data goes *via* a function to be processed. There is a distinction, although a subtle one, that can be made between function parameters and arguments. Parameter refers to the variable listed within the parenthesis in the function description. On the other hand, “argument” refers to a value that is supplied to the function when it is called. To assist you in comprehending the significance of function arguments and parameters, let's look at an illustration.

Example:

```
1.  # parameters example
2.  def Name_function(F_name, L_name):  # Function Definition
3.    print (F_name + " " + L_name)
4.
5.   # Two arguments value (Krishna, Kumar)
6.  Name_function("Krishna", "Kumar")  # Function Calling
```

Output:

```
Krishna Kumar
```

Default Arguments

In Python, default values for function arguments are allowed. The assignment operator (=) can be used to provide a default value for an argument. Consider the following scenario.

Example:

```
1.    def my_name(name, msg="how are you?"):   # Function Definition
2.        print ("Hello", name + ', ' + msg)
3.
4.    my_name("John")  # Function Calling
5.    my_name("James", "Welcome")   # Function Calling
```

Output:

```
Hello John, how are you?

Hello James, Welcome
```

Let's look at some other examples related to function.

Example:

```
#Python Arbitrary Arguments
def some_name(*names):
        print("Hello", names)

some_name("Jay", "Joy", "Ram", "Amit")

#Arguments passing
def fun1(food):
  for n in food:
    print(n)

var = ["apple", "banana", "cherry"]
fun1 (fruits)
```

Output:

```
Hello ('Jay', 'Joy', 'Ram', 'Amit')
apple
banana
cherry
```

Variable Scope and Lifetime

The variable cannot be accessed from any of the different parts of the program. There is a possibility that some of the variables will not even be present in the entire program. The variable's declaration determines the variable's existence and accessibility.

Scope of the Variable

It is a part of the program in which a variable is accessible. There are two types of the scope of the variable:

- *Local Scope*
- *Global Scope*

Local Scope

A variable is created in the nested function then the scope exists in that function. Suppose the variable is declared in the innermost function; then, the scope of that variable is only in the innermost function.

Global Scope

A global variable is generated in the software's main body and belongs to the global scope. Global variables of this kind are available in any scope. Furthermore, any function, method, or code easily understands the global variable.

The Lifetime of the Variable

It is the duration for which a variable exists. Let's take an example for understanding the variable scope and lifetime concept.

Example:

```
1.  # local scope
2.  def show():
3.    x = 1234  # Local variable
4.    print (x)
5.  show()
6.
7.  x = 1234  # Global Varibale
8.
9.  def show():
10.   print (x)
11. show()
12.
13. print (x)  # Global scope
```

Output:

```
1234

1234

1234
```

Local and Global Variables

Local variables are declared inside the method or block. Local variables are assessible only within the declared method or block and cannot be accessed outside that method or block.

Example:

```
1.  def my_fun():
2.      x=100      #Local variable
3.      print (x)
4.
5.  my_fun()
```

Output:

```
100
```

Global Variables

This type of variable is defined in the main body of a file. It is available throughout the program. All functions or blocks can easily understand and access a global variable.

Example:

```
1.  x=10  # Global
2.  def my_fun():
3.      x=5  # Local scope within my_fun()
4.      print (x) # Print local variable
5.
6.  my_fun()
7.
8.  print (x)  # Print Global variable
```

Output:

```
5

10
```

Global Statement

The block in which the local variable is stated is the only place in which its scope may be evaluated. In addition, a variable is considered local when constructed inside a function by the programmer. However, using the global keyword within a function, we can create a variable accessible across the program. This is the only place where it may be used. To better understand the concept, let's look at an example of a global statement that uses the global keyword.

Example:

```
1. def my_fun():
2.   global x
3.   x = 10  # x is global
4.
5. my_fun()
6.
7. print ( x)  # print x
```

Output:

```
10
```

Suppose we have a variable with the same name as a global variable in the program. In such a case, a new local variable of that name is created, different from the global variable.

Example:

```
1.  x=10
2.  def show():
3.      x=5
4.      print ("In Function x is=",x)
5.  show()
6.  print ("Outside function is=",x)
```

Output:

```
In Function x is= 5

Outside function is= 10
```

Return Statement

It's used to finish the function call's execution and return the result (value or expression). There are a few key points to remember regarding ***return*** statements:

- Return None is the same as a return statement with no arguments.
- The statements following the return statements are skipped.
- The unique value None is returned if there is no expression in the return declaration.
- You can't use a return statement outside of a function.

Syntax:

```
def fun_name():
        function body
        .
         .
         .
        return [expression]
```

Example:

```
1.  # return example
2.  def addition(n1, n2 ):
3.      total = n1+ n2
4.      return total.
5.
6.  total = addition(1, 2 );
7.  print ("Addition is =", total)
```

Output:

```
Addition is = 3
```

Lambda Functions

Lambda functions are used when an anonymous function or a function that is declared without a name exists. In most cases, functions are defined using the def keyword, whereas anonymous functions are defined using the ***lambda*** keyword. The syntax for lambda functions is described below.

The following are essential points about lambda functions:

1. The number of arguments for a lambda function is infinite, but only one expression exists.
2. We use lambda functions if we need an anonymous function for a short time.
3. The meaning of Lambda does not include a "return" statement; instead, it always includes an expression that is returned.

Syntax:

```
lambda arguments: expression
```

Example 1:

```
1.  # Add 50 to the argument and return
2.  res = lambda a : a + 50
3.  print (res(25))
```

Output:

```
75
```

Example 2:

```
1.  # Sum of three numbers
2.  res = lambda n1, n2, n3: n1 + n2 + n3
3.  print (res (1, 2, 3))
```

Output:

```
6
```

Recursive Functions

When we construct a function and then call that function within the definition of the function, we have created a recursive code. Recursion is the name given to the process in which a function calls itself. Programming using recursion is a very effective method; nevertheless, it is important to keep in mind that the recursion must be stopped at some point. It is a sign that the code functions accurately if the ending condition can be found within the recursion. If the termination condition is absent, it indicates that the recursive code will execute an unlimited number of times.

Syntax:

```
def func_name(): <-
            |
            | (recursive call)
            |
   func_name() ----
```

The followings are the advantages and disadvantages of a recursive function:

Advantages

- Using recursion, a function can be broken down into smaller sub-problems.
- Often creating a sequence is more straightforward than nested iteration.
- The use of recursive functions makes the code appear simple and efficient.

Disadvantages

- Recursive calls use a lot of memory and time, which is why they are costly.
- Debugging recursive functions is difficult.
- The logic behind recursion can be challenging to understand at times.

Example:

```
1.  # Program for Fibonacci series up to k terms
2.  def fib(k):     # Recursive function
3.    if k <= 1:
4.      return k
5.    else:
6.      return(fib(k-1) + fib(k-2))
7.
8.  term=8
9.  if term <= 0:
10.   print ("Invalid input ! Please input a positive value")
11. else:
12.   print ("Fibonacci series:")
13.   for i in range(term):
14.     print (fib(i))
```

Output:

```
0
1
1
2
3
5
8
13
```

Example:

```
1.  # Recursive program to compute the factorial value
2.   def fact(k):   # Recursive function
3.    if k == 1:
4.       return k
5.    else:
6.       return k * fact(k-1)
7.
8.  n = 5
9.   # check if the input is valid or not
10. if n < 0:
11.   print ("Invalid input ! Please enter a positive number.")
12. elif n == 0:
13.   print ("Factorial of number 0 is 1")
14. else:
15.   print ("Factorial of number", n, "=", fact(n))
```

Output:

```
Factorial of number 5 = 120
```

Function Redefinition

Python provides a facility for the redefinition of functions because Python is dynamic. Redefinition of function means redefining an already defined function. Let's take an example for understanding the concept of Function redefinition.

Example:

```
1.  # function redefinition example
2.  from time import gmtime, strftime
3.
4.  def show(msg):      # Function definition
5.      print (msg)
6.  show("Start")
7.
8.  def show(msg):      # Function redefinition
9.      print (strftime("%H:%M:%S", gmtime()))
10.     print (msg)
11. show ("Processing.")
```

Output:

```
Start
11:20:20
Processing.
```

CONCLUDING REMARKS

In this chapter, we went through the function and how it is crucial for modular programming to focus on the functions. It is concerned with how it is used to send parameters into a function. We have also gone through the differences between local and global variables, statements, and the scope of each. Therefore, this chapter covers every conceivable variant of the function and the process by which it operates. As we move on to the next topic that Python supports, strings, lists, and the associated features will be covered in the following chapter.

CHAPTER 5

Sequence-String and List

Abstract: Unlike primitive data types like integers, floats and Boolean, a string is an ordered sequence of characters each of which can be accessed easily. Further, one of the important built-in data types of Python is lists. Lists and strings share many similarities like, lists are a sequence of values. List indices work similarly to string indices. But unlike strings lists are mutable. In this chapter, we introduce the core concepts of lists and strings and several operators that are used to make programming with these user-friendly.

Keywords: Mutable, Slicing, Ordered collection.

INTRODUCTION

When we want a collection of characters that are pretty like one another, we have to use a sequence of characters. For instance, if you want to keep a record of your name in the computer's memory, you will need a variable capable of storing your name. However, it is necessary to have a series of characters because the name is a collection of characters. This string consists of nothing more than a succession of characters.

Python's equivalent of the sequence data type is called a list. It is the most effective and may be expressed as a list of values delimited by commas and enclosed in square brackets. A list is frequently utilized to store the sequence of various kinds of data. The list is changeable, which indicates that its elements can be altered after the list has been constructed. This chapter discusses a variety of operations that may be carried out on a list's elements.

STRING

A series of characters is referred to as a string. Python relies heavily on this idea to function correctly. There are a few key aspects to consider about string.

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

- Strings are amongst the most popular types in Python.
- It can create them simply by enclosing characters in quotes (single, double, or triple).
- Python treats single quotes the same as double quotes.
- A string is a sequence of Unicode characters, and a character is simply a symbol.

To generate a list of names, it must type each name surrounded by quotation marks, such as “Krishna”. The string may also be assigned to a variable to carry out additional operations and make further use of that string. Let's start with the most fundamental examples to grasp the idea of string.

Example:

```
1.  str = "This is a string."
2.  print (str)
```

Output:

```
This is a string
```

Python does not have a character data type in its standard library. If we consider a single character to be a string, then the length of that string would be one. You may access the string's constituent components using square brackets ([]).

Example:

```
1.  str = "Python"
2.  print (str[0])
3.  print (str[1])
4.  print (str[2])
5.  print (str[3])
6.  print (str[4])
7.  print (str[5])
```

Output:

```
P
y
t
h
o
n
```

The string can use to loop through character by character.

Example:

```
for i in "Krishna":
  print (i)
```

Output:

```
K
r
i
s
h
n
a
```

Other concepts about string include the *len()* function for getting the length of a string, keyword *in*, and *not in* for checking whether the substring or character is present in a string.

Example:

```
# print a length of a string
str = "Python"
print (len(str))  #String Length

# print "simple" in a string
str1= "Python is so simple"
print ("simple" in str1)

#Print only if "simple" is present in a string:
str2 = "Python is so simple"
if "simple" in str2:
  print ("Yes, 'simple' is present.")

# print "simple" is not in a string
str3 = "Python is so simple"
print ("programming" not in str3)
```

Output:

```
6
True
Yes, 'simple' is present.
True
```

String Concatenation

Combining two strings is what "string concatenation" refers to. If we have two strings placed into two different variables, str1 and str2, then the two strings may be concatenated together or merged into a single string by using one of those two methods. There are four ways to concatenate the strings.

1. Using + Operator
2. Using join() method
3. Using % operator
4. Using format() function

Using + Operator

It is a straightforward way to concatenate two strings. It can add multiple strings together. Let us take an example to understand this concept.

Example:

```
# Defining strings
S1 = "Python"
S2 = "Programming"

# + Operator is used to combine strings
S = S1 + S2
print (S)
```

Output:

```
PythonProgramming
```

Using join() Method

The join() method is a string method that returns a string in which the sequence elements have been joined using an str separator.

Example:

```
fname = "My"
lname = "Name"


print ("".join([fname, lname]))

# join() method string with a separator Space(" ")
name = " ".join([fname, lname])
 print (name)
```

Output:

```
MyName
MyName
```

Using % Operator

It can use a string formatting operator (*%*) for concatenating the strings.

Example:

```
fname = "My"
lname = "Name"

# % operator
print ("% s % s" % (fname, lname))
```

Output:

```
MyName
```

Using Format() Function

This is an illustration of a way to format strings. It is possible to make many substitutes and format the value in multiple ways. *str.format()* is necessary to concatenate the strings. This approach achieves concatenating components within a series by using positional formatting. The following code demonstrates how to use the ***format()*** function to combine the strings that are now stored in the fname and lname variables into a single string that is then saved under a different variable name. It is possible to place strings in a specific sequence using curly

bracaes. The first variable is tracked in the first set of curly braces and second variable is traced in the second set of curly braces.

Example:

```
1.  fname = "Malika"
2.  lname = "Acharya"
3.    print ("{} {}".format(fname, lname))
4.
5.  # result in name
6.  name = "{} {}".format(fname, lname)
7.    print (name)
```

Output:

```
MalikaAcharya
MalikaAcharya
```

Appending Strings

Appending to a string means adding one string to another. Suppose we have two strings, and those strings have been assigned to two variables, **S1** and **S2**; then, it can be appended or added **S1** to **S2**. There are two ways to append one string to another.

1. Using += operator
2. Using join()

Using += Operator

Utilizing a specialized function to complete this one activity is an example of the conventional method used in other languages, although this approach is simpler. To better grasp this idea, let's look at a concrete illustration.

Example:

```
# initializing string
str1 = "String1"
  # initializing add_string
str2 = "String2"
  # printing original string
print ("The original string : " + str(str1))
  # printing original add string
print ("The add string : " + str(str2))

# adding one string to another
str1 += str2          # Using += operator
  # print result
print ("The appended string is : " + str1)
```

Output:

```
The original string: String1
The add string: String2
The appended string is: String1String2
```

Using Join()

To join the strings together, this approach is utilized. The preceding technique has an advantage over this method when we need to concatenate more than just two strings, such as when we have multiple strings to concatenate. To better grasp this idea, let's look at a concrete illustration.

Example:

```
# initializing string
str1 = "Python"

# initializing add_string
str2 = "Book"
 # printing original string
print ("The original string : " + str(str1))
 # printing original add string
print ("The add string : " + str(str2))

# adding one string to another
result = "".join((str1, str2))          # Using join()
  # print result
print ("The appended string is : " + result)
```

Output:

```
The original string : Python
The add string : Book
The appended string is : PythonBook
```

Multiplying Strings

Because string multiplication in Python is analogous to the behavior of another data type, the process is much simplified. The process of multiplying strings can be done in several different ways. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.      str1= "Python Programming"
2.      str2= 2*str1    # Method 1
3.      print (str2)
4.
5.      str3=4*(str1) # Method 2
6.      print (str3)
7.
8.      str4= 3*('Python', 'Programming') # Method 3
9.      print (str4)
```

Output:

```
Python ProgrammingPython Programming
Python ProgrammingPython ProgrammingPython ProgrammingPython Programming
('Python', 'Programming', 'Python', 'Programming', 'Python', 'Programming')
```

Immutable Strings

When a string is immutable, the value of the string cannot be altered in any way. In addition, once we have assigned the string value, an error will be generated if we try to perform this thing, and it cannot reassign itself. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  str= "Book"
2.  print (str)
3.  str[0] = "P"   # Can not reassign
```

Output:

```
str[0] = "V"   # Can not reassign
TypeError: 'str' object does not support item assignment
```

String Formatting Operator

It uses the % operator for string formatting operations. Various symbols can be used along with %, as shown in Table **1**.

Table 1. String formatting symbols and their description.

Symbol	Description
%f	It is for a floating-point real number
%d	It is for a signed decimal integer
%u	It is for an unsigned decimal integer
%o	It is for an octal integer
%X	It is for a hexadecimal integer (Uppercase letters)
%e	It is for an exponential notation (with lowercase 'e')
%E	It is for an exponential notation (with UPPERcase 'E')
%g	It is for the shorter of %f and %e
%G	It is for the shorter of %f and %E
%c	It is for a character
%s	It is for a string conversion *via* str() before formatting
%i	It is for a signed decimal integer
%x	It is for a hexadecimal integer (lowercase letters)

Example:

```
# Initialize the variable as a string
var = '27'
string = "Variable as string = %s" %(var)
print (string)

# Printing as raw data
print ("Variable as raw data = %r" %(var))

# Convert the variable to an integer
# and apply other formatting options
var = int(var)
string = "Variable as integer = %d" %(var)
print (string)
print ("Variable as float = %f" %(var))

print ("Variable as printing with special char = %c Ram" %(var))

print ("Variable as hexadecimal = %x" %(var))
print ("Variable as octal = %o" %(var))
```

Output:

```
Variable as string = 27
Variable as raw data = '27'
Variable as integer = 27
Variable as float = 27.000000
Variable as printing with special char = Ram
Variable as hexadecimal = 1b
Variable as octal = 33
```

Built-in String Functions

There are various types of built-in string functions. We use those functions and get the desired results. One crucial point is that these types of functions return the new string. It cannot change the original string. We are showing some essential built-in string functions as shown in Table **2**.

Example:

```
1.  # built-in string functions
2.  txt = "Python is a good prog., Python is easy to learn, Python is easy to understand."
3.  cnt = txt.count("Python")
4.  print (cnt)
5.
6.  low = txt.islower()  # check if all the characters in the text are in lowercase
7.  print (low)
8.
9.  spl= txt.split()  #Split a string into a list where each word is a list item
10. print (spl)
11.
12. ew = txt.endswith(".") #Check if the string ends with a (.)
13. print (ew)
14.
15. cen = txt.center(10) # space of 10 characters
16. print (cen)
17.
18. an = txt.isalnum()  #Check if all the characters in the text are alphanumeric
19. print (an)
20.
21. isn = txt.isnumeric()  #Check if all the characters in the text are numeric
22. print (isn)
23.
24. tit = txt.title() # first letter in each word upper case
25. print (tit)
```

Output:

```
3

False

['Python', 'is', 'a', 'good', 'prog.,', 'Python', 'is', 'easy', 'to', 'learn,', 'Python', 'is', 'easy', 'to', 'understand.']

True

Python is a good prog., Python is easy to learn, Python is easy to understand.

False

False

Python Is A Good Prog., Python Is Easy To Learn, Python Is Easy To Understand.
```

Table 2. Built-in string functions.

count()	It returns the number of times a specified value occurs in a string
islower()	It returns True if all characters in the string are lowercase
split()	It splits the string at the specified separator and returns a list
endswith()	It returns true if the string ends with the specified value
center()	It returns a centered string
isalnum()	It returns True if all characters in the string are alphanumeric
isnumeric()	It returns True if all characters in the string are numeric
title()	It converts the first character of each word to upper case

Slice Operation

The slice() function returns a slice object that can be used to slice strings, lists, tuples, *etc.* There is the following syntax with parameters description of slice() function.

Syntax:

```
slice(stop)

slice(start, stop, step)
```

A **start** parameter provides the index at which an object's slicing begins. The index at which the slicing of an object is stopped can be determined by using a **stop** argument. The **step** parameter is an optional statement that determines the increment between each index throughout the slicing process. There are the following important points about the *slice()* function:

- It returns a sliced object that only contains elements from the defined set.
- If only one parameter is passed, then the start and step are considered to be none.

Example:

```
# It starts the slice object at position 2,  slice to position 4, and return the result:
var = ("P", "Y", "T", "H", "O", "N")
sl = slice(2, 4)
print (var[sl])

#It use the step parameter to return every second item:
var = ("P", "Y", "T", "H", "O", "N")
sl = slice(0, 6, 2)
print (var[sl])
```

Output:

```
('T', 'H')

('P', 'T', 'O')
```

Example:

```
# String slicing
str ='Pythonbook'
sl1 = slice(2)
sl2 = slice(1, 4, 3)
print ("String slicing")
print (str[sl1])
print (str[sl2])

# List slicing
lst = [5, 6, 7, 8, 9]
sl1 = slice(2)
sl2 = slice(1, 4, 2)
print ("\nList slicing")
print (lst[sl1])
print (lst[sl2])

# Tuple slicing
tpl = (4, 5, 6, 7, 8)
sl1 = slice(3)
sl2 = slice(1, 5, 2)
print ("\nTuple slicing")
print (tpl[sl1])
print (tpl[sl2])
```

Output:

```
String slicing
Py
List slicing
[5, 6]
[6, 8]
Tuple slicing
(4, 5, 6)
(5, 7)
```

The ord() Function

It is a built-in feature that returns the integer value corresponding to a single Unicode character as an integer. The Unicode code point is represented by the number that was returned. When a string of length 1 is provided to the **ord()** function, it returns an integer representing the character's Unicode code point. If the string's length is larger than one, a TypeError will be generated. Let us look at an example better to comprehend the notion of ord() functions.

Example:

```
1.  res1 = ord('A')
2.  res2 = ord('a')
3.  print (res1, res2)   # prints the Unicode value
```

Output:

```
65 97
```

The chr() Function

The **chr()** function converts an integer into a character or string and returns the result. The value of the number is the Unicode code point for the character. To better comprehend the notion of **chr()** functions, let us look at an example.

Example:

```
1. res1 = chr(65)   #return a character of Unicode valve 65
2. res2 = chr(97)   #return a character of Unicode valve 97
3. print (res1, res2)   # prints the character value
```

Output:

```
A a
```

String Comparison

Comparing strings means identifying whether the two strings are equivalent to each other or not. There are three ways to compare the strings:

1. Using Relational Operators
2. Using is and is not
3. Using user-defined function

Using Relational Operators

The relational operators compare the Unicode values of the characters of the strings. It returns a Boolean value according to the operator used. There are the following relational operators available in Python (Table **3**).

Table 3. String comparison using relational operators.

==	It checks whether two strings are equal.
!=	It checks if two strings are not equal.
<	It checks if the string on its left is smaller than on its right.
<=	It checks if the string on its left is smaller than or equal to that on its right.
>	It checks if the string on its left is greater than that on its right.
>=	It checks if the string on its left is greater than or equal to that on its right.

Example:

```
1.    print ("Python" == "Python")
2.    print ("Python" < "python")
3.    print ("Python" > "python")
4.    print ("Python" <= "python")
5.    print ("Python" >= "python")
6.    print ("Python" != "Python")
```

Output:

```
True

True

False

True

False

False
```

Using *is* and *is not*

This operator checks whether both operands refer to the same object or not. Let us take an example to understand this concept.

Example:

```
1.  str1 = "Book"
2.  str2 = "Book"
3.  str3 = str1
4.
5.  print (str1 is not str1)
6.  print (str1 is not str2)
7.  print (str1 is not  str3)
8.
9.  print (str1 is str1)
10. print (str1 is str2)
11. print (str1 is str3)
```

Output:

```
False
False
False
True
True
True
```

Using User-defined Function

We can make user-defined functions for string comparison. A user-defined function will compare the strings based on the number of digits.

Example:

```
def str_cmp(str1, str2):
    count1 = 0
    count2 = 0
    for i in range(len(str1)):
        if str1[i] >= "0" and str1[i] <= "9":
            count1 += 1

    for i in range(len(str2)):
        if str2[i] >= "0" and str2[i] <= "9":
            count2 += 1
    return count1 == count2

print (str_cmp("1234", "5672"))
print (str_cmp("1246", "Python"))
print (str_cmp("123Python", "Python123"))
```

Output:

```
True

False

True
```

Iterating String

Iterating over a string means accessing each of its characters simultaneously. There are various ways to iterate over the characters of a string in Python.

Example:

```
# Using simple iteration and range() function
str1 = "Programming"

for var in str1:
    print (var, end=' ')
print ("\n")



str2 = "Book"

for i in range(0, len(str2)):  # Iterate over index
    print (str2[i])
```

Output:

```
P r o g r a m m i n g

B

o

o

k
```

Example:

```
1.  #Using enumerate() function
2.  str3 = "Python"
3.
4.  # Iterate over the string
5.  for i, v in enumerate(str3):
6.      print (v)
```

Output:

```
P
y
t
h
o
n
```

In Python, we can also iterate over the words of a string. A string of several words separated by spaces is given. In Python, many methods for iterating over words in a string exist.

split() Function

We can split the string into a list of words, but this method fails in a string containing punctuation marks. Let us take an example.

Example:

```
1.  str1 = "Python is a good prog., Python is easy to use, Python is easy to understand"
2.  print ("The original string is: " +  str1)
3.
4.  result = str1.split()   # using split() to extract words from string
5.
6.  print ("\nThe words of string are")
7.  for i in result:
8.      print (i)
```

Output:

```
The original string is: Python is a good prog., Python is easy to use, Python is easy to
understand


The words of string are
Python
is
a
good
prog.,
Python
is
easy
to
use,
Python
is
easy
to
understand
```

re.findall() Function

This approach necessitates the use of regular expressions to complete the job. After filtering the string and extracting words while ignoring punctuation marks, the ***findall()*** function returns the list.

Example:

```
1.  #findall() example
2.  import re
3.
4.  str1= "Python is a good prog., Python is easy to use, Python is easy to understand."
5.  print ("The original string is: " +  str1)
6.
7.  result = re.findall(r'\w+', str1)
8.
9.  print ("\nThe words of string are")
10. for i in result:
11.     print (i)
```

Output:

```
The original string is: Python is a good prog., Python is easy to use, Python is easy to understand.

The words of string arePython

is

a

good

prog

Python

is

easy

to

use

Python

is

easy

to

understand
```

The String Module

The string module is a built-in module for using classes and constants. First, we have to import it before it can be used.

String Module Constants

There are various types of string constants defined in this module are:

string.ascii_letters

The following constants are concatenated: ascii_lowercase and ascii_uppercase.

string.ascii_lowercase

The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.

string.ascii_uppercase

The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.

string.digits

The string '0123456789'.

string.hexdigits

The string '0123456789abcdefABCDEF'.

string.punctuation

String of ASCII characters which are considered punctuation characters in the C locale: !”#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

Example:

```
# import string library function
import string

# Storing the value in variable result variables
result1 = string.ascii_letters
result2 =string.ascii_lowercase
result3 =string.ascii_uppercase
result4 =string.digits
result5 =string.hexdigits
result6 =string.punctuation
# Printing the value
print (result1)
print (result2)
print (result3)
print (result4)
print (result5)
print (result6)
```

Output:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
0123456789abcdefABCDEF
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

String capwords() Function

This function splits the specified string into words and capitalizes each word.

Example:

```
# import string library function
import string
str= 'My name is james'
result= string.capwords(str)
print (result)
```

Output:

```
My Name Is James
```

String Module Classes

There are two types of string module classes:

1. Formatter
2. Template

Formatter

This class becomes beneficial if we wish to subclass it and provide the format string syntax. It is also identical to the **str.format()** function. Let's look at an example to grasp the Formatter class idea further.

Example:

```
from string import Formatter

formatter = Formatter()
print (formatter.format('{Faculty}', Faculty='Dr. John William'))
print (formatter.format('{} {Faculty}', 'Hello', Faculty='Dr. John William'))

print ('{} {Faculty}'.format('Hello', Faculty='Dr. John William')) # it is same as forma
        t()
```

Output:

```
Dr. John William

Hello Dr. John William

Hello Dr. John William
```

Template

We can design output specification syntax that is simpler to comprehend thanks to the Template class. The format generates placeholder names by combining the symbol **$** with Python identifiers that allow the use, such as alphanumeric letters and underscores. Let's look at an example together so that you can better grasp the concept behind the Template class.

Example:

```
# template example
from string import Template
# Creating a template
temp = Template('P is $P')
# Substitute value
print (temp.substitute({'P' : 1}))
# List of faculty stores the name and their id
Faculty = [('James William',11), ('John Smith',22), ('David Sam',33)]
temp = Template('Hello $name,   $id is your employee id')
        for var in Faculty:
            print (temp.substitute(name = var[0], id = var[1]))
```

Output:

```
P is 1

Hello James William,   11 is your employee id

Hello John Smith,   22 is your employee id

Hello David Sam,   33 is your employee id
```

Regular Expression

A regular expression is a unique string of characters that employs a particular syntax to assist you in fitting or locating other strings or sets of strings. This may be accomplished *via* the usage of a regular expression. Additionally, it establishes a pattern consisting of a group of strings that correlate to it. The functions at your disposal can make use of the **metacharacters** listed below in Table **4**.

Table 4. Metacharacters description.

\	It is used to drop the special meaning of a character.
[ ]	It represents a character class.
^	It matches the beginning.
$	It matches the end.
.	It matches any character except newline.
?	It matches zero or one occurrence.
|	It matches with any of the characters separated by it.
*	It shows any number of occurrences (including 0 occurrences)
+	It is for one or more occurrences.

(Table 4) cont.....

{}	It indicates several occurrences of a preceding regular expression to match.
()	It encloses a group of regular expressions.

Example:

```
1.  import re
2.  p = re.compile('[a-m]')
3.  print (p.findall("My name is John Willian"))
```

Output:

```
['a', 'm', 'e', 'i', 'h', 'i', 'l', 'l', 'i', 'a']
```

The '\' metacharacter plays a crucial function in signaling different sequences. There are the following uses of metacharacter (\), which are shown in Table **5**.

Table 5. Metacharacters along with backslash (\).

Symbol	Description
\d	It's equivalent to the set class [0-9] and matches every decimal digit.
\D	It can be used to match any non-digit character.
\s	It is compatible with every whitespace character.
\S	It can be used to match any non-whitespace character.
\w	It is equivalent to the class [a-zA-Z0-9], which matches every alphanumeric character.
\W	It can be used to match any non-alphanumeric character.

Example:

```
1.  import re
2.  p1 = re.compile('\d')  # \d is equivalent to [0-9].
3.  print (p1.findall("My age is 40 year and my year of Birth is 1950"))
4.
5.  p2 = re.compile('\d+')  # \d+ will match a group on [0-9], group of one or greater size
6.  print (p2.findall("My age is 40 year and my year of Birth is 1950"))
7.
8.  p3 = re.compile('\w')   # \w is equivalent to [a-zA-Z0-9_].
9.  print (p3.findall("Welcome to the world of python. *"))
10.
11. p4 = re.compile('\w+')  # \w+ matches to group of alphanumeric character.
12. print (p4.findall("My age is 40 year and my year of Birth is 1950"))
13.
14. p5 = re.compile('\W')   # \W matches to non alphanumeric characters.
15. print (p5.findall("Welcome to the world of python. *"))
16.
17. p6 = re.compile('br*')  # '*' replaces the no. of occurrence of a character.
18. print (p6.findall("brbrbrbrbrbrbrbbbrrrrrr"))
```

Output:

```
['4', '0', '1', '9', '5', '0']
['40', '1950']
['W', 'e', 'l', 'c', 'o', 'm', 'e', 't', 'o', 't', 'h', 'e', 'w', 'o', 'r', 'l', 'd', 'o', 'f', 'p', 'y', 't', 'h', 'o', 'n']
['My', 'age', 'is', '40', 'year', 'and', 'my', 'year', 'of', 'Birth', 'is', '1950']
[' ', ' ', ' ', ' ', ' ', '.', ' ', '*']
['br', 'br', 'br', 'br', 'br', 'br', 'br', 'b', 'b', 'brrrrrr']
```

Example:

```
1.  import re
2.
3.  # Upon matching, 'y' is replaced by '!!' and CASE has been ignored
4.  print (re.sub('y', '~*' , 'My age is 36 year and my year of Birth is 1985', flags = re.IGNOR
        ECASE))
5.
6.  # Consider the Case Sensitivity, 'y' in "year", will not be reaplced.
7.  print (re.sub('y', '!!' , 'My age is 36 year and my year of Birth is 1985'))
8.
9.  # As count has been given value 1, the maximum times replacement occurs is 1
10. print (re.sub('y', '!!' , 'My age is 36 year and my year of Birth is 1985', count=1, flags = re
        .IGNORECASE))
11.
12. # 'r' before the patter denotes RE, \s is for start and end of a String.
13. print (re.sub(r'\sAND\s', ' & ', 'Book and note book', flags=re.IGNORECASE))
```

Output:

```
M~* age is 36 ~*ear and m~* ~*ear of Birth is 1985
M!! age is 36 !!ear and m!! !!ear of Birth is 1985
M!! age is 36 year and my year of Birth is 1985
Book & note book
```

LISTS

A list is an essential data type in Python, written as a comma-separated value enclosed with square brackets. There are the following important points about lists:

- List items are ordered, and that order will not change except for some list

methods.
- The list allows duplicate values.
- The list is changeable, which means we can add and delete items in a list after it has been created.
- The first item has an index [0], and indexing will increase.

We can make any list by putting the series in square brackets [] and separating it with commas. It may contain any number of items of various kinds, such as integers, floats, strings, and so on. Let us take a basic example.

Example:

```
1.  # empty list
2.  l1 = []
3.  # list of integers
4.  l2 = [4, 5, 6, 7]
5.  # list with mixed data types
6.  l3 = [2, "Python", "Book", 120.8]
7.  print (l1)
8.  print (l2)
9.  print (l3)
```

Output:

```
[]
[4, 5, 6, 7]
[2, 'Python', 'Book', 120.8]
```

Every value in a list is indexed and can be accessed by referring to the index number. There are various ways to access a list of elements.

Using List Index

The index operator [] can gain access to a list object. The indices start at 0 in this system. Suppose there are seven items on the list; the order of the numbers then goes from 0 to 6. If you try to access values that are not contained in the index, you will receive an IndexError. One of the most important things to remember is that an index's value must always be an integer, and values of any other kind are not permitted. If you try to use a different data type for the index value, you will receive a TypeError message. Negative indexing is also allowed for list sequences. In negative indexing, -1 refers to the last item, -2 to the second last, -3 to the third last, and so on.

Example:

```
# List indexing
ListExample = ['P','Y','T','H','O','N']
print (ListExample [0])
print (ListExample [1])
print (ListExample [2])
print (ListExample [3])
print (ListExample [4])
#Negative index
print ('--Negative index--')
print (ListExample [-1])
print (ListExample [-2])
```

Output:

```
P
Y
T
H
O
--Negative index--
N
O
```

List Operations

A slicing, deleting, removing, and clearing operation is performed on a list of items. By utilizing the slicing operator, it is possible to gain access to it across the range of items included in a list. The colon is used to symbolize it (:). The assignment operator (=) enables adding new elements to the list and modifying existing ones. The **delete()** method may be used to remove a particular object, while the **pop()** method can be used to remove an item from a certain index. We may use the **clear()** function to clean a list thoroughly. Let's look at an example to help better grasp how the mentioned operators and methods work.

Example:

```
1.  ml = ['p','y','t','h','o','n']
2.  print(ml[2:5])  #elements 3rd to 5th
3.  print(ml[:-6]) #elements beginning to 5th
4.  print(ml[3:]) #elements 4th to end
5.  print(ml[:]) #elements beginning to end
6.  del ml[2]  # delete one item
7.  print(ml)
8.  del ml[1:5] # delete multiple items
9.  print (ml)
10. mlist = ['P','Y','T','H','O','N']
11. mlist.remove('Y')
12. print (mlist)
13. print (mlist.pop(1))
14. print (mlist)
15. print (mlist.pop())
16. print (mlist)
17. mlist.clear()
18. print (mlist)
19. del ml   # delete entire list
20. print (ml)  # Error: List not defined
```

Output:

```
['t', 'h', 'o']
[]
['h', 'o', 'n']
['p', 'y', 't', 'h', 'o', 'n']
['p', 'y', 'h', 'o', 'n']
['p']
['P', 'T', 'H', 'O', 'N']
T
['P', 'H', 'O', 'N']
N
['P', 'H', 'O']
[]
Traceback (most recent call last):
  File "<string>", line 25, in <module>
NameError: name 'ml' is not defined
>
```

Updating Values in Lists

We can update single or multiple lists. It can use the (=) assignment operator to append and extend the method to accomplish the task. Let us take an example for understanding the mentioned operators and methods.

Example:

```
1.    # Add/Change List Elements
2.    var = [1, 2, 3, 4]
3.    var[0] = 9    # change the 1st item
4.    print (var)
5.
6.    var[1:4] = [22, 33, 44]  # change 2nd to 4th items
7.    print (var)
8.
9.    # Appending and Extending lists in Python using the append()
10.   var = [11, 22, 33]
11.   var.append(44)
12.   print (var)
13.
14.   var.extend([55, 66, 77])  #Add several items using extend() method
15.   print (var)
16.
17.   print (var + [224,353,464]) # Add + operator to combine two lists.
18.
```

Output:

```
[9, 2, 3, 4]
[9, 22, 33, 44]
[11, 22, 33, 44]
[11, 22, 33, 44, 55, 66, 77]
[11, 22, 33, 44, 55, 66, 77, 224, 353, 464]
```

Nested List

A nested list means a list within a list. Moreover, a list contains a sublist of elements. Suppose we have a list named **lst** that includes a list and sublist. A nested list is made up of a series of sublists separated by commas.

```
lst = ['ab', ['cd', ['efg', 'hij'], 'kl', 'mn'], 'o', 'p']
```

Example:

```
lst = ['ab', ['cd', ['efg', 'hij'], 'kl', 'mn'], 'o', 'p']

print (L[2]) # Prints' o'

print (L[2][2]) # Prints' kl'

print (L[2][2][0]) # Prints 'k'
```

Output:

```
o
kl
k
```

It can add and remove items in a Nested list by using the *append(), insert(), extend(), pop(), del, and remove()* methods. It can also find length using *len()* and iterate through a nested list using for loop. Let us take an example to understand this concept.

Example:

```
lst = ['P', ['ab', 'cd'], 'Y']
lst[1][1] = 0
print (lst)

lst[1].append('X')
print (lst)

lst[1].insert(0,'PP')
print (lst)

        lst[1].extend([77,88,99])
        print (lst)
```

Output:

```
['P', ['ab', 0], 'Y']
['P', ['ab', 0, 'X'], 'Y']
['P', ['PP', 'ab', 0, 'X'], 'Y']
['P', ['PP', 'ab', 0, 'X', 77, 88, 99], 'Y']
```

Aliasing

When we assign one variable to another, all of those variables will refer to the same object since variables are themselves objects. Additionally, the process of using two different IDs to refer to the same variable is known as aliasing. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  var1 = [555, 666,777]
2.  var2 = var1    #Aliasing
3.  print (var1 is var2)
4.  print (var1)
5.  print (var2)
```

Output:

```
[True
[555, 666, 777]
[555, 666, 777]
```

The same list has two different names in the above example, var1, and var2; however, after executing the assignment statement var2 = var1, you can see that var1 and var2 relate to the same list.

Example:

```
1.   var1 = [555, 666,777]
2.   var2 = [555, 666,777]
3.   print (var1)
4.   print (var2)
5.   print (var1 == var2)
6.   print (var1 is var2)
7.
8.   var2 = var1
9.   print (var1 == var2)
10.  print (var1 is var2)
11.
12.  var2[0] = 999
13.  print (var1)
14.  print (var2)
```

Output:

```
[555, 666, 777]
[555, 666, 777]
True
False
True
True
[999, 666, 777]
[999, 666, 777]
```

Cloning Lists

If we want to change a list while keeping a copy of the original, we must copy the entire list, not just the reference. This method is often referred to as cloning to prevent the ambiguity of the word clone. The quickest and most straightforward way to clone a list is to use the slicing process.

Example:

```
1.  list1 = [777, 888, 999]
2.  list2 = list1[:]      # make a clone using slice
3.  print (list1 == list2)
4.  print (list1 is list2)
5.  list2[0] = 555
6.  print (list1)
7.  print (list2)
```

Output:

```
True
False
[777, 888, 999]
[555, 888, 999]
```

There are various types of ways to clone a list.

Cloning List using Slicing

It is the most straightforward method of cloning a list. We use this form when we want to change a list while keeping a copy of the original. Let us take an example to understand this concept.

Example:

```
1.  def cloning_List(li1):
2.      lst1 = li1[:]   #Slice Operator
3.      return lst1
4.
5.
6.  lvar1 = [55,66,77,88,99]
7.  lvar2 = cloning_List(lvar1)
8.  print ("Original List:", lvar1)
9.  print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Cloning List using Extend()

The extend() function can be used to copy the lists into a new list. Let us take an example to understand this concept.

Example:

```
1.  def cloning_List(li1):
2.      lst1 = []
3.      lst1.extend(li1)
4.      return lst1
5.
6.  lvar1 = [55,66,77,88,99]
7.  lvar2 = cloning_List(lvar1)
8.  print ("Original List:", lvar1)
9.  print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Cloning List using List()

A list can be cloned using a list() function. Let us take an example to understand this concept.

Example:

```
def cloning_List(li1):
    lst1 = list(li1)
    return lst1

lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Cloning List using List Comprehension

A list comprehension method is used to copy all the elements from one list to another. Let us take an example to understand this concept.

Example:

```
def cloning_List(li1):
    lst1 = [i for i in li1]
    return lst1


lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Cloning List using Append()

The **append()** function can be used to append or add elements to the list or copy them to a new list. It's used to add elements to the list's last location. Let us take an example to understand this concept.

Example:

```
def cloning_List(li1):
    lst1 = []
    for i in li1: lst1.append(i)
    return lst1


lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Cloning List using Copy()

Use this method to copy all the elements from one list to another. Let us take an example to understand this concept.

Example:

```
def cloning_List(li1):
    lst1 =[]
    lst1 = li1.copy()
    return lst1


lvar1 = [55,66,77,88,99]
lvar2 = cloning_List(lvar1)
print ("Original List:", lvar1)
print ("After Cloning:", lvar2)
```

Output:

```
Original List: [55, 66, 77, 88, 99]
After Cloning: [55, 66, 77, 88, 99]
```

Built-in Functions

There are various types of built-in functions that are performed on the list. We can easily accomplish our tasks by directly using these functions (Table **6**).

Table 6. List Built-in functions.

cmp(list1, list2)	This function compares elements of both lists.
len(list)	This function returns the length of the list.
max(list)	This function returns an element with a max value from the list.
min(list)	This function returns an element from the list with a min value.
list(seq)	This function converts a tuple into a list.

Example:

```
1.  list1 = ['Amit', 10]
2.  list2 = ['David', 'James']
3.
4.  def cmp(p, q):
5.      return (p > q) - (p < q)
6.
7.  print (cmp(list1, list2))  # List comparison
8.
9.  print (len(list1))  # Length of list
10.
11.         list3 = [53535, 68868, 67656]
12.         print ("Maximum value element : ", max(list3))  # Getting Max value from list
13.
14.         print ("Minimum value element : ", min(list3)) # Getting Min value from list
```

Output:

```
-1
2
Maximum value element :  68868
Minimum value element :  53535
```

List Built-in List Methods

Some other built-in methods are also performed on the list. We can easily accomplish our task by directly using this method (Table **7**).

Table 7. List built-in list methods.

list.append(obj)	This method appends object obj to the list.
list.count(obj)	This method returns a count of how often obj occurs in the list.
list.extend(seq)	This method appends the contents of seq to the list.

(Table 7) cont.....

list.index(obj)	
list.insert(index, obj)	This method inserts object obj into a list at offset index.
list.pop(obj=list[-1])	This method removes and returns the last object or obj from the list.
list.remove(obj)	This method removes object obj from the list.
list.reverse()	This method reverses objects of the list in place.
list.sort([func])	This method sorts objects of the list.

Example:

```
1.lst1 = ['p', 'y', 't', 'h', 'o', 'n']
2.
3.lst1.append('programming')
4.print (lst1)
5.
6.lst1.extend(['x','y','z'])
7.print (lst1)
8.
9.print ("Count  : ", lst1.count('programming'))
10.
11.   print ("Index for programming : ", lst1.index( 'programming' ) )
12.
13.   lst1.insert( 3, 100)
14.   print (lst1)
15.
16.   print ("Poped List : ", lst1.pop())
17.
18.   lst1.remove('programming')
19.   print (lst1)
20.
21.   print (lst1.reverse())
22.
23.   lst2 = [44,464,686,313,668,242,575,1212,464]
24.
25.   lst2.sort()
26.   print (lst2)
27.
28.   print (lst2.reverse())
29.   print (lst2)
```

Output:

```
['p', 'y', 't', 'h', 'o', 'n', 'programming']
['p', 'y', 't', 'h', 'o', 'n', 'programming', 'x', 'y', 'z']
Count  :  1
Index for programming :  6
['p', 'y', 't', 100, 'h', 'o', 'n', 'programming', 'x', 'y', 'z']
Poped List :  z
['p', 'y', 't', 100, 'h', 'o', 'n', 'x', 'y']
None
[44, 242, 313, 464, 464, 575, 668, 686, 1212]
None
[1212, 686, 668, 575, 464, 464, 313, 242, 44]
```

List as Arrays

The list is a collection of elements of various data kinds, such as logical character values, numeric values, and so on. Because an array is used in many computer languages, including C, C++, Java, *etc.*, it is also well recognized. A group of items of homogenous make up an array. An exception with the message "Incompatible data types" is raised if the array's items have different data types. Let's use an illustration to help you grasp this idea.

Example:

```
1.  # Simple list
2.  list1 = ['John','James', 10, 20, -5, 'David']
3.  print (list)
4.  # Using array in Python
5.  import array
6.  A = array.array('i', [30, 50, 80])
7.  # accessing elements of array
8.  for i in A:
9.      print (i)
```

Output:

```
<class 'list'>
30
50
80
```

There are differences between a list and an array in Python:

- A list can consist of elements belonging to different data types, but an array only consists of elements belonging to the same data type.
- A list does not need to import a module for declaration explicitly, but an array needs to import a module for representation explicitly.
- A list cannot directly handle arithmetic operations, but an array can directly manage arithmetic operations.
- A list can be nested to contain different elements, but an array must contain all nested elements of the same size.
- A list is preferred for a shorter sequence of data items, but an array is preferred for a longer sequence of data items.
- A modification is easier in the list compared to an array.
- The entire list can be printed without explicit looping but array using a loop to print or access the components.

Loops in Lists

When working with lists, "looping" involves displaying each item in the list in succession. In addition, the Python operation is an iteration through a list. Python provides a wide variety of options for iterating through a list in a program.

For Loop in the List

Displaying each item on a list may be accomplished with the help of a for loop. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  list = [11, 22, 33, 44, 55]
2.
3.  for var in list:
4.      print (var)
```

Output:

```
11
22
33
44
55
```

Let's say we wish to utilize the standard for loop, which iterates through the element until the range we provide. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  list = [11, 22, 33, 44, 55]
2.
3.  length = len(list) # getting the length of a list
4.
5.  for var in range(length):
6.      print (list[var])
```

Output:

```
11
22
33
44
55
```

While Looping in the List

Although it operates in a manner comparable to a for loop, its syntax is distinct. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  list = [11, 22, 33, 44, 55]
2.
3.  length = len(list) # getting length of list
4.  x1 = 0
5.  while x1 < length:
6.      print (list[x1])
7.      X1 += 1
```

Output:

```
11
22
33
44
55
```

List Comprehension

In Python, it is necessary to use it to iterate through a list. It is an alternative method for constructing a list of components with a particular quality. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  list = [11, 22, 33, 44, 55]
2.
3.  [print (i) for i in list]   # Using list comprehension
```

Output:

```
11
22
33
44
55
```

Enumerate() Method

This method can convert the list into an iterable list of tuples. Let us take an example to understand this concept.

Example:

```
1.  list = [11, 22, 33, 44, 55]
2.
3.  for i, val in enumerate(list):   # Using enumerate()
4.      print (i, "=",val)
```

Output:

```
0 = 11
1 = 22
2 = 33
3 = 44
4 = 55
```

Numpy in Loop

When working with extensive n-dimensional lists, it is often preferable to use an external library such as NumPy. To better grasp this idea, let's look at a concrete illustration.

Example:

```
import numpy as np

var = np.arange(4)
var = var.reshape(2, 2)

for var1 in np.nditer(var): # iteration
    print (var1)
```

Output:

```
0
1
2
3
```

List in Functions

When you provide a list to a function, you pass a list variable to the function. The value of the list variable is then copied to the function arguments by the function. By using the function's arguments, it is, therefore quite simple to display the list's values. To better grasp this idea, let's look at a concrete illustration.

Example:

```
def students(f):
    for i in f:
      print (i)
l1 = ["David", "James", "William", "John"]
students(l1)
```

Output:

```
David
James
William
John
```

A function can return a list. When you say “returning list to function,” you are telling the function that was called to get the list's value and catch. To print the results, another variable has assigned a calling function. To better grasp this idea, let's look at a concrete illustration.

Example:

```
1.  def students(f):
2.      for i in f:
3.        return (i)
4.  l1 = ["David", "James", "William", "John"]
5.  students(l1)
6.
7.  for i in l1:
8.      students(i)
9.      print (i)
10.
```

Output:

```
David
James
William
John
```

CONCLUDING REMARKS

In this chapter, we spoke about strings and lists with their different operations, such as concatenating, appending, multiplying, and other things. The string formatting operator and the process that it uses is the focus of this section. The built-in string methods slice, ord(), and chr(), comparing and iterating a string, and further clarifying regular expression with an excellent example are also covered in this section. In the following chapter, we will move on to the next topic utilized in Python: tuples and dictionaries.

CHAPTER 6

Tuple and Dictionaries

Abstract: Another important data type of Python is dictionary. In the previous chapter, we acquainted the readers with the lists and strings. In this chapter, we shall discuss dictionaries and lists. lists are the ordered collection of objects but dictionaries are an unordered collection of objects. dictionary values are referred to using key-value pair instead of positional offset. Due to this, they have found great usage in search tables, records and aggregation. Another concept introduced in this chapter is tuples. These are immutable like strings and represent a stable collection of arbitrary items.

Keywords: Hash tables, Immutable and mutable, Mappings.

INTRODUCTION

One of Python's most significant data types is called the tuple, which holds several components as an object. It is also known as an immutable data type, which indicates that its members cannot be altered in any way after they have been set. A dictionary is another data type that may be used in Python, and it functions similarly to tuples and lists. An associative array is another name for this structure. A dictionary is a collection of "key-value pairs". Each key-value pair connects the key to the value that corresponds to it. In this chapter, you will learn about various operations and attributes that can be used with dictionaries and tuples. In addition, the many features and actions of dictionaries and tuples are broken down and illustrated using examples and codes.

TUPLE

A tuple is a group of ordered, immutable items arranged in a specific way. Tuples are structured similarly to lists and strings. In this context, the definition of immutable is that the components of the tuple are not subject to change. Once a tuple has been generated, it is not possible to add or remove items from the tuple. Even we cannot change the order in which the tuple members are presented. In addition, the length of the tuple cannot be altered. It is necessary to generate a new tuple if we wish to modify an existing one by adding or taking something away from it. Unlike lists, tuples are denoted by parentheses and cannot be edited. Tuples are a sort of sequence equivalent to strings in terms of structure. Tuples

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

can hold any components, in contrast to strings, which can only store characters. It indicates that the tuple contains either a list of students' names or employee IDs, depending on which one is selected. Tuples may also be used to store varied elements, implying that a single tuple can contain components of several data formats, such as decimal formats, integers, and characters. A sequence of music files, picture files, and other data types can also be stored in tuples.

To create a tuple in Python, all the elements are enclosed in () parenthesis, separated by a comma. A tuple can store heterogeneous data elements. Below are examples of creating tuples.

Example:

```
1.  # Create an empty tuple
2.  Tuple1 = ()
3.  # tuple with 5 elements
4.  Tuple2 = (10,20,30,40,50)
5.  print (Tuple2)
6.  # tuple of character elements
7.  Tuple3 = ('x', 'y', 'z')
8.  print (Tuple3)
9.  # tuple of strings
10. Tuple4 = ("Python", "Programming", "Book")
11. print (Tuple4)
12. # tuple of heterogeneous data elements
13. Tuple5 = (100, 20.123, "Python Programming")
14. print (Tuple5)
15. # tuple of string and List
16. Tuple6 = ("Python Book", [10, 20, 25])
17. print (Tuple6)
```

Output:

```
(10, 20, 30, 40, 50)

('x', 'y', 'z')

('Python', 'Programming', 'Book')

(100, 20.123, 'Python Programming')

('Python Book', [10, 20, 25])
```

To create a tuple of a single element, it should be followed by a comma. The following example creates a tuple of a single element.

Tup1 = (100,)

In the above case, if we do not put a comma after 100, then Python would treat **Tup1** as an integer rather than a tuple variable.

Example:

```
1. Tup1 = (100,)
2. type (Tup1)
3. Tup2=(100)
4. type (Tup2)
```

Output:

```
<class 'tuple'>

<class 'int'>
```

The*Tuple ()*Function

Python has a built-in function called ***tuple()*** that may be used to make tuples. While we can build tuples without utilising this function, it offers a different method. The ***tuple()*** function is used to construct a tuple in the following example.

Example:

```
1. # example of tuple() function
2. # creating an empty tuple
3. T1 = tuple()
4. print (T1)
5. # creating a tuple from List
6. T2 = tuple([1,2,3,4,5])
7. print (T2)
8. # creating a tuple from strings
9. T3 = tuple("Python Programming")
10. print (T3)
```

Output:

```
()

(1, 2, 3, 4, 5)

('P', 'y', 't', 'h', 'o', 'n', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g')
```

Accessing Tuple

To access the elements of a tuple, indexes are used. Tuple indexes start at 0, like string indices, and can be sliced, concatenated, and so on. The following examples show how we can access values in tuples.

Example:

```
# tuple of characters
tup1 = ('A', 'B', 'C', 'D')
# tuple of integers
tup2 = (10, 20, 30, 40, 50, 60, 70 )
# displaying first element of tuple1
print  tup1[0]
# displaying second to fifth elements of tuple 2
print tup2[1:5]
```

Output:

```
A

(20, 30, 40, 50)
```

Tuple Indexes

Indexes in Python can be positive or negative, like list and string indexes. Negative indices begin at -1, whereas positive indices begin at 0. While negative indices are used to access items from the tuple's end, positive indexes are used to access elements from the tuple's beginning. The below example demonstrates the use of a negative index.

Example:

```
Tuple1 = (10, 20, "Books", 108.92)
# Accessing last element using negative index
print (Tuple1[-1])
# Accessing third element from last
print (Tuple1[-3])
```

Output:

```
108.92

20
```

Tuple Slicing

The action of slicing is analogous to that of strings. We need to specify the beginning and end of the slicing. The result is a tuple that lies between those two places and incorporates each of the components. We have to use the [] operator on the tuple to slice a tuple. When indexing a tuple, the index is taken from the count of the tuples on the left if the integer we are working with is positive. If the index is negative, the index is derived from the tuple count on the right. The code that follows is an illustration of slicing operations that may be performed on the tuple.

Example:

```
1. T1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
2. # print elements from 3rd to 6th
3. print (T1[2:6])
4. # print elements from start to 4th position
5. print (T1[:4])
6. # print elements from 4th to last
7. print (T1[3:])
8. # print elements from 6th to second last
9. print (T1[5:-1])
10. # print entire tuple
11. print (T1[:])
```

Output:

```
(3, 4, 5, 6)

(1, 2, 3, 4)

(4, 5, 6, 7, 8, 9, 10)

(6, 7, 8, 9)

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Updating Tuple

Tuples are immutable, which means that the values of the items included within a tuple will not be altered or changed in any way. However, if we want to update the nested items they are changeable. In the following piece of code, for instance, we modify the list element that is included within the tuple. Because the list's components are adjustable, it is acceptable to do so.

Example:

```
1. T1= (1, [2, 3, 4,5], "Python")
2. print (T1)
3. # Updating is not valid for tuples
4. #T1[0] = 100    # It generates error
5. # Changing the list element, this is valid because List is mutable
6. T1[1][2] = 10
7. print (T1)
```

Output:

```
(1, [2, 3, 4, 5], 'Python')

(1, [2, 3, 10, 5], 'Python')
```

We can create a new tuple using the portion of an existing tuple. The below code demonstrates the creation of a new tuple.

Example:

```
1. T1 = (1, 2, 3, 4.5)
2. T2 = ('ABCD', 'XYZW')
3. # New tuple creation from existing
4. T3 = T1 + T2[0:1]
5. print T3
```

Output:

```
(1, 2, 3, 4.5, 'ABCD')
```

Deleting Elements from a Tuple

We have already shown that the elements of a tuple are immutable, which means they cannot be altered or removed. However, it is best to remove the entire tuple if feasible. To remove a whole tuple, use the ***del*** statement.

Example:

```
1. T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
2. # Printing tuple
3. print (T2)
4. #del T2[2]    #it will generate an error
5. # deleting an entire tuple
6. del T2
7. #print after deleting a tuple
8. print (T2)        #it will generate error
```

Output:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Traceback (most recent call last):
  File "main.py", line 11, in <module>
    print(T2)        #it generates error
NameError: name 'T2' is not defined
```

Tuple Operations

The many operations that may be performed on tuples are relatively similar to those that can be performed on lists. In most cases involving tuples, we only encountered the + operator. It works with tuples in exactly the same way as it does with lists. Tuples respond to the + and * operators similarly to strings; they also imply concatenation and repetition in this context; however, a new tuple is a result rather than a string. When subjected to any general sequence operations, tuples behave similarly to strings. Addition operators combine all the elements of two or more tuples into a new tuple. If a tuple is multiplied by any integer, *x* would create another tuple with all the elements repeated *x* times from the first tuple. *T*5* indicates that the elements of tuple *T* will be replicated five times.

Example:

```
1. T1 = (1, 2, 3)
2. T2=(4, 5)
3. T3=(10,)
4. #Addition
5. T4=T1+T2+T3
6. print (T4)
7. #Multiplication
8. print (T1*5)
9. #replicate string 3 times
10. print ("Python"*3)
```

Output:

```
(1, 2, 3, 4, 5, 10)
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
PythonPythonPython
```

Tuple Basic Functions

Table **1** lists some fundamental functions and keywords that are applicable to tuples.

Table 1. Tuple basic functions.

len()	This function is used to get the number of elements in the tuple.
max()* and *min()	These functions are used to get the maximum and minimum values from the tuple.
sum()	This function is used to get the sum of all the elements of the tuple.
in	This keyword is used with tuples same as strings and lists. It is used to check whether any element is present or not in the tuple. If the element is present, it returns True. Otherwise, it returns False.
cmp()	In order to compare two tuples, the **cmp()** function is called upon. It will return either 1, 0, or -1 depending on whether or not the two tuples being compared are comparable. The comparison of these two tuples is handled by this function, which accepts them as arguments. If T1 and T2 are two different tuples, then the comparison will provide the following result based on what it finds. *if T1 > T2, then cmp(T1, T2) returns 1,* *if T1 = T2, then cmp(T1, T2) returns 0* *if T1 <T2, then cmp(T1, T2) returns -1* *cmp() function is only in python 2.x. It does not work in python 3.x.*
index()	It is a built-in Python function that searches for a given element in a tuple and returns the lowest index where the element appears. *index()* function can take a maximum of three arguments and follows the given syntax: **tuple.index(element, start, end)** Here, the **element** is the first argument that must be searched in the tuple. **Start** and **end** are optional arguments that indicate starting and ending point index of the searching element.
count()	The *count()* function returns the number of occurrences in the given tuple for an element.

Example:

```
#tuple with six elements
T = (10, -2, 300, 40, 302, 900)
# length of the tuple
print ("Length:", len(T))
# Max value from tuple
print ("Max:", max(T))
# Min value from tuple
print ("Min:", min(T))
# Sum of all elements of tuple
print ("Sum:", sum(T))
# Check whether 40 is present or not in tuple
print (40 in T)
# Check whether 500 is present or not in tuple
print (500 in T)

#======================index() Example
# Tuple of four elements
T1 = (100, 20,30,40)
# tuple of characters
T2=("PYTHON")
# print the index of 100 in tuple T1
print ("Index No.:",T1.index(100))
# print the index of T in tuple T2
print ("Index No.:",T2.index('T'))
# search the index of 20, start from 1 to 4
print ("Index No.:",T1.index(20,1,4))
#==============count() Example
#tuple 1
T1 = (1, 1, 1, 2, 2, 2, 1, 2, 3, 1, 1)
#tuple 2
T2 = ('a', 'a', 'a', 'b', 'b', 'a', 'c', 'b', 'a','b')
#tuple 3
T3 = ['Bat', 'Cat', 'Sat', 'Bat', 'cat', 'Bat']
# Counts the number of times 1 appears in tuple 1
print ("Count of 1:", T1.count(1))
# Counts the number of times 'a' appears in tuple 2
print ("Count of a:", T2.count('a'))
# Counts the number of times 'Bat' appears in tuple 3
print ("Count of Bat:", T3.count('Bat'))
```

Output:

```
Length: 6
Max: 900
Min: -2
Sum: 1550
True
False
Index No.: 0
Index No.: 2
Index No.: 1
Count of 1: 6
Count of a: 5
Count of Bat: 3
```

Tuple Assignment

Tuple assignment is a very helpful function that Python provides. It is named so because it allows values to be assigned to a tuple of variables on the left-hand side of an assignment from a tuple on the right-hand side of the assignment. **Unpacking** a tuple of values into a variable is another name for this operation. When we **pack**, the data is placed inside a new tuple, but when we unpack, we remove those values and place them inside a single variable. With the assistance of this feature, it is possible to assign more than one variable at once while a sequence is displayed on the left. The following illustration shows how one tuple, T1, may be used to assign values to five distinct variables. In this task, the number of variables listed on the left-hand side of the equation must match the number of tuple elements listed on the right-hand side of the equation. If this requirement is not met, an error will be generated.

Example:

```
# tuple with five elements/Packing
T1= ('David', 'India', 1990, 1001)
print ("Tuple Elements are:", T1)
# tuple assignment to different variables/Unpacking
(EmpName, Country, BirthYear, EmpId)=T1
# printing individual variables
print ("--------------------------")
print ("EmpName:",EmpName)
print ("Country:", Country)
print ("BirthYear:", BirthYear)
print ("EmpId:",EmpId)
```

Output:

```
Tuple Elements are: ('David', 'India', 1990, 1001)

-------------------------

EmpName: David

Country: India

BirthYear: 1990

EmpId: 1001
```

In the above example, we have a tuple T1 which contains five elements and assign different values to the variables EmpName, Country, BirthYear, and EmpId in a single statement. We can also assign individual element values to variables such as:

Example:

```
# tuple with five elements
T1= ('David', 'India', 1990, 1001)
print ("Tuple Elements are:", T1)
# assigning one value at a time
EmpName=T1[0]
Country=T1[1]
BirthYear=T1[2]
EmpId=T1[3]
#printing individual values
print ('------------------')
print (EmpName)
print (Country)
print (BirthYear)
print (EmpId)
```

Output:

```
Tuple Elements are: ('David', 'India', 1990, 1001)
------------------
David
India
1990
1001
```

Tuples and Functions

When calling a function with the prefix *, a unique method can be used to get the parameters in the form of a tuple. This is useful information when the number of arguments in the function is taken as a function. Because of the * prefix placed on the args variable; any further arguments sent to the function will be saved in that variable as a tuple.

Example:

```
1.  #This function receives a tuple as an argument
2.  def fun1(*args):
3.      print (type(args))  #printing the type of args
4.      print (args)
5.  # creating a tuple
6.  T1=("Python", "Programming book", 2022)
7.  # passing tuple to a function
8.  fun1(T1)
```

Output:

```
<class 'tuple'>
(('Python', 'Programming book', 2022),)
```

It is not feasible to return numerous values from a function in C or other programming languages, but in Python, doing so is not only conceivable but also rather straightforward. We can return numerous values in Python by using simple returning statements that are delimited by commas. The following code snippet demonstrates how to define a function called ***fun_ret_mul()*** that will return both a string and a number.

Example:

```
#Returning multiple values as tuple
# This function returns a tuple
def fun_ret_mul():
    s1 = "Python book"
    x = 2022
    return (s1, x)
# calling function
T1= fun_ret_mul() #T1 is a tuple
print (type(T1))
print (T1)
```

Output:

```
<class 'tuple'>

('Python book', 2022)
```

Nesting of Tuples

Python allows us to declare a tuple inside another tuple; this type of tuple is referred to as a nested tuple. Nesting tuples are used whenever there is a need to organize data into hierarchical structures. If only one element is included within a tuple, then we cannot refer to it as a tuple. It should consist of a comma after it as a trailing mark to indicate to the interpreter that it is a tuple. In the following example, we have declared a tuple contained within another tuple.

Example:

```
# Nested tuples
Tuple1 = ("Python", [1,2,3], (5, 6.6, "Programming", "Book"))
# Print all elements of tuple
print ("Tuple values are:", Tuple1)
# Print first element
print ("first Element:", Tuple1[0])
#print second element or sub-tuple
print ("sub tuple:", Tuple1[1])
#print third element or sub-tuple
print ("sub tuple:", Tuple1[2])
```

Output:

```
Tuple values are: ('Python', [1, 2, 3], (5, 6.6, 'Programming', 'Book'))
first Element: Python
sub tuple: [1, 2, 3]
sub tuple: (5, 6.6, 'Programming', 'Book')
```

Tuples from String and List

In Python, a tuple may be constructed not just from a text but also from a list. This kind of conversion is required whenever multiple types of data are available and the need to create a new type. You can convert a string or a list into a tuple by using the ***tuple()*** function. The following illustration provides further clarification of this idea.

Example:

```
1.  # List and string creation
2.  L1 = ['Python',1,2,3,4,5,6]
3.  S1 = "Programming Language"
4.  # printing original list and string
5.  print ("The original List : " , L1)
6.  print ("The original String : " , S1)
7.  print (type(L1))
8.  print(type(S1))
9.  #List to tuple creation
10. result1 = tuple(L1)
11. #String to tuple creation
12. result2=tuple(S1)
13. print ("Converted List : " , result1)
14. print (type(result1))
15. print ("Converted String : " , result2)
16. print (type(result2))
```

Output:

```
The original List :  ['Python', 1, 2, 3, 4, 5, 6]
The original String :  Programming Language
<class 'list'>
<class 'str'>
Converted List :  ('Python', 1, 2, 3, 4, 5, 6)
<class 'tuple'>
Converted string:  ('P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', ' ', 'L', 'a', 'n', 'g', 'u', 'a', 'g', 'e')
<class 'tuple'>
```

zip() Function

The Python zip () function is mainly used to combine data of two container elements. This function takes iterables and aggregates their elements based on their order in the iterables. As an input, it takes any number of iterables and returns an iterator of tuples. Iterables may be lists, strings, *etc*. Below is an example of group items of two different lists based on the same index.

Example:

```
1. #creating two lists
2. List1 = ['A', 'B', 'C', 'D']
3. List2  = [10, 20, 30, 40]
4. #combine using zip
5. Result = tuple(zip(List1, List2))
6. print (Result)
7. print (type(Result))
```

Output:

```
(('A', 10), ('B', 20), ('C', 30), ('D', 40))
<class 'tuple'>
```

The returned iterator takes the length of the shortest iterator passed to the function if the passed iterables are of different lengths. In the below example, we have two lists of size four and two, respectively. The result tuple has length two, based on the shortest list **list2**.

Example:

```
#creating two lists
List1 = ['A', 'B', 'C', 'D']
List2  = [100, 200]
#combine using zip
Result = tuple(zip(List1, List2))
print (Result)
print (type(Result))
```

Output:

```
(('A', 100), ('B', 200))
<class 'tuple'>
```

The *zip()* function also works for more than two iterables, such as:

Example:

```
#creating three tuples
T1 = ('Amit', 'Sumit', 'Joy', 'Dev')
T2 = (50000, 20000, 25000, 80000)
T3=('HR', 'Clerk','IT', 'Manager')
#combine using zip
Result = tuple(zip(T1, T2, T3))
print (Result)
print (type(Result))
```

Output:

```
(('Amit', 50000, 'HR'), ('Sumit', 20000, 'Clerk'), ('Joy', 25000, 'IT'), ('Dev', 80000, 'Manager'))
<class 'tuple'>
```

Inverse *zip(*)* Function

The * operator can be used with the zip () function. It unzips the tuples into independent lists. In other words, the * operator with the zip () function unpacks a sequence into positional arguments. The below example shows the concept of unzipping tuples.

Example:

```
# using zip() and * operator to unpack sequences
# creating tuple
T1 = [('A', 100), ('B', 200), ('C', 300), ('D', 4000)]
# Original tuple
print ("Original tuple is : ", T1)
# using zip() and * operator to perform unzipping
T2,T3 = zip(*T1)
T4=zip(*T1)
# Printing modified tuple 1
print ("Modified Tuple1 : ", T2)
print (type(T2))
# Printing modified tuple 2
print ("Modified Tuple2 : ", T3)
print (type(T3))
# Printing modified tuple 3
print ("Modified Tuple2 : ", tuple(T4))
```

Output:

```
Original tuple is :  [('A', 100), ('B', 200), ('C', 300), ('D', 4000)]

Modified Tuple1 :  ('A', 'B', 'C', 'D')

<class 'tuple'>

Modified Tuple2 :  (100, 200, 300, 4000)

<class 'tuple'>

Modified Tuple2 :  (('A', 'B', 'C', 'D'), (100, 200, 300, 4000))
```

Tuple Sorting

We have an in-built function named *sorted()* in Python to sort the tuple elements. This function takes one argument as a parameter to sort tuples.

```
sorted (arg1)
```

The tuple is by default sorted in ascending order using the *sorted()* function. If we want to sort in descending order, we must set the parameter reverse to True.

```
sorted (arg1, reverse=True)
```

Example:

```
# Create a tuple
Tuple1 = (20, 4, 6, -1, 4, 7.8, 2.79, 5.90)
# Sorting in ascending order
TDefault=sorted(Tuple1)
# Sorting in descending order
TReverse=sorted(Tuple1, reverse=True)
# Printing original tuple
print ("Original tuple:", Tuple1)
# Printing sorted tuple- Ascending
print ("Sorted tuple:", TDefault)
# Printing sorted tuple- Descending
print ("Sorted tuple in reverse order:", TReverse)
```

Output:

```
Original tuple: (20, 4, 6, -1, 4, 7.8, 2.79, 5.9)

Sorted tuple: [-1, 2.79, 4, 4, 5.9, 6, 7.8, 20]

Sorted tuple in reverse order: [20, 7.8, 6, 5.9, 4, 4, 2.79, -1]
```

DICTIONARY

Another essential built-in data type in Python is referred to as the dictionary. Data can be saved there in the form of collections. dictionary usage is not as widespread as other data structures, such as arrays and lists. In Python, a dictionary is a collection that stores the data or value that is associated with a certain set of keys. The relationship between values and keys is known as a mapping relation. Hence dictionary entries in Python take into account mapping relations between items. Compared to lists and arrays, the operation of dictionaries is rather distinct. The collection items are saved in the list and may be retrieved using index positions. When we wish to retrieve a particular element in a list or array, it implies that we must do a lookup using that element's index number. Programmers are not interested in working with indexing structures, yet many applications call for a more versatile method in terms of how information is retrieved from the collection. As a result, dictionaries can offer a superior option. Some concrete examples might help illustrate the meaning behind the notion of dictionaries. If we want to find out the employee's phone number based on their name, or if we want to know the city's name based on the supplied pin code, we may use this. The result is an effective data structure referred to as a dictionary and followed mapping between different values and keys. This particular combination is referred to as a *key-value* pair in the realm of computer

programming. With this combination, we can retrieve any information or value (the name of a city) that is connected with a certain key (pin code). The basic structure of the dictionary, together with an example of a *key-value pair*, is depicted in Fig (**1**).

A dictionary is a hash or associative array in other computer languages. This illustrative example demonstrates the use of a dictionary to store city names with their respective zip codes.

{"Name": "A", "Course": "Python Programming", "Duration": 1, "Grade": 8}

dict= {'Name': 'A', 'Course':'Python Programming', 'Duration': 1, 'Grade': 8}

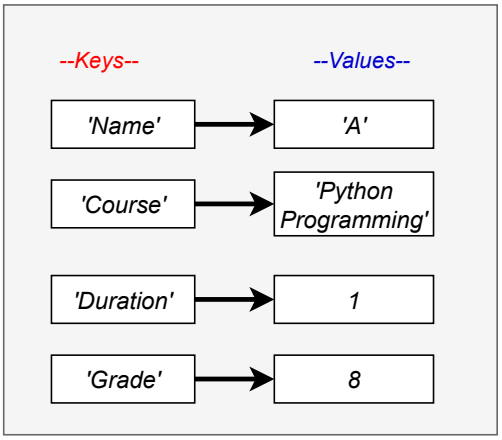


Fig. (1). Simple structure of a dictionary.

In a Python dictionary, all the elements are enclosed in curly brackets such as **{** and **}**. A colon (:) is used to join the *key-value* pair. The following is the syntax of the dictionary.

Syntax:

```
dictionary = {
   <key1>: <value1>,
   <key2>: <value2>,
    .
    .
    .
   <keyn>: <valuen>
}
```

Dictionary Creation

Python allows the creation of dictionaries by enclosing a sequence of elements in curly braces **{ }** and separated by commas **(,)**. The dictionary has several values, each corresponding to a certain key. Values can be of any data type and replicated in a dictionary, whereas keys cannot be copied and are immutable. Because of this, it's important to use the correct case when entering a dictionary key. A dictionary can be made in several different ways.

The first method of dictionary creation is to create an empty dictionary and add items.

Example:

```
# Creating an empty dictionary
Dict1 = {}
# Adding items to Dictionary
Dict1["key1"] = [1, 2, 3, 4, 5]
Dict1["key2"] = ["India", "US", "Australia","Japan","China"]
print (Dict1)
# Checking type of Dict1
print (type(Dict1))
```

Output:

```
{'key1': [1, 2, 3, 4, 5], 'key2': ['India', 'US', 'Australia', 'Japan', 'China']}
<class 'dict'>
```

Another way of creating a dictionary is by passing key-value pairs of literals.

Example:

```
# Creating a dictionary
Dict1 = {
    1 : "India",
    2 : "US",
    3 : "Australia",
    4 : "Japan",
    }
# Printing dictionary
print (Dict1)
    # Checking type of Dict1
    print (type(Dict1))
```

Output:

```
{1: 'India', 2: 'US', 3: 'Australia', 4: 'Japan'}
<class 'dict'>
```

We can create a dictionary by passing key-value pairs to dict() function.

Example:

```
# Creating a dictionary
Dict2 = dict(
    Name='David',
    Age=55,
    Address='Canada',
    Salary='$5000')
# Printing dictionary
print (Dict2)
# Checking type of Dict1
print (type(Dict2))
```

Output:

```
{'Name': 'David', 'Age': 55, 'Address': 'Canada', 'Salary': '$5000'}
<class 'dict'>
```

The next method of creating dictionaries is to pass the list of tuples to dict () function.

Example:

```
# Passing list of tuples
Dict1= dict(
    [('Name','William'),
    ('Age',40),
    ('Address','Japan'),
    ('Salary','$5000')])
# Printing dictionary
print (Dict1)
# Checking type of Dict1
print (type(Dict1))
```

Output:

```
{'Name': 'William', 'Age': 40, 'Address': 'Japan', 'Salary': '$5000'}
<class 'dict'>
```

Accessing Dictionary Values

Accessing any value in a dictionary is accomplished by using *key* names but accessing values in other forms of data is accomplished by indexing. To retrieve the values associated with a key, you can use the name of the key enclosed in square brackets []. The detailed syntax below may be utilized to obtain any value connected with the key in the dictionary.

Syntax:

```
Dictionaryname[keyname]
```

Example:

```
# Creating a dictionary
D1= {
    101 : "James",
    102 : "William",
    103 : "John",
    104 : "Sam",
     }
# Retrieving values associated with keys
print ('101:',D1[101])
print ('103:',D1[103])
print ('104:',D1[104])
```

Output:

```
101: James
103: John
104: Sam
```

Use the ***get()*** function with the key to acquire the corresponding value from the dictionary. This is an alternative method. The following are some possible applications of this function:

Example:

```
# Creating a dictionary
D1= {
    101 : "James",
    102 : "William",
    103 : "John",
    104 : "Sam",
     }
# Retrieving values associated with keys
print ('101:',D1.get(101))
print ('103:',D1.get(103))
print ('104:',D1.get(104))
```

Output:

```
101: James

103: John

104: Sam
```

Modifying Dictionary

As a mutable data type, dictionaries allow the element values to be altered whenever necessary. By combining an assignment operator with keys, we can either add new items or change the value of existing ones. If the key is already in use, then the value will be modified to reflect the addition of the new key. If the key does not already exist in the dictionary, it will be replaced with a new *key-value* pair. When adding a new entry to a dictionary, the syntax that is shown above should be used.

Syntax:

```
Dictionaryname[key]=value
```

Example:

```
# Creating a dictionary
D1= {
    101 : "James",
    'age' : 40,
    'city' : "Delhi",
     }
# Print the elements
print (D1)
# Add  a new element
D1['Post'] = 'Manager'
# After adding new element, print all values
print (D1)
# modifiying and esisting value
D1['age'] = 50
# After modifying value, print all values
print (D1)
```

Output:

```
{101: 'James', 'age': 40, 'city': 'Delhi'}

{101: 'James', 'age': 40, 'city': 'Delhi', 'Post': 'Manager'}

{101: 'James', 'age': 50, 'city': 'Delhi', 'Post': 'Manager'}
```

As we have seen in the above example, if a key (age) exists in a dictionary, then it modifies or replaces the old value (40) with the new value (50).

Delete an Element from the Dictionary

You can remove any element from the dictionary by utilizing the corresponding keys for that element. To remove a component from the dictionary that is currently being used, you can make use of the *"del"* operator. The disadvantage of utilizing this is that an exception will be thrown if the key is not there. This is something that should be considered before using it. The syntax that is presented below is what is utilized when removing an element from a dictionary.

Syntax:

```
del Dictionaryname[key]
```

The below example deletes an element such as age with the associated value from the dictionary D1.

Example:

```
# Creating a dictionary
D1= {
    101 : "James",
    'age' : 40,
    'city' : "Delhi",
     }
# Print the elements
print (D1)
# Add  a new element
D1['Post'] = 'Manager'
print ("Available Elements are:", D1)
# After adding new element, print all values
print ("Before Deletion:",D1)
# deleting an element age
del D1['age']
# Values after deleted an element
print ("After Deletion:", D1)
```

Output:

```
{101: 'James', 'age': 40, 'city': 'Delhi'}
Available Elements are: {101: 'James', 'age': 40, 'city': 'Delhi', 'Post': 'Manager'}
Before Deletion: {101: 'James', 'age': 40, 'city': 'Delhi', 'Post': 'Manager'}
After Deletion: {101: 'James', 'city': 'Delhi', 'Post': 'Manager'}
```

The del operator can even completely delete the dictionary, such as:

```
del D1
```

After deleting dictionary **D1**, if we try to access **D1,** it will cause an error because **D1** no longer exists. The *pop()* function is another approach that may be used to remove a key and the associated value from a dictionary. The benefit of using this instead of the del operator is that if you try to delete a non-existent element, it will output the value or message you want rather than fail. Second, in addition to finishing a fundamental delete operation, it also returns the value of a key that is being removed. This is in addition to the fact that it deletes the key.

Example:

```
# Initializing a dictionary
D1= {"John" : 30, "Sam" : 25, "james" : 28, "David" : 29, "Bob" :30}
# Printing dictionary before element deletion
print ("The dictionary before performing deletion is : " , D1)
# Using pop() function to delete an element such as John
del_value = D1.pop('John')
# Printing dictionary after element deletion
print ("The dictionary after deletion is : " , D1 )
print ("The removed key's value is : ", del_value)
# Again deleting an element which is not present
# assigns 'Not available' to del_value
del_value1 = D1.pop('Jai', 'Not available')
# Printing dictionary after element deletion
print ("The Dictionary after remove is : ", D1)
print ("The removed key's value is : " , del_value1)
```

Output:

```
The dictionary before performing deletion is :  {'John': 30, 'Sam': 25, 'james': 28, 'David': 29, 'Bob': 30}

The dictionary after deletion is :  {'Sam': 25, 'james': 28, 'David': 29, 'Bob': 30}

The removed key's value is :  30

The dictionary after remove is :  {'Sam': 25, 'james': 28, 'David': 29, 'Bob': 30}

The removed key's value is :  Not available
```

The *Clear ()* Function

By utilizing the *clear ()* function, we can remove all of the dictionary's entries simultaneously. The following example removes all items from the dictionary known as D1.

Example:

```
# Example of deleting all element using clear () function
# Initializing a dictionary
D1= {"Amit" : 20, "Sumit" : 25, "John" : 28, "David" : 29, "Bob" :30}
# Printing dictionary before element deletion
print ("The dictionary before performing deletion is : " , D1)
# Deleting all elements
D1.clear()
# Printing dictionary after element deletion
print ("The Dictionary after remove is : ", D1)
```

Output:

```
The dictionary before performing deletion is :  {'Amit': 20, 'Sumit': 25, 'John': 28, 'David': 29, 'Bob': 30}

The dictionary after remove is :  {}
```

Sorting Dictionaries

An element in a dictionary is made up of a key and the value that corresponds to it. The process of sorting entries in a dictionary may be carried out as a parameter using either the key or the value components. dictionary components can be sorted either by their keys or their values with the help of the *sorted ()* method.

Example:

```
# Initializing a dictionary
D1= {"James" : 50, "William" : 30, "John" : 25, "David" : 60, "Bob" :20}
print (type(D1))
# Printing dictionary before sorting
print ("The original dictionary is: " , D1)
# Sorting by key
D2= dict(sorted(D1.items(), key=lambda x: x[0]))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
<class 'dict'>
The original dictionary is:  {'James': 50, 'William': 30, 'John': 25, 'David': 60, 'Bob': 20}
After sorting by key:  {'Bob': 20, 'David': 60, 'James': 50, 'John': 25, 'William': 30}
<class 'dict'>
```

In this case, the *D1.items()* method produces a list of tuples that each include the keys and their corresponding values. The lambda function will return the key for a certain item tuple when it is called (0^{th} element of the dictionary). After these have been given to the *sorted()* method, the latter will produce a sorted sequence, which will then be entered into a dictionary.

This method may be used in Python versions 3.6 and later since it similarly considers ordered sequences and dictionaries. In previous versions, we may replace the lambda function located in the operator module with the *itemgetter()* described in the following example.

Example:

```
1.  from operator import itemgetter
2.  # Initializing a dictionary
3.  D1= {"James" : 50, "William" : 30, "John" : 25, "David" : 60, "Bob" :20}
4.  print (type(D1))
5.  # Printing dictionary before sorting
6.  print ("The original dictionary is: " , D1)
7.  # Sorting by key
8.  D2= dict(sorted(D1.items(), key=itemgetter(0)))
9.  print ("After sorting by key: ", D2)
10. print (type(D2))
```

Output:

```
<class 'dict'>
The original dictionary is:  {'James': 50, 'William': 30, 'John': 25, 'David': 60, 'Bob': 20}
After sorting by key:  {'Bob': 20, 'David': 60, 'James': 50, 'John': 25, 'William': 30}
<class 'dict'>
```

Sorting Dictionary by Value

It is a good analogy to ordering a dictionary by its keys rather than its values. The

sole difference is that the value component of this kind's corresponding element will serve as the parameter on which the sorting will be carried out. The dictionary is arranged in descending order by the value argument in the following example:

Example:

```
# Initializing a dictionary
D1= {"James" : 50, "William" : 30, "John" : 25, "David" : 60, "Sam" :20}
print (type(D1))
# Printing dictionary before sorting
print ("The original dictionary is: " , D1)
# Sorting by value
D2= dict(sorted(D1.items(), key=lambda x: x[1]))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
<class 'dict'>

The original dictionary is:  {'James': 50, 'William': 30, 'John': 25, 'David': 60, 'Sam': 20}

After sorting by key:  {'Sam': 20, 'John': 25, 'William': 30, 'James': 50, 'David': 60}

<class 'dict'>
```

In the above example, the dictionary D1 is sorted according to the values returned by the lambda function (x [1] value for x element). We use the following code for the older versions of Python.

Example:

```
from operator import itemgetter
# Initializing a dictionary
D1= {"James" : 50, "William" : 30, "John" : 25, "David" : 60, "Sam" :20}
print (type(D1))
# Printing dictionary before sorting
print ("The original dictionary is: " , D1)
# Sorting by value
D2= dict(sorted(D1.items(), key=itemgetter(1)))
print ("After sorting by key: ", D2)
print (type(D2))
```

Output:

```
<class 'dict'>
The original dictionary is:  {'James': 50, 'William': 30, 'John': 25, 'David': 60, 'Sam': 20}
After sorting by key:  {'Sam': 20, 'John': 25, 'William': 30, 'James': 50, 'David': 60}
<class 'dict'>
```

Sorting Dictionary in Reverse Order

Another input called reverse is taken into consideration by the *sorted()* method. The sequence in which the sorting should be carried out may be determined using this information, which can be used as a guide. If *True* is passed, the sorting will be done in the other direction, which is called descending order. In addition, if the default value of *False* is used, the sorting will be carried out ascendingly.

Example:

```
1.  # Initializing a dictionary
2.  D1= {"James" : 50, "William" : 30, "John" : 25, "David" : 60, "Sam" :20}
3.  # Printing dictionary before sorting
4.  print ("The original dictionary is: " , D1)
5.  # Sorting by key in reverse order
6.  D2= dict(sorted(D1.items(),reverse=True, key=lambda x: x[0]))
7.  print ("After Reverse order Sorting by key: ", D2)
8.  # Sorting by value in reverse order
9.  D3= dict(sorted(D1.items(),reverse=True, key=lambda x: x[1]))
10. print ("After Reverse order Sorting by value: ", D3)
```

Output:

```
The original dictionary is:  {'James': 50, 'William': 30, 'John': 25, 'David': 60, 'Sam': 20}
After Reverse order Sorting by key:  {'William': 30, 'Sam': 20, 'John': 25, 'James': 50, 'David': 60}
After Reverse order Sorting by value:  {'David': 60, 'James': 50, 'William': 30, 'John': 25, 'Sam': 20}
```

Iterating Over a Dictionary

Although the dictionary itself is not an iterable object, the methods *items(), keys(),* and values produce iterable view objects that may be used for iterating through the dictionary's entries. We can use a for loop to go through all the data's key-

value pairs. Using the items() function will return a list of tuples, with each tuple consisting of a key and value pair.

Example:

```
# Creating a dictionary
Dict1= {0 : 'Zero', 1 : 'One', 2 : 'Two', 3 : 'Three', 4 :'Four', 5: 'Five'}
# Printing dictionary
print ("The original dictionary is: " , Dict1)
# Iterating with for loop
print ('Iterating with for loop:')
for t in Dict1.items():
    print (t)
```

Output:

```
The original dictionary is:  {0: 'Zero', 1: 'One', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'Five'}

Iterating with for loop:

(0, 'Zero')

(1, 'One')

(2, 'Two')

(3, 'Three')

(4, 'Four')

(5, 'Five')
```

We can also iterate the dictionary by storing the key and value out of each pair in two separate variables such as.

Example:

```
# Creating a dictionary
Dict1= {0 : 'Zero', 1 : 'One', 2 : 'Two', 3 : 'Three', 4 :'Four', 5: 'Five'}
# Printing origional dictionary
print ("The original dictionary is: " , Dict1)
# Iterating with for loop and storing key and value seperatelhy
print ('Iterating with storing separate values:')
for key,value in Dict1.items():
    print (key,value)
```

Output:

```
The original dictionary is:  {0: 'Zero', 1: 'One', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'Five'}
Iterating with storing separate values:
0 Zero
1 One
2 Two
3 Three
4 Four
5 Five
```

Another way of dictionary iteration is to use the *key()* function. Using *key()* function, the associated value can be obtained as follows.

Example:

```
1. # Creating a dictionary
2. Dict1= {0 : 'Zero', 1 : 'One', 2 : 'Two', 3 : 'Three', 4 :'Four', 5: 'Five'}
3. # Printing origional dictionary
4. print ("The original dictionary is: " , Dict1)
5. # Iterating with key() function
6. print ('Iterating using key() function:')
7. for k1 in Dict1.keys():
8.     print (k1, Dict1.get(k1))     #or  print(k1, Dict1[k1])
```

Output:

```
The original dictionary is:  {0: 'Zero', 1: 'One', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'Five'}
Iterating using key() function:
0 Zero
1 One
2 Two
3 Three
4 Four
5 Five
```

Nested Dictionaries

A dictionary is a collection of components that are not arranged in any particular sequence, as we have seen previously in this lesson. It has a variety of components in the form of *key-value* pairs that are enclosed in curly brackets. The term "nested dictionary" refers to a situation in which one dictionary is kept inside another dictionary. It works much the same way that nested records and nested structures do in other programming languages. To put it another way, we may argue that it is a compilation of several dictionaries into a single, unified dictionary.

Syntax:

```
Nested_Dict1 = {
        'Dict1': {'key1': 'value1'},
         'Dict2': {'key1': 'value1'},
        'Dict3': {'key1': 'value1'}
                }
```

Example:

```
1.  # Nested dictionary
2.  Emp = {'e1' : {'name': 'James', 'age': '50', 'Post': 'Manager'},
3.        'e2' : {'name': 'william', 'age': '45', 'Post': 'Clerk'},
4.        'e3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
5.           }
6.  # Printing complete nested dictionary
7.  print (Emp)
8.
9.  print ('-----Details of e1 dictionary-------')
10. print (Emp['e1']['name'])
11. print (Emp['e1']['age'])
12. print (Emp['e1']['Post'])
13.
14. print ('-----Details of e2 dictionary-------')
15. print (Emp['e2']['name'])
16. print (Emp['e2']['age'])
17. print (Emp['e2']['Post'])
18.
19. print ('-----Details of e3 dictionary-------')
20. print (Emp['e3']['name'])
21. print (Emp['e3']['age'])
22. print (Emp['e3']['Post'])
```

Output:

```
{'e1': {'name': 'James', 'age': '50', 'Post': 'Manager'}, 'e2': {'name': 'william', 'age': '45', 'Post': 'Clerk'}, 'e3': {'name': 'David', 'age': '35', 'Post': 'HR'}}
-----Details of e1 dictionary-------
James
50
Manager
-----Details of e2 dictionary-------
william
45
Clerk
-----Details of e3 dictionary-------
David
35
HR
```

The following code snippet demonstrates how to extend a nested dictionary with a new sub-dictionary. If we want to create a new dictionary, we need to access the element by its key and provide it with a corresponding value.

Example:

```
1.  Emp = {'e1' : {'name': 'James', 'age': '50', 'Post': 'Manager'},
2.        'e2' : {'name': 'william', 'age': '45', 'Post': 'Clerk'},
3.        'e3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
4.           }
5.  # Printing dictionary elements before adding a sub dictionary
6.  print ('Nested Dictionary before adding a sub dictionary:', Emp)
7.  # Adding a sub dictionary
8.  Emp['e4'] = {'name': 'Bob', 'age': '32', 'Post': 'Manager'}
9.  # Printing dictionary elements after adding a sub dictionary
10. print ('Nested Dictionary after adding a sub dictionary:', Emp)
```

Output:

```
Nested dictionary before adding a sub dictionary: {'e1': {'name': 'James', 'age': '50', 'Post': 'Manager'}, 'e2': {'name': 'william', 'age': '45', 'Post': 'Clerk'}, 'e3': {'name': 'David', 'age': '35', 'Post': 'HR'}}

Nested dictionary after adding a sub dictionary: {'e1': {'name': 'James', 'age': '50', 'Post': 'Manager'}, 'e2': {'name': 'william', 'age': '45', 'Post': 'Clerk'}, 'e3': {'name': 'David', 'age': '35', 'Post': 'HR'}, 'e4': {'name': 'Bob', 'age': '32', 'Post': 'Manager'}}
```

Updating Nested Dictionary

It is simple to alter the dictionaries' nested parts. Changing the value of an object is as easy as referring to it by its key. The existing key's value is overwritten with the new one if it is found in the dictionary. The following code snippet shows how to modify the e1 dictionary values within the Emp dictionary.

Example:

```
# Nested dictionary
Emp = {'e1' : {'name': 'James', 'age': '50', 'Post': 'Manager'},
       'e2' : {'name': 'william', 'age': '45', 'Post': 'Clerk'},
       'e3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
           }
# Printing dictionary elements before update
print ('Nested Dictionary before update:', Emp)
# Updating values of D1
Emp['e1'] = {'name': 'Bob', 'age': '32', 'Post':'Director'}

# Printing dictionary elements after update
print ('Nested Dictionary after updating values:', Emp)
```

Output:

```
Nested dictionary before update: {'e1': {'name': 'James', 'age': '50', 'Post': 'Manager'}, 'e2': {'name': 'william', 'age': '45', 'Post': 'Clerk'}, 'e3': {'name': 'David', 'age': '35', 'Post': 'HR'}}

Nested dictionary after updating values: {'e1': {'name': 'Bob', 'age': '32', 'Post': 'Director'}, 'e2': {'name': 'william', 'age': '45', 'Post': 'Clerk'}, 'e3': {'name': 'David', 'age': '35', 'Post': 'HR'}}

>
```

Deleting Elements from a Nested Dictionary

We can use *del* operator to delete elements from the nested dictionary. The below

example deletes the value of age according to the given key (age) from the D2 dictionary.

Example:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
       'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
       'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
       }
# Printing dictionary elements before delete
print ('Nested Dictionary before delete:')
print (Emp)
# Deleting element from D2
del Emp['D2']['age']
# Printing dictionary elements after Delete
print ('Nested Dictionary after Deleting values:')
print (Emp['D2'])
```

Output:

```
Nested dictionary before delete:

{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D2': {'name': 'Joy', 'age': '25', 'Post': 'Clerk'}, 'D3': {'name': 'David', 'age': '35', 'Post': 'HR'}}

Nested Dictionary after Deleting values:

{'name': 'Joy', 'Post': 'Clerk'}
```

Deleting a Dictionary from the Nested Dictionary

We can delete a complete sub-dictionary from the nested dictionary using *del* operator same as elements. The below example deletes the D2-sub dictionary from the nested dictionary Emp.

Example:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
       'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
       'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
       }
# Printing dictionary elements before Deleting a sub dictionary
print ('Nested Dictionary before delete:')
print (Emp)
# Deleting  D2
del Emp['D2']
# Printing dictionary after Deleting a sub dictionary
print ('Nested Dictionary after Deleting D2:')
print (Emp)
```

Output:

```
Nested dictionary before delete:
{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D2': {'name': 'Joy', 'age': '25', 'Post': 'Clerk'}, 'D3': {'name': 'David', 'age': '35', 'Post': 'HR'}}
Nested Dictionary after Deleting D2:
{'D1': {'name': 'Amit', 'age': '30', 'Post': 'Manager'}, 'D3': {'name': 'David', 'age': '35', 'Post': 'HR'}}
```

Iterating Through a Nested Dictionary

We can traverse through each element in a nested dictionary using the *for* loops such as.

Example:

```
# Creating a nested dictionary
Emp = {'D1' : {'name': 'Amit', 'age': '30', 'Post': 'Manager'},
       'D2' : {'name': 'Joy', 'age': '25', 'Post': 'Clerk'},
       'D3' :{'name': 'David', 'age': '35', 'Post': 'HR'}
       }
# Iterating dictionary elements using for loop
for Dict_id, value in Emp.items():
    print ("\n Dict ID:", Dict_id)
    for key in value:
        print (key + ':', value[key])
    print ("------------------")
```

Output:

```
Dict ID: D1
name: Amit
age: 30
Post: Manager
----------
Dict ID: D2
name: Joy
age: 25
Post: Clerk
----------
Dict ID: D3
name: David
age: 35
Post: HR
----------
```

In the above example, the first loop returns all the keys in the nested dictionary **Emp**. It consists of the Dict Id of each employee. We have used these IDs to unpack the values of each employee. The second loop goes through the values of each employee. Then, it returns all of the keys' name, age, and post of each employee's dictionary. Then the keys of the employees and the associated values are printed.

Built-in Dictionary Functions

Python provides its users with a wide variety of built-in functions and methods that may be utilized for the purpose of carrying out a variety of actions on dictionaries. The following are some of the most significant functions:

len()	This function returns the total length of the dictionary elements. This would be equal to the number of elements in the dictionary. It can be used as follows:
str()	This function produces a printable string representation of a given dictionary. It has the following syntax.

cont.....

type()	It returns the type of the passed parameter. If the passed parameter is a dictionary, then it would return type as a dictionary.
cmp()	The comparison of two dictionary entries using their respective keys and values is handled by the cmp(). It will return either 1, 0, or -1 depending on whether or not the two dictionaries that are being compared are identical to one another. In order to do a dictionary comparison, this function requires two dictionaries as input parameters. The following result will be produced as a direct consequence of the comparison if D1 and D2 are two different dictionaries. This functionality is not available in Python 3.x.
any()	This function check whether any key is True or not. It takes an iterable parameter such as dictionary. If all keys are False or the dictionary is empty, it returns False. If at least one key is true, it returns True.
all()	When all elements are True in the given dictionary, this function returns True. If not, False will return it. In other words, If all keys are True or the dictionary is empty, True is returned by all() function. Else, like all other cases, this returns False.

Example:

```
# Built-in dictionary functions
# Creating a dictionary
D1= {"James" : 30, "William" : 25, "John" : 45, "David" : 40, "Bob" :25}
# Finding the length of the dictionary
print ("The length of dictionary is : " , len(D1))
# String representation of Dictionary
D2=str(D1)
print ("The String representation of dictionary is : " , D2)
print (type(D2))
print ('The type of D1 is:', type(D1))
#----------------------any()---------------------
print("-------------any()---------------")
D1 = {0: 'It is False'}
print (any(D1))
# 1 or True
D2 = {0: 'It is False', 1: 'It is True'}
print (any(D2))
# Both keys are false
D3 = {0: 'It is False', False: 'It is False'}
print (any(D3))
# Empty Dictionary
D4 = {}
print (any(D4))
# All are True
D5 = {True: 'It is True', 1: 'It is True'}
print (any(D5))
#-------------------------all()------------------
print("-------------all()---------------")
D1 = {0: 'It is False'}
print (all(D1))
# 1 or True
D2 = {0: 'It is False', 1: 'It is True'}
print (all(D2))
# Both keys are false
D3 = {0: 'It is False', False: 'It is False'}
print (all(D3))
# Empty Dictionary
D4 = {}
print (all(D4))
# 1 or True
D5 = {True: 'It is True', 1: 'It is True'}
print (all(D5))
```

Output:

```
The length of dictionary is :  5
The String representation of dictionary is :  {'James': 30, 'William': 25, 'John': 45, 'David': 40, 'Bob': 25}
<class 'str'>
The type of D1 is: <class 'dict'>
----------any()----------
False
True
False
False
True
----------all()----------
False
False
False
True
True
>
```

The *Copy ()* Method

The *copy ()* method returns a shallow copy of the dictionary. The original dictionary does not change it. The following example illustrates the use of *copy ()* method.

Example:

```
1. D1 = {'name': 'Python', 'year': 2023}
2. print ('Original Value of D1:', D1)
3. # creating a copy of D1
4. D2 = D1.copy()
5. print ('Copied Version :', D2)
```

Output:

```
Original Value of D1: {'name': 'Python', 'year': 2023}

Copied Version: {'name': 'Python', 'year': 2023}
```

Formatting Dictionaries

dictionary lookups allow us to perform an operation known as "interpolation." Because of the syntax they use, we need to enclose the key in parentheses and place it between the % operator and the character that represents the conversion. For example, if we want to format an int that is stored in a '**salary**' key to look like '**xxxx.xx**', we may use **% (salary).2f** in the spot where we want it to be presented. This will give us the desired result.

Example:

```
1. # Creating a dictionary
2. D1= {'name': 'Sam', 'salary': 30000, 'Post': 'Manager'}
3. # Printing dictionary
4. print (D1)
5. print ('--------------')
6. print ('----------Formated dictionary----------')
7. # Formatted printing
8. print ("My Name is: %(name)s" %D1)
9. print ("My Salary is: %(salary).2f and POST is: %(Post)s " %D1)
```

Output:

```
{'name': 'Sam', 'salary': 30000, 'Post': 'Manager'}

--------------

----------Formated dictionary----------

My Name is: Sam

My Salary is: 30000.00 and POST is: Manager
```

CONCLUDING REMARKS

This chapter introduced tuples and dictionaries and covered their fundamental ideas and characteristics. We also spoke about the many procedures that may be carried out on these sequences. The principles are explained logically with the help of several examples and Python code. You can learn to code by emulating these examples. Python's file-handling concepts will be discussed in the next chapter.

CHAPTER 7

File Handling

Abstract: From the previous chapters, we anticipate that the readers have garnered enough working knowledge of Python and its elementary concepts. In this chapter, we introduce to the user the concept of files. These are one of the major built-in object types in Python. We can create, call, work, and close the files using several functions as enunciated below. One of the primary tasks of files is method exporting and common file-processing tasks such as input and output display to external files, flush buffers, *etc*.

Keywords: Buffering, Directory path in OS, Storage in OS.

INTRODUCTION

When information must be stored in a file in an unalterable manner, file management is essential. A file is a designated location on the disc where data is kept that is important to the purpose of the file. Once the program has been closed, we will be able to retrieve the information that was previously stored. The concept of file management has been adapted and implemented in various other languages. However, doing so can be difficult or time-consuming, depending on the language.

However, unlike other Python ideas, this one is not complicated and can be understood easily. You will learn various theories and procedures about the management of files in this chapter. In addition, we have illustrated a variety of file actions by using examples and programs to illustrate our points.

FILE

A collection of bytes that may be used to store information is called a file. This data is structured in a specific format, which might be anything from a straightforward text file to an intricate program executable, depending on its complexity. In the end, these byte files are changed into binary, consisting of 1s and 0s so that the computer can process them more quickly. Much like many other programming languages, Python can handle files, allowing users to read, write, and execute many other file-related operations. This functionality is known as file handling. It is essential to understand that Python processes files differently based

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

on whether they contain text or binary data. Each line of code is made up of a string of individual characters that, when combined, make up a text file. A special character terminates each line in a file referred to as the EOL (End of Line) character. Examples of EOL characters include the comma and the newline character. It signals to the interpreter that the line being read has finished and that a new line has begun. On most of today's file systems, files are broken up into three distinct parts:

- *Header:* details about the file's contents (file name, size, type, and so on).
- *Data:* the file's contents as written by the author or writer.
- *End of file:* a unique character that denotes the file's termination.

In programming, it is possible that a specific piece of input data must be created more than once. Occasionally, it is insufficient only to display data on the console. Large amounts of data may be presented. The console can only display a certain amount of data; as memory is volatile, it is difficult to restore programmatically created data frequently. The local file system, which is volatile and always available, is where we can store things if we need to. The file-handling capabilities of Python must be used for this. We can use our Python application to create, modify, read, and destroy files on the local file system, thanks to file management in Python. When accessing a file on an operating system, a file path is necessary. A string indicates the location of a file called a file path. There are three main sections in it:

- **Directory Path:** the location of a file or folder on a file system, separated by a forward slash (/) in Linux or Unix or a backslash (\) in Windows.
- **File Name:** the file's real name.
- **Extension:** it defines the file type.

Let's say you needed to open the T1.txt file, and the position you were in now was the same as the **location**. You must first travel to the **Location** folder, then to the **Folder1** directory, and then to the **T1.txt** file to access the file. The path of this file is "***Location/Folder1/T1.txt.***"

Open() Function

It must be open first to perform reading and writing operations on a file. To open a file in Python, the user must first create a file object associated with a physical file. In addition, the ***open()*** function is used to open a file in Python. Python's open() function takes two arguments: the file name and the access mode. The function returns a file object, which can be used for reading, writing, and other operations. The below syntax is used to open a file in Python.

Opening a file is a prerequisite for performing any action on it, including reading and writing. To access the contents of a file using Python, the user must first build a file object corresponding to the file's actual location. In Python, opening a file is also accomplished with the help of the ***open()*** function. Python's ***open()*** function requires two pieces of information before it can be called: the *file name* and the *access mode*. The method will return a file object that may be utilized for various tasks like reading, writing, and others. In Python, opening a file is accomplished with the syntax that is presented below:

Syntax:

```
File_object_Name = open(<file_name>, <access_mode>, <buffering>)
```

Various modes, such as read, write, and append, are available for accessing the files. The access mode to open a file is defined in Table **1**.

Table 1. File Access Modes.

R	The file is opened in read-only mode. The file pointer is present at the start. If no access mode is defined, the file is opened in this mode by default.
rb	It converts the binary file into a read-only mode. The file pointer is present at the start of the file.
r+	It opens the file for both reading and writing. The file pointer is present at the start of the file.
rb+	It opens the file in binary format for reading and writing. The file pointer is present at the start of the file.
W	It only allows you to write to the file. If a file with the same name already exists, it is overwritten; otherwise, it is created. The file pointer is present at the start of the file.
wb	It opens the file so it can only be written in binary format. If the file already exists, it is overwritten; otherwise, it generates a new one. The file pointer is present at the start of the file.
w+	It opens the file for both writing and reading. It differs from r+ in that it overwrites the previous file if one exists, while r+ leaves the previously written file alone. If no such file exists, it creates one. The file pointer is present at the start of the file.
wb+	It opens the file in binary format for both writing and reading. The file pointer is present at the start of the file.
A	The file is opened in append mode. If there is one, the file pointer is at the end of the previously written file. If no file with the same name exists, it creates a new one.
ab	It opens the file in binary format in append mode. The pointer is at the end of the file that was previously written. If no file of the same name remains, it produces a new binary file.
a+	It opens a file for both appending and reading. If a file exists, the file pointer stays at the end of it. If no file with the same name exists, it produces a new one.
ab+	It opens a binary file for appending and reading. The file pointer is already at the file's end.

Example:

```
1. obj = open("Myfile.txt","r") #Opening Myfile.txt in read mode
2.
3. if obj:
4.     print (File opened successfully.")
```

Output:

```
File opened successfully.
```

Close() Function

Since Python allows us to do any operation on the file by utilizing the file system that is presently open, it is recommended practice to close the file after all processes have been finished. After we have finished working with the file in every possible way, we will shut it by calling the ***close()*** function. Any data that has not yet been written is discarded when a file object's close() function is invoked.

Syntax:

```
Object_Name.close()
```

Example:

```
1. obj = open("Myfile.txt","r") #Opening Myfile.txt in read mode
2.
3. If obj:
4.     print (File opened successfully.")
5. obj.close()          # file will be closed
```

Output:

```
File opened successfully.
```

Write() Function

A string is written to a file using the ***write()*** function. To write text to a file, we must first open it with the open method and one of the access modes mentioned below:

- **w:** If the file already exists, it will be overwritten. The file pointer is located at the start of the file.
- **a:** The file will be appended to the current one. The file pointer is at the file's end. If no such file exists, it creates one.

Example:

```
1. # Open the file in write mode
2. file1 = open("file1.txt","w")
3. #Write contents in file
4. file1.write("Hello \n")
5. file1.write("My Python Programming book \n")
6. file1.write("Python is easy \n")
7. file1.write("Python is developers' choice \n")
8. #Close the file
9. file1.close()
```

Output:

You can see the file1 created and can open it to see the contents.

```
1 Hello
2 My Python Programming book
3 Python is easy
4 Python is developers choice
```

Writelines() Method

The ***writelines()*** method is a Python built-in method for writing a list of strings or list items to a file.

Syntax:

```
fileobject.writelines(list_of_items)
```

Example:

```
1. #Open the file in write mode
2. f1 = open("f1.txt","w")
3. #Add the list of contents to be added
4. l1 = ["My first Python book \n","Python is simple \n","File handling is easy\n"]
5. f1.write("Hello \n")
6. f1.writelines(l1)
7. f1.close()
8. #To read the contents of the file, open it
9. f1 = open("f1.txt","r+")
10.     #Select to read it
11. print(f1.read())
12. print()
```

Output:

```
Hello

My first Python book

Python is simple

File handling is easy
```

Writing Numbers to a File

Python reads files as strings by default, and the ***write()*** function expects only to write strings. If we want to write numbers to a file, we'll need to use the ***str()*** function to cast them as strings.

Example:

```
1.  a = range(1,30)
2.  f2 = "f2.txt"
3.  # Open file to write
4.  f2= open(f2, 'w')
5.  for num in a:
6.    f2.write(str(num) + " ")
7.  f2.close() #Close the file
8.  #To read the contents of the file, open it
9.  f2 = open("f2.txt","r+")
10. #Select to read it
11. print(f2.read())
12. print()
```

Output:

```
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
```

Read() Method

In Python, there are several ways to read a text file. The ***read()*** method can be used to retrieve a string that includes all the characters in a file.

Syntax:

```
File_Object.read()
```

Before reading a file, it must be opened in '**r**' (read) mode.

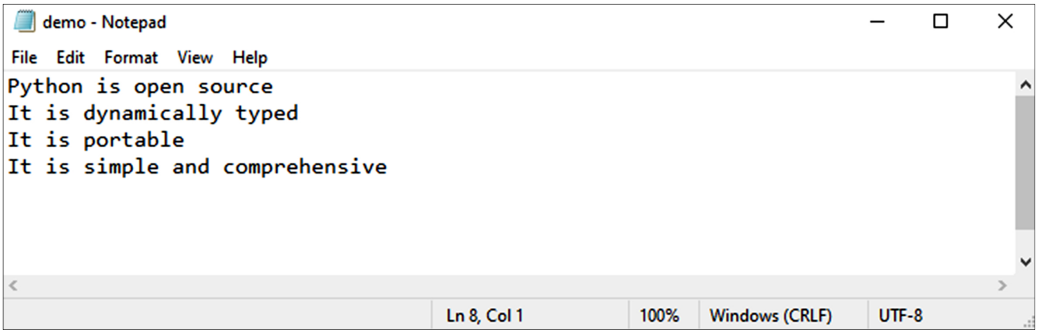


Fig. (1). Contents of demo.txt.

NOTE: For demonstrating the use of read(), readline() and readlines() commands, we will consider Fig. (**1**) as the specimen. This figure shows the file containing demo.txt that is retrieved using read commands.

Example:

```
1. #Open the file in read mode
2. f3 = open("demo.txt","r+")
3. print(f3.read())
4. print()
5. f3.close() # change the mode
```

Output:

```
Python is open source

It is dynamically typed

It is portable

It is simple and comprehensive
```

Another technique to read a text is limiting yourself to a particular number of characters. For instance, the interpreter will read the first twenty characters of the data that has been saved and will then return it as a string by utilizing the following code:

Example:

```
1. #Open the file in read mode
2. f4 = open("demo.txt","r+")
3. #Select to read it
4. print(f4.read(16))
5. print()
6. f4.close() # change the mode
```

Output:

```
Python is open s
```

Readline() Method

Python's ***readline()*** function makes it simple to read lines of text from a file in a sequential order. Since the ***readline()*** function reads lines from the file beginning with the first one, we will obtain the first three lines of the file if we call it three times, and it will start from the beginning. Every time it is invoked, this function will provide the calling code with a character string that contains a single line of data read from the file.

Example:

```
1. #Open the file in read mode
2. f5 = open("demo.txt","r+")
3. #Select to read it
4. print(f5.readline())
5. print()
6. f5.close()# change the mode
```

Output:

```
Python is open source
```

To retrieve only a specified number of characters, say four, specify the number of the line along with readline().

Example:

```
1. #Open the file in read mode
2. f6 = open("demo.txt","r+")
3. #Select to read it
4. print(f6.readline(4))
5. print()
6. f6.close() # change the mode
```

Output:

```
Pyth
```

Readlines() Method()

If we wanted to get back every line in the file separated adequately. The same function will be used but in a different format. The ***FileObject.readlines()*** method is used to accomplish this. It returns a list of lines up to the end of the file (EOF).

Example:

```
1. #Open the file in read mode
2. f7 = open("demo.txt","r+")
3. #Select to read it
4. print(f7.readlines())
5. print()
6. f7.close() # change the mode
```

Output:

```
['Python is open source\n', 'It is dynamically typed\n', 'It is portable\n', 'It is simple and comprehensive']
```

Reading Contents with a Loop

The loop-over approach may be utilized to read all of the lines from a file in a

quicker and more memory-efficient manner than any other method. One of the benefits of utilizing this strategy is that the code that corresponds to it is straightforward to understand. Fig. (**2**) shows the contents of file **demo2.txt** that we will retrieve using a loop.

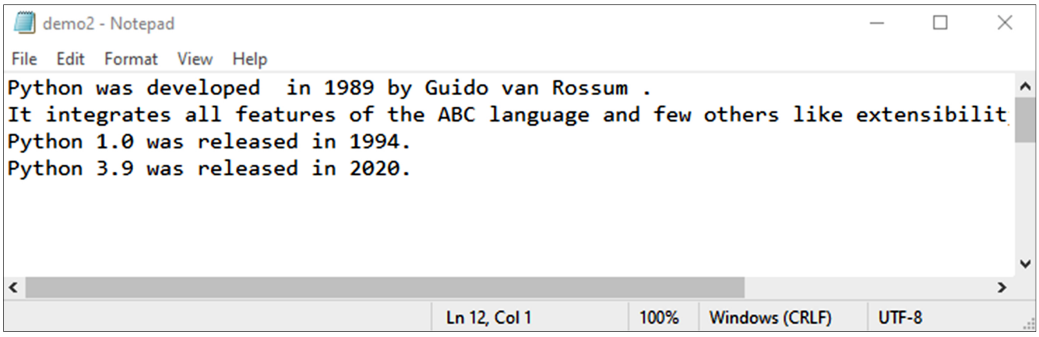


Fig. (2). Contents of demo2.txt.

Example:

```
1. #Open the file in read mode
2. f8 = open("demo2.txt", "r")
3. #Select to read it
4. for x in f8:
5.     print(x)
6. f8.close() # change the mode
```

Output:

```
Python was developed  in 1989 by Guido van Rossum .

It integrates all features of the ABC language and few others like extensibility and exception handling.

Python 1.0 was released in 1994.

Python 3.9 was released in 2020.
```

Creating a New File

With the help of the ***open()*** function, you can create a new file by using one of the following access modes:

- **x:** It creates a new file with the name you specify. It throws an error if a file with the same name already exists.

- **a:** If no such file exists, it creates a new one with the specified name. If the file with the given name already exists, it appends the content to it.
- **w:** If no such file exists, it creates a new one with the specified name. If a file exists, it overwrites.

Example:

```
1. #Creating a new empty file
2. fileobj= open ("file1.txt", "x")
3. #close this file
4. fileobj.close()
```

The above code will create a new file in the current working directory. If you want your file to be created at an absolute path on the disc, you can also give it a complete path.

Example: Creating a new file with an existing name

```
1. #Creating a new file, but myfile1.txt is already exist
2. fileobj= open ("file1.txt", "x")
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    fileobj= open("file1.txt", "x")
FileExistsError: [Errno 17] File exists: 'file1.txt'
```

In the above example, we tried to create a new file file1.txt. However, this file already exists in the same directory. We get FileExistsError.

File Object Attributes

You can get different file-related details once a file is opened and you have one file object. The following is a list of all file object attributes:

- **closed** – It is a bool value that indicates the current state of the file object.
- **encoding** - The encoding details of the file.
- **mode** – It returns the file mode of the file in which the file is opened.
- **name** – It returns file name.

Example:

```
1. import os.path
2. f10 = open("demo2.txt", 'a+')
3. #Specifying the required attributes
4. print("Status:",f10.closed)
5. print("Encoding style:",f10.encoding)
6. print("Mode:", f10.mode)
7. print("Name:",f10.name)
```

Output:

```
Status: False

Encoding style: UTF-8

Mode: a+

Name: demo2.txt
```

File Positions

When we use Python to open a file for reading, we are provided with a file handle that includes a reference to the beginning of the file. When we read from the file, the pointer will always point to the end of the reading, and the beginning of the following read will be at that location. The following methods are used, respectively, to acquire the current file position and alter the current file position:

- The **tell()** method returns the current file position; in other words, the next read or write will occur at that many bytes from the file's beginning.
- The **seek(offset[, from])** method is used to adjust the file's current location. The number of bytes to be moved is defined by the offset statement. The from statement determines the starting point for the bytes to be transferred from.

If from is set to **0** in the ***seek()*** function, the file's beginning will be used as the reference position, **1** the current position will be used as the reference position, and **2** the file's end will be used as a reference position.

Example:

```
1. # Open the file in write mode
2. f_1 = open("demo3.txt","w")
3. f_1.write("Python tools can be classified into two categories, general-
purpose tools, and functional tools.  \n")
4. f_1.writelines("Python has full-fledged support for a vast variety of built-in objects -
types.  \n")
5. f_1.writelines("Python considers memory as the private heap. \n")
6. f_1.writelines("Python community support is quite active and responsive  \n")
7. f_1.close() #to change file access modes
8. f_1= open("demo3.txt", "rb")
9. print("Position of pointer:", f_1.tell())
10.    #Reading first line
11.    l1= f_1.readline()
12.    print(l1)
13.    print("Position of pointer:", f_1.tell())
14.    #Reading second line
15.    l2= f_1.readline()
16.    print(l2)
17.    print("Position of pointer:", f_1.tell())
18.    #Reading third line
19.    l3= f_1.readline()
20.    print(l3)
21.    print("Position of pointer:", f_1.tell())
22.    #Reading fourth line
23.    l4= f_1.readline()
24.    print(l4)
25.    print("Position of pointer:", f_1.tell())
26.
27.    #changing the pointer's position
28.    #moving pointer 15 positions back from end of the file
29.    print("changed pointer position:", f_1.seek(-15,2))
30.    f_1.close()
```

Output:

```
Position of pointer: 0
Python tools can be classified into two categories, general-purpose tools, and functional tools.

Position of pointer: 99
Python has full-fledged support for a vast variety of built-in objects -types.

Position of pointer: 180
Python considers memory as the private heap.

Position of pointer: 226
Python community support is quite active and responsive

Position of pointer: 284
changed pointer position: 269
```

File Rename

Python's **OS** module offers several ways to carry out various file-processing tasks, one of which is renaming files. To utilize this module, you must first import it and then use any linked functions. The renaming of files and folders works in exactly the same way. The ***os.rename()*** method is used whenever a file's name must be changed.

Syntax:

```
os.rename(Previous_file_name, new_file_name)
```

Example:

```
1. import os
2. print ('original file',os.listdir(os.getcwd()))
3. old_name = r"/content/f1.txt"
4. new_name = r"/content/demonstration.txt"
5. # Renaming the file
6. os.rename(old_name, new_name)
7. print ('Rename:',os.listdir(os.getcwd()))
```

Output:

```
original file ['.config', 'f1.txt', 'f2.txt', 'demo3.bin', 'myfile.txt', 'demo.txt', 'demo2.txt', 'sample_data']

Rename: ['.config', 'demonstration.txt', 'f2.txt', 'demo3.bin', 'myfile.txt', 'demo.txt', 'demo2.txt', 'sample_data']
```

Deleting a File

os.remove() function is used to delete a specific file.

Syntax:

```
os.remove(file_name)
```

Example:

```
1. import os
2. print ('List of files',os.listdir(os.getcwd()))
3. # Specifying the path of file to be removed
4. os.remove("/content/f2.txt")
5. print ('Updated list:',os.listdir(os.getcwd()))
```

Output:

```
List of files ['.config', 'demonstration.txt', 'f1.txt', 'f2.txt', 'demo.txt', 'demo2.txt', 'sample_data']

Updated list: ['.config', 'demonstration.txt', 'f1.txt', 'demo.txt', 'demo2.txt', 'sample_data']
```

Binary Files

Text and binary files are the two types of files that may be used with Python. The bulk of the files that can be found on our computers are of the binary file type.

Utilizing Python's features makes working with binary files straightforward. Binary files make use of strings that are in the form of bytes. Reading binary data from a file indicates that you will receive an object of type byte in return. When working with binary files, we are required to make use of the identical modes (as before), but with the letter ***'b'*** added to the end in a way that will allow Python to understand that we are working with binary files. To write the binary file, the ***'wb'*** mode is utilized. Similarly, the ***'rb'*** and ***'ab'*** modes are utilized in binary files to read and add data, respectively.

Example:

```
num=[2,4,8,16,32,64,128]
arr=bytes(num)
print(arr)
f=open("demo3.bin","wb")
f.write(arr)
f.close()
#open binary file
f = open("demo3.bin", 'rb')
content = f.read()
content
```

Output:

```
b'\x02\x04\x08\x10 @\x80'
```

Directory Operations

A directory is a container for all connected files, folders, and documents. The OS module in Python provides tools for interacting with the operating system. The operating system-specific operations for controlling processes, files, file descriptors, directories, and other low-level OS features are accessible through this module.

Current Working Directory

getcwd() returns the current working directory's path. This is where the operating system converts a relative file name to an absolute file name.

Syntax:

```
#importing os module
import os
#print the current working directory
print (os.getcwd())
```

Output:

```
/home/user
```

List of Directories

We can get a directory list from a specific location. It would help if you used the ***listdir(location)*** function. The function will return a list of strings containing the names of the directories in the specified location if you pass it to the location. The simple **listdir()** function will return the contents of the current directory.

Example:

```
import os
data = os.listdir() #list of contents from current direcotry
print (data)

print (os.listdir('/usr')) # list of contents from  usr direcotry
```

Output:

```
['main.py']

['games', 'local', 'include', 'share', 'lib', 'bin', 'src', 'sbin']
```

Creating a Directory

The ***os.mkdir()*** method is used to create a directory. Let's make a new directory called "**mydir**" The os.listdir() method will print a list of directories along the path.

Example:

```
import os
 #create directory
os.mkdir("dir1")
print (os.listdir()) # list of contents
```

Output:

```
['dir1', 'main.py']
```

Change Directory

To change the directory, we must first import the os module and then use the ***os.chdir()*** method to change our program's base path.

Example:

```
1. import os
2. #change directory
3. os.chdir('/Users/folder1/')
4.
5. #print current working directory
6. print (os.getcwd())
```

Output:

```
/Users/folder1
```

Renaming a Directory

The ***os.rename()*** method allows you to rename a folder from one name to another.

Example:

```
1. import os
2. #rename directory
3. os.mkdir('mydir1')
4. print (os.listdir(os.getcwd()))
5. #rename
6. os.rename("mydir1","mydir2")
7. print (os.listdir(os.getcwd()))
```

Output:

```
['mydir1', 'main.py']

['mydir2', 'main.py']
```

Delete a Directory

The ***rmdir()*** function is used to remove an already empty directory. The directory

will not be removed if it is not empty.

Example:

```
1. import os
2. os.mkdir('mydir1')
3. print ('Before deletion:',os.listdir(os.getcwd()))
4.
5. #delete directory
6. os.rmdir("mydir1")
7. print ('After deletion:',os.listdir(os.getcwd()))
```

Output:

```
Before deletion: ['mydir1', 'main.py']

After deletion: ['main.py']
```

CONCLUDING REMARKS

In this chapter, we went through the ideas involved in Python file management. In addition, we spoke about the many different file modes that it may be opened in so that it can be processed. The processing of files, including reading, writing, deleting, renaming, and other operations, is demonstrated using a variety of examples and Python code. You should be able to develop your code for processing files based on these examples. In the next chapter, we will explore the notion of managing exceptions in Python and look at several instances.

CHAPTER 8

Exception Handling

Abstract: In this chapter, we introduce the concept of exception handling in Python. They are used to specify the alternate sequence of actions the program needs to jump to at the occurrence of the event. For example, if we want to print several pages from the printer and somewhere in the middle of the job the paper gets stuck in the printer. In such as situation we would want to jump to the function that aborts the printing and handles this situation by instant shut down of the printer. In such events comes the exception handling. When the program jumps to the exception handler part the current sequence of commands is abandoned and the commands given to the exception handler are executed. After the exception is tackled the programming returns to the point where the marker left.

Keywords: Error handling, Event notification.

INTRODUCTION

Error handling makes your code more robust by shielding it from the kinds of mistakes that may result in an abrupt shutdown of your application. On the other hand, Python exceptions can be handled in contrast to errors. Errors can be syntax errors, and although various exceptions might happen during execution, they aren't always unusable. An error could be a syntax mistake. An application that is reasonable and well-designed should avoid the critical issues indicated by an Error.

In contrast, an application that is suitable and well-designed should try to capture the conditions that are indicated by an Exception. Programmers should avoid handling errors wherever possible since they are a form of uncontrolled exception that cannot be recovered from. An example of this type of error is the **ZeroDivisionError**. Think about what would happen if you produced code that was later used in production but still ended because of an error. Because the customer would be dissatisfied, handling the exception in advance and eliminating uncertainty is preferable. There are two different kinds of mistakes, the first being syntax errors and the second being exceptions. A syntax error will occur when the parser identifies a grammatical problem in your code. Syntax errors are also commonly referred to as parsing errors. To better understand it, let's look at an example.

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

Example:

```
1. #Syntax error example
2. x = 1
3. y = 2
4. z = x y
```

Output:

```
File "<string>", line 3

z = x y

^

SyntaxError: invalid syntax
```

The arrow in the report shows that the code parser ran into an error when executing the program. The failure may be traced back to the token before the arrow. Because it will output the file's name and the line number where the issue occurred, Python will handle a significant portion of the troubleshooting work for you when attempting to resolve errors of this type.

An error that arises as a result of the execution of a program is referred to as an exception. Exceptions are occurrences that do not obey a general rule, which is how non-programmers understand the term. When a statement or expression is executed, an error through the syntax may occur. Python's exceptions are faults that may be seen during the execution of the program but are not always catastrophic. An exception object is created whenever a Python program generates a runtime error. The program will be terminated unexpectedly and without warning if the code does not expressly handle the exception. In most cases, programs will disregard exceptions, which will lead to error messages such as the following:

Example:

```
1. # Example of type error
2. name= 'David'
3. age=45
4. age+name
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Another example of division by zero exception can be seen in the below example.

Example:

```
1. #Example of division by zero exception
2. 10 * (15/0)
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
```

Python exceptions can take several forms, indicated right next to the message they produce. For example, the types of exceptions that have just been discussed are **TypeError** and **ZeroDivisionError**. Both error messages include, the kind of exception, and the name of the Python built-in exception that was encountered. The remaining half of the error line comprises information about what led to the mistake; the specifics of this information are determined by the type of exception that was thrown.

HANDLING EXCEPTIONS

Python's approach to managing exceptions is similar to Java's. The code that might result in the throwing of an exception is contained within a **try block**. In Java, exceptions are handled using catch clauses, but in Python, exceptions are handled by sentences inserted using the **except** keyword. Personalized deviations from the norm can be accommodated if necessary. By utilizing the "raise" command, it is possible to coerce the occurrence of an exception. You may safeguard your application by enclosing any potentially malicious code that can cause an exception in a try: block. This will prevent the exception from being thrown. Place an except declaration after the try: block, then immediately after that, a block of code that fixes the problem in the most elegant way possible should follow it. The following is an example of the syntax for **try...except...else** blocks:

Syntax:

```
try:
        # Staement that may generage exception
 except Exception_1:
        #If Exception_1 occurs, this block will be executed.
except Exception_2:
        # If Exception_2 occurs, this block will be executed.
except Exception_3:
        # If Exception_3 occurs, this block will be executed.
else:
        # If No exception occurs, this block will be executed.
```

The main components of exception handling are shown in Table **1**.

Table 1. Exception handling keywords.

try	It will execute the code block in which you expect an error.
except	It catches the exception, which occurs in a try block.
else	This block of code will be executed if there are no exceptions.
finally	This block of code will always be executed, regardless of whether there is an exception.

Below is an example of a ValueError exception.

Example:

```
1. #Example of ValueError exception
2. try:
3.     myvar = input ("Enter any Number here: ")
4.     myvar = int(myvar)
5.     print(myvar)
6. except ValueError:
7.         print("Exception occurs")
8. else:
9.     print('No exception occurred')
```

Output:

```
------------------------------Run program with following input------------------------------
Enter any Number here: 10
10
No exception occurred
------------------------------Run program with following input------------------------------
Enter any Number here: Python
Exception occurs
```

EXCEPT BLOCK

A single except statement can be used with a try block. This block can catch all types of exceptions. In this type of exception handling, the programmer cannot find the actual reason for the problems.

Example:

```
1. try:
2.     x = 15
3.     y = 0
4.     ans = x/y
5.     print(ans)
6. except:
7.     print("Error: Division by 0 not possible.")
```

Output:

```
Error: Denominator cannot be 0.
```

One **try** block can have many exception blocks attached to it at the same time. Nevertheless, only one of the exemption provisions will be put into effect. Let's look at an example written in Python that uses numerous except blocks in conjunction with a single try block. When the interpreter encounters an exception, it investigates the try block's associated except blocks first. It is possible to define the categories of exceptions that these except blocks will handle. When the interpreter comes across a comparable exception, it will execute unless blocked.

Example:

```
try:
  x1 = int(input("First number:"))
  x2 = int(input("Second number:"))
  sum = x1 + x2
  div =  x1 / x2
  print(sum)
  print(div)
except ZeroDivisionError:
  print("Division by zero")
except ValueError:
  print("Error: Enter an integer only")
```

Output:

```
First number:45

Second number:f

Error: Enter an integer only
```

We can also handle multiple exceptions with a single except block using parentheses to do this. The interpreter will return a syntax error if this is not done.

Example:

```
def test(x):
  try:
    x1 = int(x)
    if x1 > 60:
      print("Valid age")
  except TypeError:
    print("x is invalid..")
  except ValueError:
    print("x can only be integer")
  except:
    print("Unexpected error")

----Run the program with following set of outputs---------
x = [3.5,4]
test(x)

-----Run the program with following set of outputs--------
x= 85
test(x)
```

Output:

```
x is invalid...


Valid age
```

ELSE AND FINALLY, KEYWORD

Python's try and except blocks, together with the keywords else and finally, can be used in conjunction with one another. If an error occurs in the try block while the program is being performed, the except block will be carried out instead. However, if the try block does not produce any exceptions, the code in the else block will be carried out. In the end, another statement is utilized in the exception-handling process. Regardless of whether an exception is thrown, the final block's contents are always carried out in their entirety. As a consequence, the error-free try block advances directly to the finally block without running the except clause. It then continues to execute the remaining code.

Syntax:

```
try:
    #Statements
except:
    # Statements, if an exception occurs
else:
    #Statements, if no exception occurs
finally:
    #This code will always run
```

Example:

```
1.  def test_except():
2.    try:
3.      a= int(input("Enter input 1:"))
4.      b= int(input("Enter input 2:"))
5.      add= a - b
6.    except TypeError:
7.      print("x is invalid..")
8.    except ValueError:
9.      print("x can only be integer")
10.   else:
11.     print(add)
12.   finally:
13.     print("exception tested")
14. test_except()
```

Output:

```
Enter input 1:4

Enter input 2:5.05

x can only be integer

exception tested
```

RAISING EXCEPTIONS

Python also has the raise keyword, which may be used for managing exceptions in various situations. It causes an explicit exception to be thrown as a consequence. Errors that are built into the system will be raised automatically. However, when executing the code, you can compel either a built-in or a custom exception. The raise statement can be used in the following manner: its syntax is detailed below:

Syntax:

```
raise Exception_class (<value>)
```

Example:

```
details =  {'Languages':'Python', 'code':'C70458','enrolled': 75}
key= input("Search:")
try:
  print(f'{key} is {details[key]}')
  if key not in details:
    raise KeyError()
except  KeyError:
    print("Key used is invalid")
```

Output:

```
Search:name

Key used is invalid

Search:code

code is C70458
```

The below code is also an example of raising an exception. Here ArithmeticError class is used with the raise keyword.

Example:

```
def airth_err(x,y):
    try:
        print(x/y)
    except ZeroDivisionError as e:
        print("work complet e)
    else:
        print("No error")


airth_err(10,5)
airth_err(50,0)
```

Output:

```
2.0

No error

division by zero
```

Built-in Exceptions

When errors occur in Python, several built-in exceptions are raised. The ***local()*** built-in functions can be used to display these built-in exceptions as follows:

Example:

```
1.    print (dir(locals()['__builtins__']))
```

Output:

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError',
'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError',
'__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__',
'__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint',
'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'execfile', 'filter', 'float',
'format', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round',
'runfile', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Some important built-in exceptions are described in below Table **2**:

Table 2. Built-in exceptions.

Name	Condition
AssertionError	when an assertion fails.
AttributeError	when attribute assignment or reference fails
EOFError	when an end-of file is witnessed by input() function
FloatingPointError	when there are floating point errors
GeneratorExit	on the call of close() operation of generator
ImportError	when the module imported is not found
IndexError	At the out-of-bound index
IndentationError	Raised at an erroneous indentation
KeyError	When the key is not present in the dictionary
KeyboardInterrupt	Raised at the interruption of the execution by the user
MemoryError	Raised when the action is out of memory
NameError	Raised when the named variable is not present in the local or global scope
NotImplementedError	Raised by abstract methods
OSError	Raised when there occur OS-related failures
OverflowError	Raised when the result of the operation is too large to be expressed
ReferenceError	At the use of a weak reference proxy to access a garbage collection, this error is raised
RuntimeError	When the error is out of the scope of any other category
StopIteration	When the iterator reaches the end and there are no more items to return, the next() function raises this error
SyntaxError	Raised at the occurrence of syntax mistakes
SystemError	Raised at the occurrence of internal error
SystemExit	Raised by sys.exit() function.
TabError	Raised when there is a mix of erroneous tabs and space.
TypeError	Raised when the function is applied to the wrong object type.
UnboundLocalError	Raised when there is a reference to a local variable that the value has not bounded
UnicodeError	At the occurrence of Unicode-related encoding and decoding
UnicodeEncodeError	When the Unicode-related error occurs during encoding
UnicodeDecodeError	When the Unicode-related error occurs during decoding
UnicodeTranslateError	When a Unicode-related error occurs during translation
ValueError	Raised when the function receives the argument of valid type but the wrong value
ZeroDivisionError	Raised at the division or modulo with zero

User-defined Exceptions

You may create your own exceptions by creating classes from the built-in exceptions in Python. Programmers can invoke their exceptions by developing a new exception class. Inheritance from the Exception class is required for exceptions, either directly or indirectly. An illustration of a user-defined exception may be seen in the code below:

Example:

```
# define Python user-defined exceptions
class SeniorCitizenException(Exception):
    "Raised when age is less than 60"
    pass
age = 60
try:
    yr_age= int(input("Enter your age: "))
    if yr_age >= age:
      print("Senior citizen")
      print("Eligible")
    else:
      raise SeniorCitizenException
except SeniorCitizenException:
    print("Exception occurred: Not eligible")
```

Output:

```
------------------------------Running the program with following inputs------------------------------

Enter your age: 45

Exception occurred: Not eligible

Enter your age: 65

Senior citizen

Eligible
```

In the example above, we create a custom exception **SeniorCitizenException** defined in a class derived from the built-in Exception class. When the user's age is less than 60, an exception is generated, and the rest of the statements in the try block are skipped. The user-defined exception *SeniorCitizenException* activates the except block, and the statement inside this block is executed.

CONCLUDING REMARK

In this chapter, we discussed Exception handling in Python. We also discussed the uses of try, except, else, and finally blocks for managing exceptions. In the next chapter, we will discuss the concepts of Modules and packages.

CHAPTER 9

Modules and Packages

Abstract: Useful codes are often stored as separate files to increase modularity and reusability. Modules refer to a single file of code while a package is a collection of modules. A good programmer utilises both these aspects to enhance the program view and manage the hierarchy. In this chapter, we introduce the basics of working with the modules and packages.

Keywords: Scoping, Modular programming, Standalone script.

INTRODUCTION

MODULES

A complicated and unmanageable program can be broken down into several manageable subprograms, each referred to as a **module** through modular programming. One activity can be carried out by utilizing each component individually. The creation of modules in Python may be accomplished by utilizing Python files, which can include a variety of *and* statements. It is possible to define variables, classes, and functions in a module. A module can also make use of code that is executable. When the code is divided into modules, it is much simpler to comprehend and more convenient to utilize. Additionally, it logically arranges the code.

To put it another way, the file containing our Python source code with the extension (.py) is considered the module. Python modules can store code that can be executed. When we want to use the features of one module in another, we must import the specific module first. Let's say you've generated a file on your computer named **module1.py**, and inside it, you have the following code:

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

Filename: module1.py

```
#(module1.py)
def add_value (x, y):
 return (x+y)
def sub_value (x, y):
 return (x-y)
```

To call the functions *add_value()* and *sub_value()* specified in the module named file (**module1.py**), we must include this module in our main module. To use the module's functionality, we must first load it into our Python code. Python has ***import*** and ***from..import*** statement to include a module.

Import Statement

Our Python program can connect to a module by utilizing an import statement. The import of many modules may be accomplished with a single import line; however, despite the number of times a module has been imported into our register, it is only loaded once each time. The syntax of the import statement is as follows:

Syntax:

```
import module1, module2, module3, ..... module n
```

When an import statement is found the interpreter imports the module specified within the search path. The interpreter explores every directory in the search path when importing a module. For instance, add the following line to the program's top to import the module **module1.py**.

Example:

```
# importing module module1.py
import module1
print (add_value (2, 4))
print (sub_value (10, 50))
```

Output:

```
6

60
```

From…Import Statement

Python enables users to import only a module's specified properties into the namespace rather than the entire module. For this, the from…import statement may be utilized. The following syntax makes use of the ***from...import*** expression.

Syntax:

```
from <module_name> import <name_1>, <name_2>….<name_n>
```

Consider the following module, **module 1**, which includes the functions *add_value()* and *sub_add()*.

Example:

Filename: module1.py

```
1.  # module1.py
2.  def add_value(x, y):
3.   return (x+y)
4.  def sub_value(x, y):
5.   return (x-y)
```

If we want to import only *add_value()* function from this module, then the following code will be used.

Filename: main.py

```
1.  # importing add_value()from module1.py
2.  from module1 import add_value()
3.  print (add_value(100, 200))
```

Output:

```
300
```

We can also import any built-in module in our program. The below example imports the **pi** function from the *math* module.

Example:

```
1.  #importing pi from the math module
2.  from math import pi
3.  print ("pi value=", pi)
```

Output:

```
pi value= 3.141592653589793
```

The **from...import** statement may be used if we know the properties that need to be imported from the module in advance. * may also be used to import all of a module's attributes, which won't make our code any larger. Look at the example below, where all *module1's* functions are imported.

Example:

Filename: main.py

```
1.  # importing all attributes from module1.py
2.  from module1 import *
3.  print (add_value(10, 20))
4.  print (sub_value(100, 50))
```

Output:

```
30

50
```

Renaming A Module

Python allows us to import modules with specific names and then use those names to refer to those modules in our source code. To rename a module in Python code ***as*** keyword is used. The syntax for renaming a module is as follows:

Syntax:

```
import <module-name> as <new-name>
```

Example:

```
1.  # importing module1.py
2.  import module1 as mm    # rename module1 to mm
3.  print (mm.sub_value(20, 10))
4.  print (mm.add_value(100, 20))
```

Output:

```
>>> dir(module1)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add_value',
'sub_value']
```

Dir() Built-In Function

The *dir()* function can be used to find names specified within a module. In module *module 1*, for example, we've specified two functions: *add_value()* and *sub_value()*. In the *module1* module, we can use *dir()* as follows:

```
>>> dir(module1)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add_value',
'sub_value']
```

Here, we can see a list of names that have been sorted (along with *add_value* and *sub_value*). The module's default Python characteristics may be identified by their names, beginning with an **underscore (_)** character. Finding all of the names that are specified in our current namespace may be accomplished by using the *dir()* function without passing any parameters to it.

Example:

```
> import math
> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'math']
```

Reload () Function

As stated earlier, a module is only loaded once, regardless of how often it is imported into the Python source file. However, Python gives us the *reload()* function to reload a module that has already been imported and rerun the top-level code. The following syntax takes advantage of the *reload()* method.

Syntax:

```
reload(<module-name>)
```

Example:

```
reload(module1)
```

Built-In Modules

In addition to built-in functions, libraries included with Python releases also offer many pre-defined functions. Built-in modules describe the tasks that are specified in modules. Built-in modules developed in C are incorporated into the Python shell. Resources for system-specific tasks like OS administration, disc I\O, and other similar tasks are included in each built-in module. The standard library also contains many Python scripts with the ***.py*** extension that provide practical utility. To view a complete listing of all available modules, the following command is used:

```
help('modules')
```

Some commonly used built-in modules include:

1. OS Module: It facilitates OS-related operations like, create folders, delete folder, fetch the contents, change, and identify the current directory.
2. Sys Module: It allows modification of different parts Python runtime environment.
3. Math Module: This module supports mathematical functions like trigonometry, angle conversion, logarithmic function, *etc*.
4. Statistics module: It supports the statistical operation over numeric data.
5. Collection Module: It provides an alternative to built-in container data types like lists, tuple, and dict. It supports OrderedDict(), nametuple(), deque(), *etc*.
6. Random Module: It is used to generate pseudo-random variables.

Example:

```
1.  import math
2.  #natural log
3.  print("natural log:", math.log(45))
4.  #standard log
5.  print("standard log:", math.log10(84))
6.  #exponential
7.  print("exponential:", math.exp(9))
8.  #power function
9.  print("power value:",math.pow(3,4))
10. #ceiling
11. print("ceiling function:", math.ceil(6.78954))
12. #floor
13. print("floor function:",math.floor(1.544847))
```

Output:

```
natural log: 3.8066624897703196

standard log: 1.9242792860618816

exponential: 8103.083927575384

power value: 81.0

ceiling function: 7

floor function: 1
```

There is a large number of built-in modules. The below example shows the uses of the datetime built-in module.

Example:

```
from datetime import date
# date object for current date
today = date.today()
print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)

# Datetime from timestamp
date_time = datetime.fromtimestamp(19745274674)
print("Datetime from timestamp:", date_time)
```

Output:

```
Current year: 2023

Current month: 1

Current day: 5

Datetime from timestamp: 2595-09-14 06:31:14
```

PACKAGE

To make finding and managing many files simpler, we classify them into different categories and subfolders depending on predetermined criteria. However, a module in Python takes the idea of modularity to its logical conclusion. As you may already know, a module can include many objects, including classes, functions, *etc*. In a package, one or more related modules may be present. A package is a folder that physically contains one or more module files. Similar to how discs and directories enable us to store data in an operating system, packages help us store additional sub-packages and modules that the user may utilize at any time.

Creating Package

We build a file called **__init__.py** within a directory to inform Python that it is a package, and then it is considered a package, and we can create other modules and sub-packages inside it. This **__init__.py** file may be left blank or filled in with the package's initialization code.

We can follow these three basic steps to build a package in Python:

1. First, we create a directory and name it a package, preferably one related to its function.
2. After that, we add the required classes and functions.
3. Finally, inside the directory, we construct an **__init__.py** file to inform Python that it is a package.

Example:

1. Make a directory called **Details**.

2. After that, we must build modules.

    a. Create a file called **program.py** and write the below code.

Filename: program.py

```
class program:
    def __init__(self):
        self.data1 = ['BTech', 'MSC', 'IMSc']

    def show(self):
        print("Program details:")

        for s in self.data1:
            print(s)
```

    b. Create a file called **course.py** and write the below code.

Filename: course.py

```
class course:
    def __init__(self):
        self.data2 = ['TOC ', 'ADA', 'CN']

  #function
    def show(self):
        print('course details)
        for s in self.data2:
            print(s)
```

    c. Create a file called **duration.py** and write the below code.

Filename: duration.py

```
class duration:
    def __init__(self):
        self.data3 = [4, 2, 5]
#function
    def show(self):
        print('duration')
        for s in self.data3:
            print(s)
```

3. Create **__init__.py** file in the **Details** directory and either leave it blank or fill it with the initialization code.

Filename: __init__.py

```
from program import program
from course import course
from duration import duration
```

4. Create a **Test.py** file in the same directory as the **Details** package and write the following code:

Filename: Test.py

```
#Import classes from the Details Package
from Details import program
from Details Import course
from Details import duration

# Create an object of program class
obj1 = program()
obj1.show()

# Create an object of course class
obj2 = course()
obj2.show()

# Create an object of duration class
obj3 =duration()
obj3.show()
```

Output:

```
Program details:
BTech
MSC
IMSc
course details
TOC
ADA
CN
duration
```

Fig. (**1**) depicts the hierarchy of modules in the package.

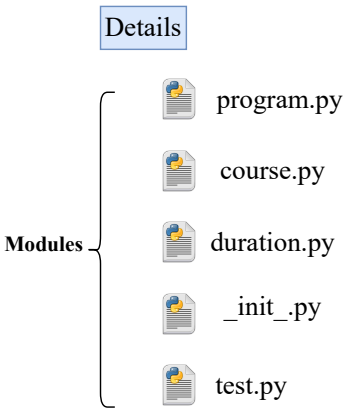


Fig. (1). Hierarchy of Modules in the Details package.

__init__.py

The content of the package is stored in a particular file named **__init__.py**, which is found in the package folder. It has two functions:

1. If a folder contains the *__init__.py* file, the Python interpreter recognizes it as a package.
2. *__init__.py* makes resources from its modules available for import.

When this package is imported, an empty *__init__.py* file makes all modules'

functions accessible. It is worth noting that the folder's *__init__.py* file is needed for Python to recognize it as a package.

Sub Packages

The package hierarchy detailed above can have additional nested packages within the same package. This structural hierarchy aids in program management as one can accentuate different modules properly. Further, the division into different modules reduces the length of the program and makes it easy to handle. Code modification requires less time as only the required module can be modified and not the whole program.

Fig. (**2**) is the specimen of the nested packages. The **Details** package comprises two other sub-packages namely, **Subpackage1** and **Subpackage2.** Each of these packages further encompasses the module files. As in the previous example, there are three modules namely, ***program.py, course.py, duration.py, _init_.py,*** and ***test.py.*** Each of these modules is further encapsulated in two sub-packages, rather than in a single main package. The package hierarchy doesn't complicate the import process much. Now for import purposes, you have to specify the name of the subpackage as well alongside the main package separated by the dot (.) as shown in the example.

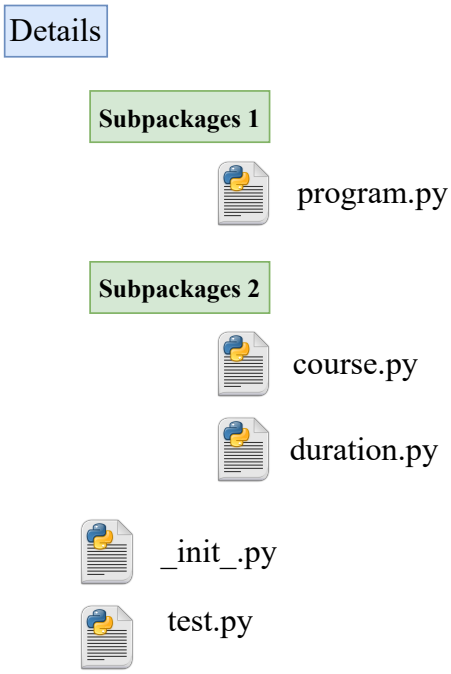


Fig. (2). Details package structure with sub-packages.

Example:

```
1. Import Details.Subpackage1.program
2. import Details.Subpackage2.course
3. import Details.Subpackage2.duration
```

CONCLUDING REMARK

We covered Python modules and packages in this chapter. Additionally presented are module implementations and their importing. Furthermore, Python codes are used to discuss package creation and services. You can write codes based on these examples. We will talk about Python's object-oriented programming concepts in the following chapter.

CHAPTER 10

Object-Oriented Programming

Abstract: In this chapter, we explore the OOP concepts in programming that offer an effective way of making coding more easy and comprehensive. It facilitates redundancy and allows customizing the existing code.

Keywords: Namespace, Superclass and Subclass.

INTRODUCTION

Object-Oriented Programming, sometimes called OOP, organizes a computer program's components into distinct objects with similar characteristics and functions. Classes and objects are the fundamental building blocks of object-oriented programming. The class serves as the blueprint, while the objects themselves are living, breathing entities capable of carrying out various operations. An object is composed of its data elements, characteristics, and behaviour, which may include actions or functions.

In procedural programming, a program is structured similar to a recipe by providing several stages, including functions and code blocks, that flow sequentially to achieve a goal. This programming paradigm is one of the most common programming paradigms. Python has always been object-oriented, much like other languages designed for general-purpose programming. It makes it possible for us to construct programs using an object-oriented methodology. Python makes it very simple to create and work with objects and classes. An object-oriented paradigm refers to building software by utilizing classes and objects. The item is connected to things in the real world, such as a computer, a house, a mobile phone, and so on. The definition of OOPS emphasizes the creation of code that may be reused. Putting together new things to use as solutions is a frequently utilized strategy. An object-oriented programming paradigm can be broken down into its core ideas, which are as follows.

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

CLASS AND OBJECT

A **class** is a name that can be given to a collection of objects. It is a logical entity with specific methods and unique properties. For example, a student's class should contain an attribute and a method, such as their name, age, address, and the classes they are enrolled in. The **object** functions as its independent entity, complete with a state and a set of behaviors. It may be a notebook, computer, pencil, or other things. In Python, everything is an object, and nearly anything may have both attributes and methods applied to it. When a class is defined, the class must first create an object before it may assign memory.

DATA ABSTRACTION

The practice of abstraction is a method that hides information on the system's internal workings and shows only its functionalities. Data encapsulation and data abstraction are two terms that are frequently used interchangeably. Because data encapsulation is the means through which data abstraction is achieved, it is possible to use either word interchangeably.

ENCAPSULATION

Encapsulation is a key notion in object-oriented programming, one of today's most prevalent programming paradigms. It explains the concept of enclosing the data and the methods that operate on the data within a single unit. This limits direct accessing methods and variables, which helps avoid unintentional data change. Only the object's method can change the object's variables; this is done to prevent unintentional changes. These particular variables fall under the category of “private variables.” Encapsulation can be illustrated by how a class stores all information about its member functions, variables, and so on. The objective of information hiding is to ensure that the state of an object is always valid by regulating access to its attributes while keeping those properties hidden from the view of the outside world.

INHERITANCE

Inheritance is the most fundamental component of object-oriented programming, miming the inheritance process that occurs in real life. Inheritance is an essential component. It states that all of the characteristics and behaviors of the parent object are passed down to the child object through inheritance. Through inheritance, we can build a class capable of taking on all of the characteristics and actions of another class. The new class is considered to be a derived class or a child class, whereas the base class or the parent class is regarded as the class

whose properties are gained. It assures that the code can be utilized in different applications.

POLYMORPHISM

The term "polymorphism" originates from the combination of the words "poly" and "morphs." The prefix poly denotes "many," while the suffix morph refers to "shape." The capacity to carry out a single activity in multiple guises is what we mean when discussing polymorphism. It employs a single category of items, such as a method, operator, or object, to stand in for several different types in various contexts. For example, we have a single addition operator capable of adding a wide variety of value types.

Defining a Class

The keyword **class**, followed by the class name, is used to build a class in Python. The following is the syntax for creating a class.

Syntax:

```
class <Class_Name>
        # data members
        # member functions
```

A class declares all its attributes in a new local namespace. Data members or functions may be included as attributes.

It also contains unique attributes that start with double underscores. For example, ***__doc__*** returns the class's docstring. The following statement can access it.

```
<class-name>. __doc__.
```

A new class object with the same name is generated when we define a class. We may use this class object to access the various attributes and create new objects of that class.

Example:

```
class Course:
    "The Python Programming course"
    lec= 30
    def fun(self):
        print('Lectures on Python')

print (Course.lec)
# call function
print (Course.fun)
# display docstring
print (Course.__doc__)
```

Output:

```
30

<function Course.fun at 0x7f8d97db5da0>

The Python Programming course
```

Creating an Object

Before we utilize a class's properties or methods in another class or function, we need to instantiate the class. Simply naming a class by its name is one way to initialize it. The following syntax is mentioned to generate a class instance (object).

Syntax:

```
<object-name> = <class-name>(<arguments>)
```

Example:

```
1.  class Course:
2.      'The Python Programming course'
3.      lec= 30
4.      enroll = 89
5.      def fun(self):
6.          print('No.of scheduled lectures', self.lec)
7.          print('Enrolled students', self.enroll)
8.
9.  detail= Course()
10. # Accessing  variables
11. print (detail.lec)
12. print(detail.enroll)
13. #Accessing function
14. print (detail.fun())
```

Output:

```
30

89

No.of scheduled lectures 30

Enrolled students 89
```

Deleting Properties or Object

Using the **del** keyword, we can delete the object's properties or the object itself. The below examples demonstrate the deletion of some properties and objects respectively.

Example:

```
1.  class Course:
2.      'The Python Programming course'
3.      lec= 30
4.      enroll = 89
5.      def fun(self):
6.          print('No.of scheduled lectures', self.lec)
7.          print('Enrolled students', self.enroll)
8.
9.  detail= Course()
10. # Accessing  variables
11. print("----------Before Deletion-----------")
12. print (detail.lec)
13. print(detail.enroll)
14. #Accessing function
15. print (detail.fun())
16.
17. del Course.lec
18. print("---------------After Deletion-------------------------------")
19. print (detail.fun())
```

Output:

```
----------Before Deletion-----------

30

89

No.of scheduled lectures 30

Enrolled students 89

---------------After Deletion-------------------------------

Traceback (most recent call last):

File "<string>", line 19, in <module>

  File "<string>", line 6, in fun

AttributeError: 'Course' object has no attribute 'lec'
```

The below example shows the deletion of an already-created object.

Example:

```
class Course:
    'The Python Programming course'
    lec= 30
    enroll = 89

    def fun(self):
        print('No.of scheduled lectures', self.lec)
        print('Enrolled students', self.enroll)

detail= Course()
# Accessing  variables
print("----------Before Deletion-----------")
print (detail.lec)
print(detail.enroll)
#Accessing function
print (detail.fun())

del detail
print("---------------After Deletion--------------------------------")
print (detail.fun())
```

Output:

```
----------Before Deletion-----------
30
89
No.of scheduled lectures 30
Enrolled students 89
None
---------------After Deletion--------------------------------
Traceback (most recent call last):
  File "<string>", line 19, in <module>
NameError: name 'detail' is not defined
```

Data Abstraction Example

The object-oriented programming paradigm places a significant emphasis on the abstraction of data. We also have the option of concealing Python data by appending a pair of underscores (__) to the beginning of the attribute that is to be concealed. After this, the attribute will no longer be accessible outside the class *via* the associated object. The idea of data abstraction is demonstrated with Python code down below:

Example:

```
1.  class Course:
2.      'The Python Programming course'
3.      lec= 30
4.      enroll = 89
5.      __year = 2023
6.      def fun(self):
7.          print('No.of scheduled lectures', self.lec)
8.          print('Enrolled students', self.enroll)
9.          print('Year', self.__year)
10.
11. detail= Course()
12. # Accessing  variables
13. print (detail.lec)
14. print(detail.enroll)
15. print (detail.year)
```

Output:

```
30

89

Traceback (most recent call last):

  File "<string>", line 15, in <module>

AttributeError: 'Course' object has no attribute 'year'
```

CONSTRUCTORS

A constructor is a particular type of function that is used to initialize the class's instance members. The constructor in C++ or Java has the same name as the class, but Python handles constructors differently. When we create an object of a class, the constructor function executes. In Python, the constructor is defined using the ***__init__()*** method. It takes the **self** keyword as a first argument, allowing access to the class's attributes and methods. Depending on the ***__init__()*** definition, we can pass any number of arguments when constructing the class object. It's mainly used to set up the class attributes. Every class must have a default constructor if we have not defined any other constructor.

Example:

```
1.  # Creating a class book
2.  class Course:
3.      #defining constructor
4.      def __init__(self,l,e,s):
5.          print ('Constructor example')
6.          self.lec = l
7.          self.enroll = e
8.          self.sub= s
9.
10.     # creating function
11.     def fun(self):
12.         print('No.of scheduled lectures:', self.lec)
13.         print('Enrolled students:', self.enroll)
14.         print('Year:', self.sub)
15.
16. # Object creation and passing arguments to constructor
17. d_1 = Course(30, 200, "Python")
18. d_2= Course(45, 150, "C++")
19. # Accessing class attribute and method through objects
20. print ('-----Values------')
21. d_1.fun()
22. d_2.fun()
```

Output:

```
Constructor example

Constructor example

-----Values------

No.of scheduled lectures: 30

Enrolled students: 200

Year: Python

No.of scheduled lectures: 45

Enrolled students: 150

Year: C++
```

When we do not want to update the value or have self as an argument, we use the non-parameterized constructor. The below code shows an example of the non-parameterized constructor.

Example:

```
1.  class demo:
2.   def __init__(self):   # default constructor
3.    self.stat = "Joe sat on the chair."
4.  # function for print
5.   def show(self):
6.    print(self.stat)
7.
8.  # Object for the class
9.  d1 = demo()
10.
11. # calling instane
12. d1.show()
```

Output:

```
Joe sat on the chair.
```

Parameterized Constructor

The parameterized constructor contains many parameters in addition to the self. The parameterized constructor is demonstrated in the Python code below:

Example:

```
class demo:
    # Parameterised constructor

    def __init__(self, arg1):
        print("This is parametrized constructor")
        self.arg1= arg1
    def show(self, adv):
        print("Joe went", self.arg1, adv)

d1 = demo("school")
d1.show("today")
```

Output:

```
This is parametrized constructor

Joe went school today
```

Default Constructor

The default constructor is used when no constructor is included in the class or is not declared. It defines the objects' initial values. The below example demonstrates the concept of a default constructor.

Example:

```
class test:
    chk= "undone"

    def assign(self):
        print(self.chk)
# object of class
v1 = test()
# calling the instance method
v1.assign()
```

Output:

```
undone
```

INHERITANCE IN PYTHON

Inheritance is an essential part of the object-oriented framework that we will be using. Inheritance enables us to recycle the program's code since we do not have to start from zero when creating new classes; instead, we can utilize an existing class as a foundation. The child class acquires access to all of the data members and functions described in the parent class, and it also inherits the attributes present in the parent class. A child class can have its take on how the parent class's functions should be implemented. Python may allow a derived class to inherit a base class by simply specifying the base class after the name of the derived class and surrounding it with brackets. Use the syntax indicated below in order to inherit properties and behaviours from a base class into a derived class.

Syntax:

```
class <derived-class_Name> (base-class_Name):
        # derived class body
```

A derived class can inherit properties from more than one base class. It is possible to do this form of inheritance by naming all base classes within the brackets. Inheriting properties from numerous classes may be accomplished using the syntax below:

Syntax:

```
class <derive-class_name> (<base-class 1>, <base-class 2>, ..... <base-class n>):
        # derived class body
```

Example:

```
1.  #creating base class
2.  class student():
3.   # Function inside base class
4.   def course(self,arg1):
5.       print ('-----Base class function-------')
6.       self.arg1=arg1
7.       print ('Course name:', self.arg1)
8.
9.  # Inherited class
10. class enroll(student):
11.      # Function inside derived class
12.     def is_enroll(self,arg2):
13.         print('--------Derived class function---------')
14.         self.arg2=arg2
15.         print("enrollment:", self.arg2)
16. enr = enroll()                          # An Object for derived class
17. print(enr.is_enroll("Lui"))
18. print(enr.course("python"))
```

Output:

```
--------Derived class function---------

enrollment: Lui

-----Base class function-------

Course name: pPython
```

TYPES OF INHERITANCE

There are four types of inheritance in Python, depending on the number of child and parent groups involved. It includes single, multiple, multi-level, and hierarchical inheritances.

Single Inheritance

A derived class can inherit properties from a single base class. This paves the way for code reutilization and the addition of new capabilities to pre-existing code. The principle of single inheritance is depicted here in Fig. (**1**) .

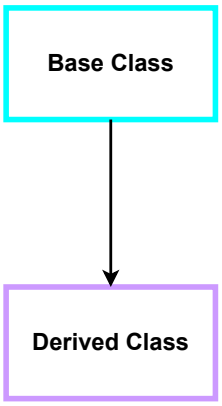


Fig. (1). Single Inheritance.

Example:

```
#creating base class
class student():
 # Function inside base class
 def grade(self,arg1):
     print ('-----Base class function-------')
     self.arg1=arg1
     print ('grade:', self.arg1)

# Inherited class
class enroll(student):
    # Function inside derived class
   def is_pass(self,arg2):
       print('--------Derived class function---------')
       self.arg2=arg2
       print("name:", self.arg2)

enr = enroll()                                  # An Object for derived class
print(enr.grade(8))
print(enr.is_pass("Lui"))
```

Output:

```
-----Base class function-------

grade: 8

--------Derived class function---------

name: Lui
```

Multi-level Inheritance

In multi-level inheritance, characteristics of the base class and the derived classes are transmitted to the newly created derived class. There is no upper bound on the number of levels that may be archived *via* multi-level inheritance in Python; this capability is unlimited. A visual representation of the multi-level inheritance notion may be found in Fig. (**2**).

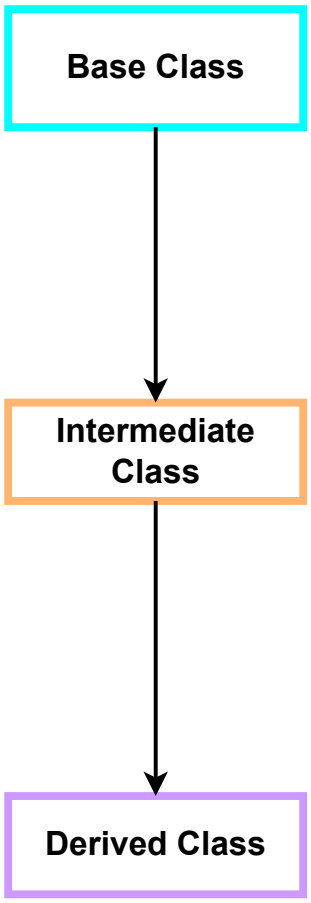


Fig. (2). Multi-level Inheritance.

Example:

```
# Base class
class Electronics:

 def __init__(self):
     print("----Base constructor----")

 def cat1(self, category):
     print("-----Base class-----")
     self.category = category
     print('Electronic  :', self.category)

# Intermediate class
class Televisons(Electronics):
 def style_1(self, style):
  self.style = style
  print("-----Intermediate class-----")
  print("Television :", self.style)

# Derived class
class model(Televisons):
 def model_name(self, name):
     print("-----Derived class-----")
     self.name = name
     print("model  :", self.name)

s1 = model()
s1.cat1('Portable')
s1.style_1('LCD')
s1.model_name('Samsung')
```

Output:

```
----Base constructor----

-----Base class-----

Electronic  : Portable

-----Intermediate class-----

Television : LCD

-----Derived class-----

model  : Samsung
```

Multiple Inheritance

A class can derive from numerous base classes in a multiple inheritance scenario. The derived class inherits all of the base classes' properties when there are multiple inheritances. Fig. (**3**) illustrates multiple inheritances, where a derived class inherits the traits of two base classes.

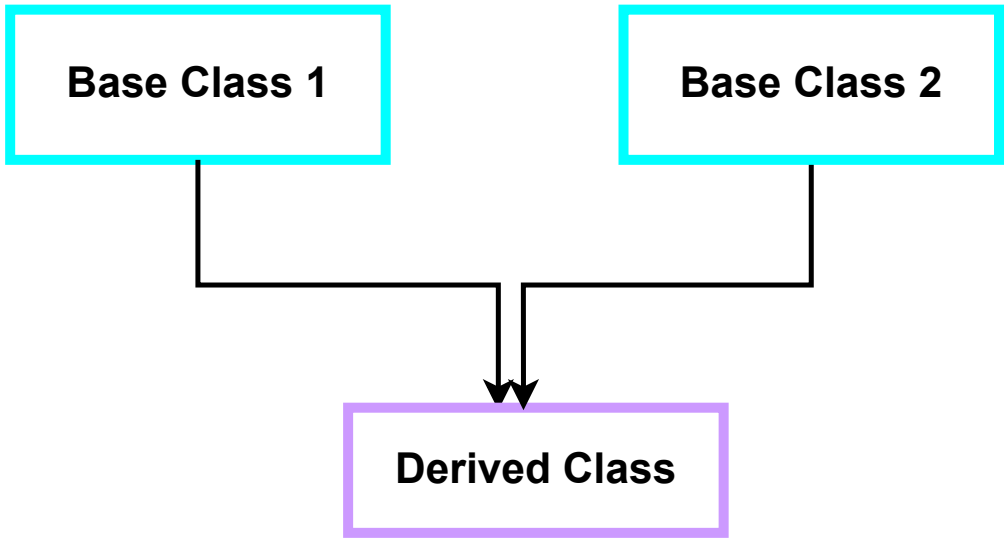


Fig. (3). Multiple Inheritance.

Example:

```
1.  # First Base class
2.  class Sneakers:
3.
4.   def cat1(self, category):
5.       print("-----Base class 1-----")
6.       self.category = category
7.       print('Sneakers  :', self.category)
8.
9.  # Second base class
10. class Sports_shoes:
11.  def style_1(self, style):
12.   self.style = style
13.   print("-----Base  class 2-----")
14.   print("Sports_shoes :", self.style)
15.
16. # Derived class
17. class shoes(Sneakers, Sports_shoes):
18.  def model_name(self, name):
19.      print("-----Derived class-----")
20.      self.name = name
21.      print("model  :", self.name)
22.
23.
24. s1 = shoes()
25. s1.cat1('Spark')
26. s1.style_1('Bata')
27. s1.model_name('casuals')
```

Output:

```
-----Base class 1-----

Sneakers  : Spark

-----Base  class 2-----

Sports_shoes : Bata

-----Derived class-----

model  : casuals
```

Hierarchical Inheritance

Multiple derived classes can be inherited from a single base class through a process known as hierarchy inheritance. In the example of hierarchical inheritance shown in Fig. **(4)** , three derived classes are descended from a single base class.

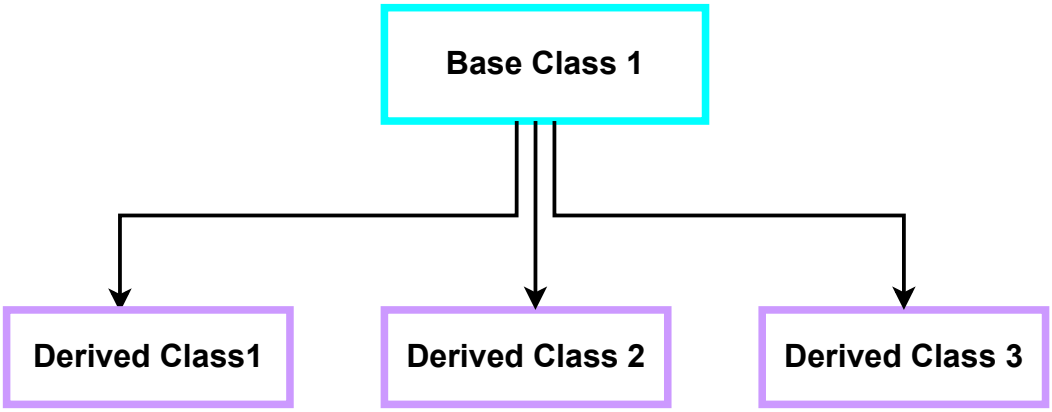


Fig. (4). Hierarchical Inheritance.

Example:

```
# Base class
class Televisons:
 def cat1(self, category):
    print("-----Base class-----")
    self.category = category
    print('Electronic  :', self.category)

# Derived   class 1
class plasma(Televisons):
     def style_1(self, style):
      self.style = style
      print("-----Derived class 1-----")
      print("Tube :", self.style)

    # Derived class 2
    class LCD(Televisons):
     def model_name(self, name):
        print("-----Derived class 2-----")
        self.name = name
        print("model  :", self.name)

    s1 = plasma()
    s1.cat1('42 inches')
    s1.style_1('LG')


    s2 = LCD()
    s2.cat1('32 inches')
    s2.model_name('Samsung')
```

Output:

```
-----Base class-----
Electronic  : 42 inches
-----Derived class 1-----
Tube : LG
-----Base class-----
Electronic  : 32 inches
-----Derived class 2-----
model  : Samsung
```

The *issubclass(sub,sup)* method()

Use the *issubclass(sub, sup)* method to determine how the defined classes relate. It returns true if the first class is a subclass of the second; otherwise, it returns false.

Example:

```
1.  # Base class
2.  class Electronics:
3.
4.   def cat1(self, category):
5.       print("-----Base class-----")
6.       self.category = category
7.       print('Electronic  :', self.category)
8.
9.  # Base  class 2
10. class Televisons():
11.  def style_1(self, style):
12.   self.style = style
13.   print("-----Intermediate class-----")
14.   print("Television :", self.style)
15.
16. # Derived class
17. class model(Electronics, Televisons):
18.  def model_name(self, name):
19.      print("-----Derived class-----")
20.      self.name = name
21.      print("model  :", self.name)
22.
23. s1 = model()
24. print(issubclass(model, Televisons))
25. print(issubclass(model, Electronics))
26. print(issubclass(Televisons, Electronics))
```

Output:

```
True

True

False
```

The *isinstance (obj, class)* method()

Use the *isinstance()* method to determine how objects and classes are related. It returns true if the first parameter, obj, is an instance of the second parameter, class.

Example:

```
1.  # Base class
2.  class Electronics:
3.
4.   def cat1(self, category):
5.       print("-----Base class-----")
6.       self.category = category
7.       print('Electronic  :', self.category)
8.
9.  # Base  class 2
10. class Televisons():
11.  def style_1(self, style):
12.   self.style = style
13.   print("-----Intermediate class-----")
14.   print("Television :", self.style)
15.
16. # Derived class
17. class model():
18.  def model_name(self, name):
19.      print("-----Derived class-----")
20.      self.name = name
21.      print("model  :", self.name)
22.
23. s1 = model()
24. print(isinstance(s1, Televisons))
25. print(isinstance(s1, Electronics))
26. print(isinstance(s1, model))
```

Output:

```
False

False

True
```

POLYMORPHISM IN PYTHON

The idea of anything being able to take on various forms is referred to as polymorphism in object-oriented programming. Polymorphism allows the utilization of a single interface for inputs of a wide variety of data types and classes and varying inputs. In Python, the concept of polymorphism can be defined in several different ways.

Polymorphism in Operators

We know that the plus (+) operator is frequently utilized in Python programming. Despite this, there is not a single field in which it may be utilized. When working with integer data types, the plus (+) operator must carry out arithmetic addition. Similarly, string data may be concatenated by using the + operator.

Example:

```
1. a = 85
2. b = 96
3. print (a+b)
4. X = "Joe"
5. Y = "went"
6. Z= "schol"
7. print (X+" "+Y+ " "+Z)
```

Output:

```
181

Joe went schol
```

Function Polymorphism

Several Python functions may be used for many different kinds of data. A good example of this type of function is the *len()* function. Python gives the ability to work with many different sorts of data. The *len()* function is compatible with

many data types, including strings, lists, tuples, sets, and dictionaries. However, it does yield precise information about particular data types.

Example:

```
print (len("Programming languages"))
print (len([78,5588,74988]))
print (len({"grade": 8.02, "status": "pass"}))
print (len('7851'))
```

Output:

```
21

3

2

4
```

Polymorphism in Class Methods

We can apply the polymorphism principle to the process of developing class methods since Python permits many classes to have methods with the same name. In this particular scenario, we might want to build a for loop that iterates over a tuple of items. Following that, we call the methods without considering the particular class type of each object. We assume that these methods are present in all classes.

Example:

```
# Polymorphism in classes
class A():
    def fun1(self):
        print ("python")

    def fun2(self):
        print ("programming language")

class B():
    def fun1(self):
        print ("Anaconda")

    def fun2(self):
        print ("IDE")

obj1 = A()
obj2 = B()

for i in (obj1, obj2):
    i.fun1()
    i.fun2()
```

Output:

```
python

programming language

Anaconda

IDE
```

Method Overriding

Python's polymorphism feature enables programmers to create child classes with methods with the same names as those in their parent classes. In the inheritance process, the child class receives the methods defined for the parent class. On the other hand, a child class may change an inherited method from a parent class that the child class has inherited. This is especially helpful in situations where the inherited method from the parent class does not exactly work for the child class. In the circumstances like these, the method is re-created from scratch within the child class. The process of re-implementing a method in a parent class within a child class is referred to as method overriding or polymorphism in inheritance.

Example:

```
1.  # Example of method overriding
2.  class Subjects:
3.      def func1(self):
4.          print ('It is Base Class')
5.      def func2(self):
6.          print (' Subjects Class')
7.
8.  class Core(Subjects):
9.
10.     def func2(self):
11.         print ('Core subject class')
12.
13. class Elec(Subjects):
14.
15.     def func2(self):
16.         print ('Elec subject class')
17.
18. a1 = Subjects()
19. a1.func1()
20. a1.func2()
21.
22. print('------------')
23. b1 = Core()
24. b1.func1()
25. b1.func2()
26. print ('------------')
27. e1 = Elec()
28. e1.func1()
29. e1.func2()
```

Output:

```
It is Base Class
Subjects Class
----------
It is Base Class
Core subject class
----------
It is Base Class
Elec subject class
```

Built-in Class Functions

The following are the class's built-in functions, as shown in Table **1**.

Example:

```
1.  class student:
2.      def __init__(self, name, ids, sub):
3.          self.name = name
4.          self.ids = ids
5.          self.sub = sub
6.
7.  #  object of the class student
8.  s1= student("Joe",408, "Java" )
9.
10. # printing attribute name of the object s1
11. print (getattr(s1, 'name'))
12.
13. # Changing the value of attribute sub
14. setattr(s1, "sub", "Python")
15.
16. # Display modified value of sub
17. print (getattr(s1, 'sub'))
18.
19. # prints true if the student contains the attribute with  id
20. print (hasattr(s1, 'ids'))
21.
22. print("------Before deletion------")
23. print (s1.ids)
24. # deleting one attribute
25. delattr(s1, 'ids')
26.  # It will show an error since the attribute id has been deleted
27. print("------After deletion------")
28. print (s1.ids)
```

Table 1. Built-in class functions.

getattr(obj,name,default)	It is used to get the object's attribute.
setattr(obj, name,value)	It is used to give an object's attribute a specific value.
delattr(obj, name)	It deletes a specific attribute.
hasattr(obj, name)	If the object has a particular attribute, it returns true.

Output:

```
Joe
Python
True
------Before deletion------
408
------After deletion------
Traceback (most recent call last):
  File "<string>", line 28, in <module>
AttributeError: 'student' object has no attribute 'ids'
```

Built-in Class Attributes

In addition to other attributes, a Python class has several built-in class attributes that provide information about the class. The built-in class attributes are shown in Table **2**.

Table 2. Built-in class attributes.

__dict__	It provides a dictionary with class namespace details.
__doc__	It has a string with the documentation for the class.
__name__	It is used to get the name of the class.
__module__	It allows to get to the module where this class is defined.
__bases__	It has a tuple that includes all base classes.

Example:

```
1.  class student:
2.      def __init__(self, name, ids, sub):
3.          self.name = name
4.          self.ids = ids
5.          self.sub = sub
6.
7.  def display(self):
8.          print (self.name)
9.          print (self.sub)
10. a1 = student("Akarck",4874, "Python")
11. print (a1.__doc__)
12. print (a1.__dict__)
13. print (a1.__module__)
14. print (display.__name__)
15. print (student.__bases__)
```

Output:

```
None

{'name': 'Akarck', 'ids': 4874, 'sub': 'Python'}

__main__

display

(<class 'object'>,)
```

Static Variables

All of the objects share static variables. It is also sometimes referred to as a class variable. Variables that are not static can change depending on the object they are associated with. Instance variables or non-static variables are variables that have values allocated to them within methods. In contrast, class variables are variables that have values assigned to them at the time of the class declaration.

Example:

```
1.  # Class declaration
2.  class Televison:
3.      cost = 150000              # Class or static variable
4.      def __init__(self,company,style):
5.          self.company = company      # non-static Variable
6.          self.style = style
7.
8.  # Objects creation
9.  a1 = Televison("LG",'LCD')
10. a2=  Televison("Samsung",'Plasma')
11.
12. print (a1.cost)  # prints 150000
13. print (a2.cost)  # prints 150000
14. print (a1.company)    # prints LG
15. print (a2.company)    # prints  Samsung
16. print (a1.style)   # prints LCD
17. print (a2.style)   # prints Plasma
18.
19. # Class variables can be accessed using class name also
20. print (Televison.cost)    # prints 150000
21.
22. # we change the fees for just s1 it won't be changed for s2
23. a1.cost = 125000
24. print (a1.cost)     # prints 125000
25. print (a2.cost)     # prints 150000
26.
27. # To change the fees for all we can change them directly from the class
28. Televison.cost = 140000
29.
30. print (a2.cost)     # prints 140000
```

Output:

```
150000
150000
LG
Samsung
LCD
Plasma
150000
125000
150000
140000
```

CONCLUDING REMARKS

In this chapter, we discussed object-oriented programming concepts and their implementation in Python. We discussed constructors, inheritance, polymorphism, *etc.* Various built-in class functions and attributes are described with the help of examples. Based on these examples, you can write your code for object-oriented programming. In the next chapter, we will discuss machine learning concepts and their implementations in Python.

CHAPTER 11

Python for Machine Learning

Abstract: Anticipating that the user has a good knowledge of the core elements of Python we now explore the applicative aspect of Python. In this chapter, we will look at Python, especially from the Machine Learning (ML) point of view. We will discuss the various libraries and their utility in ML and then lay hands over the programming demonstrations.

Keywords: Libraries, Machine learning, Packages, Prediction and classification.

INTRODUCTION

We assume the readers now have enough preliminary knowledge to dive deep into programming with Python. Python has been the developer's choice, and Machine Learning is one of the major application areas of Python. Machine Learning is the field of computer science that allows computer programs to attain the capability much like a human brain, *i.e*., learn from past experiences and perform future tasks. In this chapter, you will learn to program Machine Learning algorithms with Python.

Anomalous to traditional programming, Machine Learning requires no pre-defined rules but the design of a mathematical model for decision-making rather than human interference. Figs. (**1** and **2**) show the disparity between the two paradigms of programming. In traditional programming, rules, and data are fed to the computer, and the results are evaluated. In case of an error, the problem is studied and analyzed, and changes are made to the rules. But in the Machine Learning paradigm, the learning is leveraged on the training data, and results are evaluated. The defined model is then tested over test data, and output is produced.

Fig. (1). Traditional Programming.

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

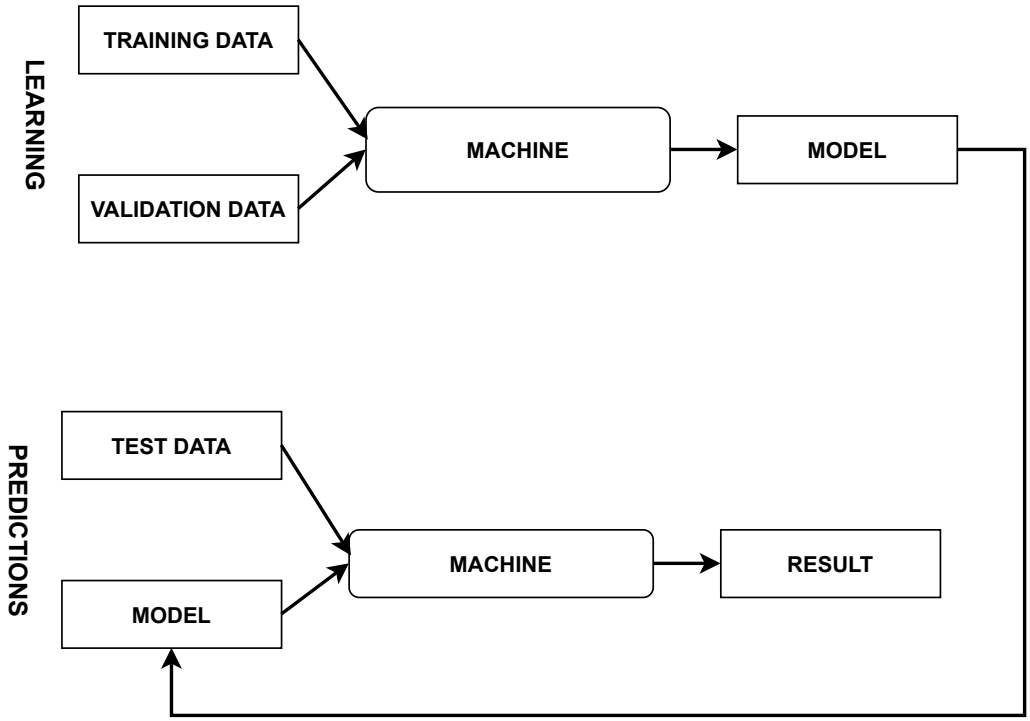


Fig. (2). Machine Learning Programming Paradigm.

Self-learning is the crux of Machine Learning. The performance based on data extraction, preprocessing, and analysis without being explicitly programmed defines the objective of Machine Learning. It aims to facilitate the machine to work directly without being programmed. Decision-making is an important task that relies on pattern extraction and information modeling based on trial and error and probabilistic reasoning. Thus, one can say that decisions are not based on the pre-set rules but on the input data. To minimize the errors, the programmers can tweak model settings called hyperparameter tuning. For learning, the programmer splits the data into training and testing data. The training set helps to learn the patterns in the data and validate the results. Finally, the developed model can be evaluated over test data. The model can be deployed for other applications if the performance is satisfactory.

Machine Learning has been divided into two categories based on the training procedure. If the machine is trained with the labeled training data, *i.e.*, the data that has been classified under different classes, then that type is called supervised learning. And, if the learning over training data is without human interference, *i.e.*, no defined classes, then that is called unsupervised learning. The difference

between supervised and unsupervised learning is given in Table **1**.

Table 1. Difference between Supervised and Unsupervised Learning.

Supervised Learning	Unsupervised Learning
Input data is labeled	Input data is unlabeled
Feedback mechanism present	A feedback mechanism is absent
Data is classified	Properties are assigned to data
Suitable for prediction tasks	Suitable for analysis tasks
A known number of classes	An unknown number of classes
Consists of explanatory and response variables	Consists only of explanatory variables

Supervised learning is applicable in two major domains, namely, classification and regression. Classification is the process of categorizing data into different categories based on the labeled data used for training. Regression is similar to classification except that it can also be applied to continuous data, unlike classification, which can only be applied to discrete values. Unsupervised learning is suitable for clustering and association tasks. Clustering is used to discover the groups in the data, while the association is used for extracting the rules from a large amount of data.

IMPORTANT PYTHON LIBRARIES

Now we move to some important Python libraries that are used in Machine Learning.

• NUMPY

It's an array-processing package suitable for processing large multi-dimensional arrays and matrices.

Example:

```
1. import numpy as np
2. arr = np.array([[4, 13, 17, 6], [2,4,3,8]])
3. print(arr)
```

Output:

```
[[ 4 13 17  6]
 [ 2  4  3  8]]
```

• PANDAS

It is one of the major data preprocessing libraries that allows easy manipulation and analysis. It plays a major role in preparing the training data. It provides tools for loading, preparing, manipulating, and analyzing data.

Example:

```
1. #Importing pandas as pd
2. import pandas as pd
3. data = {"course": ["Python", "Machine Learning", "Deep Learning", "Programming Logic"],
4.         "code": ["C007","C002", "C005", "C006"]
5.         }
6. data_table = pd.DataFrame(data)
7. print (data_table)
```

Output:

```
    course     code
0     Python     C007
1     Machine Learning     C002
2     Deep Learning     C005
3     Programming Logic     C006
```

Pandas provide two data structure, namely, series, and data frames. A series is defined as a one-dimensional array that stores various data types. They comprise multiple columns but have one parameter only. The row label of the series is called the index.

Example:

```
1. import pandas as pd
2. import numpy as np
3. info = np.array(['P','a','n','d','a','s'])
4. ans = pd.Series(info)
5. print(ans)
```

Output:

```
0  P
1  a
2  n
3  d
4  a
5  s
dtype: object
```

DataFrame works with two-dimensional arrays. They are the standardized method for data storage. They have two different indices, *i.e.*, row index, and column index. Here the columns can contain heterogeneous data types like bool, int, *etc*.

Example:

```
1. import pandas as pd
2. w1= ['Programming', 'Language']
3.
4. # Calling DataFrame
5. df = pd.DataFrame(w1)
6. print(df)
```

Output:

```
          0
0 Programming
1    Language
```

• SCIKIT-LEARN

It allows the use of various machine learning algorithms like classification, regression, clustering, reduction, dimensionality, preprocessing, *etc*. It can be used in collaboration with other libraries like NumPy, and Pandas. It can be used for both data analysis and data mining. It is one of the frequently used libraries that help build advanced-level ML models.

Example:

```
# load the iris dataset as an example
from sklearn.datasets import load_iris
iris = load_iris()

# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target

# store the feature and target names
feature_names = iris.feature_names
target_names = iris.target_names

# printing features and target names of our dataset
print("Feature names:", feature_names)
print("Target names:", target_names)

# X and y are numpy arrays
print("\nType of X is:", type(X))

# printing first 5 input rows
print("\nFirst 10 rows of X:\n", X[:10])
```

Output:

```
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target names: ['setosa' 'versicolor' 'virginica']


Type of X is: <class 'numpy.ndarray'>


First 5 rows of X:
 [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]]
```

• MATPLOTLIB

It is a library that helps make a 2D plot in Python, which is vital for visualizing data. Using this, you can easily generate a histogram, bar charts, scatter plots, and error charts. Some basic functions of matplotlib are illustrated in Table **2**.

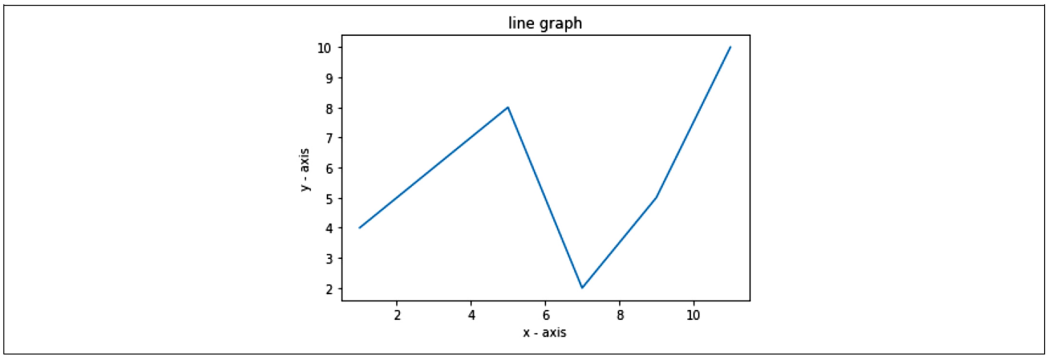
Table 2. Different elements of matplotlib.

Method	Description
Plot()	Creates a plot but doesn't show it.
Show()	Used to display the plot
xlabel()	Defines label for the x-axis
ylabel()	Defines label for the y-axis
title()	Defines the title for the graph
legend()	Provide meaning to graph elements

Example:

```
1.  # load the iris dataset as an example
2.  # importing the required module
3.  import matplotlib.pyplot as plt
4.
5.  # x axis values
6.  x = [1,5,7,9,11]
7.  # corresponding y-axis values
8.  y = [4,8,2,5,10]
9.
10. # plotting the points
11. plt.plot(x, y)
12.
13. # naming the x-axis
14. plt.xlabel('x - axis')
15. # naming the y axis
16. plt.ylabel('y - axis')
17.
18. # giving a title to my graph
19. plt.title('My first graph!')
20.
21. # function to show the plot
22. plt.show()
```

Output:



• TENSORFLOW

Developed by the Google brain team in 2015, it is registered under an Apache license. It can be used with different toolkits to provide multiple functionalities. Its prime focus is to simplify the implementation of deep learning algorithms.

Example:

```
1. from future import print function
2. import tensorflow as tf
3. # Define tensor constants.
4. a = tf.constant(4)
5. b = tf.constant(6)
6. c = tf.constant(8)
7.
8. # Various tensor operations.
9. # Note: Tensors also support python operators (+, *, ...)
10. add = tf.add(a, b)
11. sub = tf.subtract(a, b)
12. mul = tf.multiply(a, b)
13. div = tf.divide(a, b)
14.
15. # Access tensors value.
16. print("add =", add.numpy())
17. print("sub =", sub.numpy())
18. print("mul =", mul.numpy())
19. print("div =", div.numpy())
```

Output:

```
add = 10
sub = -2
mul = 24
div = 0.6666666666666666
```

• KERAS

It is primarily focused on the faster implementation of deep learning. It has been widely used because it is simple and flexible, with a powerful API capable of implementation over CNTK, TensorFlow, or Theano.

Example:

```
1. import numpy as np
2. import pandas as pd
3. # Import scikit-learn modules
4. from sklearn import preprocessing
5. from sklearn.preprocessing import StandardScaler
6. from sklearn.model_selection import train_test_split
7. # Import keras modules
8. import keras
9. from keras.models import Sequential
10. from keras.layers import Dense
```

• PYTORCH

PyTorch is an open-source framework developed by Team Facebook that focuses on deep learning. It has viable support for Python and C++. There are three levels of abstraction present in PyTorch: **Tensor, variable, and Module**. The salient features of PyTorch are:

1. **Interface**: The simple and easy execution of PyTorch makes it quite popular.
2. **Python Usage**: It uses a Python-based library and hence can leverage the functionalities provided by the Python environment.
3. **Computational graphs**: It offers dynamic computational graphs that can be changed during runtime.

NOTE: Use **pip install torch** to install PyTorch if it is not installed already.

Example:

```
import torch
import math
dtype = torch.float
device = torch.device("cpu")
# Create random input and output data
x = torch.linspace(-math.pi, math.pi, 50, device=device, dtype=dtype)
y = torch.sin(x)
print (x)
print (y)
p=torch.linspace(4, 50, steps=5)
print (p)
q=torch.linspace(-5, 20, steps=10)
print (q)
```

Output:

```
tensor([-3.1416, -3.0134, -2.8851, -2.7569, -2.6287, -2.5005, -2.3722, -2.2440, -2.1158, -1.9875,
-1.8593, -1.7311, -1.6029, -1.4746, -1.3464, -1.2182, -1.0899, -0.9617, -0.8335, -0.7053, -0.5770,
-0.4488, -0.3206, -0.1923, -0.0641,  0.0641,  0.1923,  0.3206,  0.4488,  0.5770,  0.7053,
0.8335,0.9617,  1.0899,  1.2182,  1.3464,  1.4746,  1.6029,  1.7311,  1.8593, 1.9875,  2.1158,
2.2440,  2.3722,  2.5005,  2.6287,  2.7569,  2.8851, 3.0134,  3.1416])


tensor([ 8.7423e-08, -1.2788e-01, -2.5365e-01, -3.7527e-01, -4.9072e-01, -5.9811e-01, -6.9568e-
01, -7.8183e-01, -8.5514e-01, -9.1441e-01, -9.5867e-01, -9.8718e-01, -9.9949e-01, -9.9538e-01,
-9.7493e-01, -9.3847e-01, -8.8660e-01, -8.2017e-01, -7.4028e-01, -6.4823e-01,

-5.4553e-01,  -4.3388e-01,  -3.1511e-01,  -1.9116e-01,  -6.4070e-02,  6.4070e-02,   1.9116e-01,
3.1511e-01,   4.3388e-01,   5.4553e-01,   6.4823e-01,   7.4028e-01,   8.2017e-01,   8.8660e-01,
9.3847e-01,  9.7493e-01,   9.9538e-01,   9.9949e-01,   9.8718e-01,   9.5867e-01,   9.1441e-01,
8.5514e-01,   7.8183e-01,   6.9568e-01,   5.9811e-01,  4.9072e-01,   3.7527e-01,   2.5365e-01,
1.2788e-01, -8.7423e-08])


tensor([ 4.0000, 15.5000, 27.0000, 38.5000, 50.0000])

tensor([-5.0000, -2.2222,  0.5556,  3.3333,  6.1111,  8.8889, 11.6667, 14.4444,
        17.2222, 20.0000])
```

• NLTK

NLTK is an acronym for Natural Language Toolkit that is primarily used for NLP (Natural Language Processing). It provides easy and effective interface support

for corpora and lexical resources besides text processing libraries for classification, tokenization, stemming, tagging, and parsing. Some common applications of NLTK include keyword search, voice recognition, stemming of words, and lemmatizing and handwriting. For proper working with added functionalities, installing at least version 3.5 for Python is required. You can download it from **http://nltk.org/.** NLTK works efficiently with Python versions higher than 3.6.

Installing NLTK on Mac/ Unix

1. Install NLTK: run pip install –user -U nltk.
2. Test installation: run import nltk.

Installing NLTK on Windows

1. Install the latest version of Python, preferably a higher version than 3.6
2. Install NLTK using pip install nltk
3. Test installation: Start> Python > type import nltk.

If the installation is successful, you must also install other dependencies.

4. Install dictionaries for language processing: Use nltk.download() command.

When you run this command, an installation window will prompt up. Select the 'Download' option to install dictionaries, other languages, and grammar data frames required for full NLTK functionality. NLTK supports English extensively and also provides limited support for Spanish and French. Fig. (**3**) depicts the NLTK downloader window. You can check the status of items in your collections and the download's progress. It facilitates the inventory of supported corpora and other packages, as depicted in Figs. (**4**–**5**).

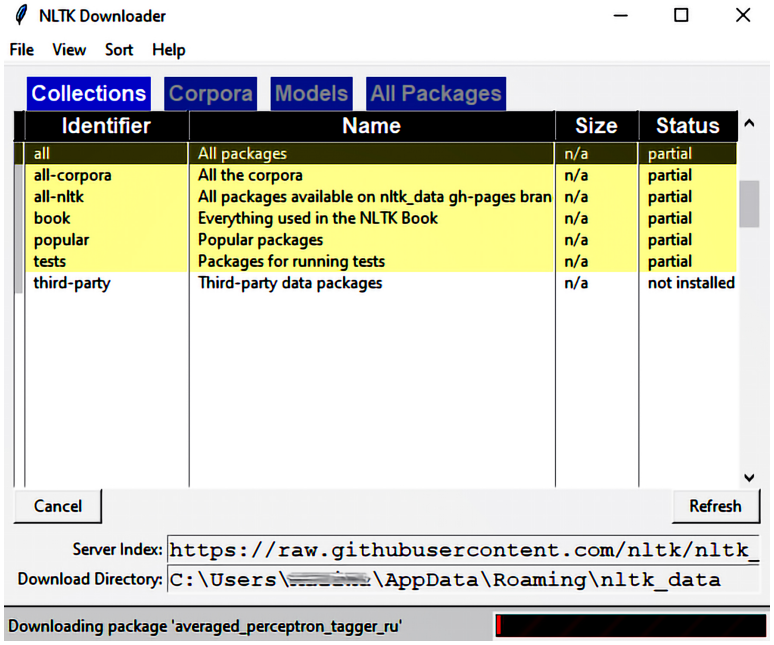


Fig. (3). Downloading the NLTK Book Collection using nltk.download().

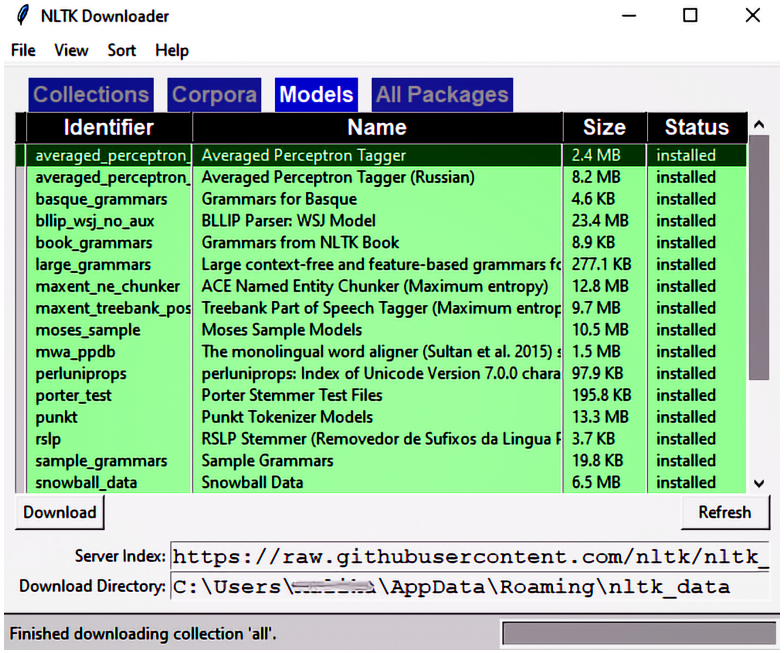


Fig. (4). NLTK models.

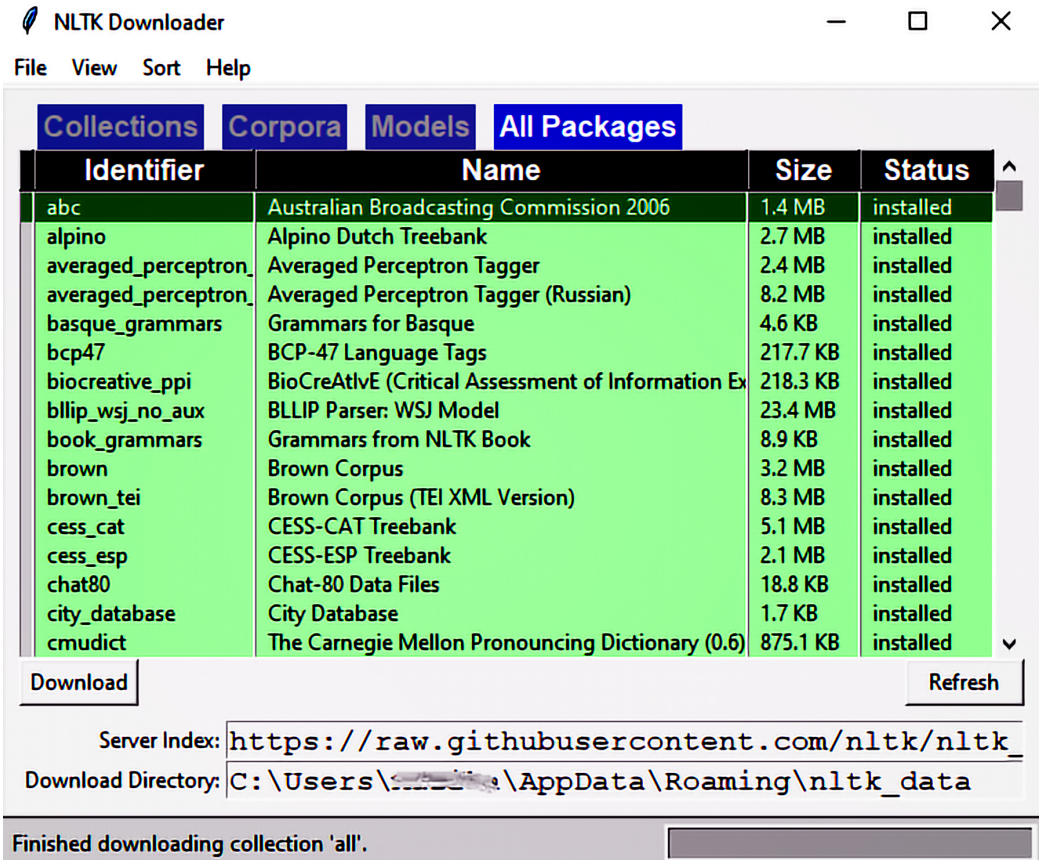


Fig. (5). NLTK Package Inventory.

Example:

```
1. import nltk
2. sentence = """Cris went out on the new bicycle ."""
3. tokens = nltk.word  tokenize(sentence)
4. print (tokens  )
```

Output:

```
['Cris', ' went ',  ' out ', ' on ', ' the ', ' new ', ' bicycle','.']
```

Example: Loading Text

```
1.  from nltk.book import *
```

Output:

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

Example: Searching Text

```
1. from nltk.book import *
2. text1.concordance("ball")
```

Output:

```
Displaying 12 of 12 matches:
 " These things are reciprocal ; the ball rebounds , only to bound forward aga
This is my substitute for pistol and ball . With a philosophical flourish Cato
t know Ahab then . " Am I a cannon - ball , Stubb ," said Ahab , " that thou w
 well ; belike the whole world ' s a ball , as you scholars have it ; and so '
ically wild and unearthly , that the ball of free will dropped from my hand ,
n again ; but the next night an iron ball , closely netted , partly rolled fro
 it to be true ; it happened on this ball ; I trod the ship ; I knew the crew
fle holds the fatal powder , and the ball , and the explosion ; so the gracefu
 like malefactors with the chain and ball . But upon flinging the third , in t
broke through the ribs -- and with a ball of Arsacidean twine , wandered , edd
ks something like a white billiard - ball . I was told that there were still s
furrow in the sea as when a cannon - ball , missent , becomes a plough - share
```

CONCLUDING REMARKS

In this chapter, we have given a gist of the various Python libraries. These libraries are indispensable for machine learning and data science applications. The range of operations facilitated by the Python libraries simplifies and speeds up the process of task automation and prediction outcome. The next chapter comprises some useful Python programs that use these libraries, demonstrating their importance.

CHAPTER 12

Programming with Python

Abstract: After the successful comprehension of the different aspects of Python and its applications in ML, it's now the time to look at how they can be combined and put to work using common examples. This will surely increase the reader's comprehension of the intricacies of Python and demonstrate the efficiency of the language in making the programming simple.

Keywords: Binary search, Factorial, Time series.

INTRODUCTION

After going through the previous chapters, we are sure you have a better understanding of the basics of Python. Now let's gear up to some programming exercises with Python. In the beginning, we provide some simple and basic Python programs using functions, lists, dictionaries, arrays, *etc*. Then we discuss some basic machine learning applications with Python programming.

BASIC PYTHON PROGRAM

Program to Solve a Quadratic Equation

Example:

```
1. # Import complex math module
2. import cmath
3. a = input('Enter your choice: ')
4. b =input('Enter your choice: ')
5. c = input('Enter your choice: ')

6. # Calculate the discriminant
7. d = (b**2) - (4*a*c)

8. # Find two solutions
9. sol1 = (-b-cmath.sqrt(d))/(2*a)
10. sol2 = (-b+cmath.sqrt(d))/(2*a)
11. print('The solution are {0} and {1}'.format(sol1,sol2))
```

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

Output:

```
Enter a: 1
Enter b: 2
Enter c: -15
The solution are (-5+0j) and (3+0j)
```

Program to Swap Two Numbers

Example:

```
1. P = int( input("Please enter value for P: "))
2. Q = int( input("Please enter value for Q: "))
3. # To swap the value of two variables
4. # We will user third variable which is a temporary variable
5. temp_1 = P
6. P = Q
7. Q = temp_1
8. print ("The Value of P after swapping: ", P)
9. print ("The Value of Q after swapping: ", Q)
```

Output:

```
Please enter value for P: 21
Please enter value for Q: 81
The Value of P after swapping:  81
The Value of Q after swapping:  21
```

Program to Find the Factorial of Two Numbers

Example:

```
1.  num = int(input("Enter a number: "))
2.  factorial = 1
3.  if num < 0:
4.     print(" Factorial does not exist for negative numbers")
5.  elif num == 0:
6.     print("The factorial of 0 is 1")
7.  else:
8.     for i in range(1,num + 1):
9.           factorial = factorial*i
10.          print("The factorial of",num,"is",factorial)
```

Output:

```
Enter a number: 9
The factorial of 9 is 362880
```

Program for Fibonacci Series Using Recursion

Example:

```
1.  def recur_fibo(n):
2.      if n <= 1:
3.          return n
4.      else:
5.          return(recur_fibo(n-1) + recur_fibo(n-2))
6.
7.  nterms = int(input("How many terms? "))
8.  # Check if the number of terms is valid
9.  if nterms <= 0:
10.     print("Plese enter a positive integer")
11. else:
12.    print("Fibonacci sequence:")
13.    for i in range(nterms):
14.        print(recur_fibo(i))
```

Output:

```
How many terms? 5
Fibonacci sequence:
0
1
1
2
3
```

Sorting Elements of Array in Ascending Order

Example:

```
arr = [2,8,6,41,35,7];
temp = 0;
print("Elements of original array: ");
for i in range(0, len(arr)):
    print(arr[i], end=" ");
#Sort the array in ascending order
for i in range(0, len(arr)):
    for j in range(i+1, len(arr)):
        if(arr[i] > arr[j]):
    temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
print();
print("Elements of array sorted in ascending order: ");
for i in range(0, len(arr)):
print(arr[i], end=" ");
```

Output:

```
Elements of original array:
2 8 6 41 35 7
Elements of array sorted in ascending order: 2,6,7,8,35,41
```

Program to Print the Sum of Elements of Array

Example:

```
arr = [4,6,8,10,15];
sum = 0;
for i in range(0, len(arr)):
sum = sum + arr[i];
print("Sum of all the elements of an array: " + str(sum));
```

Output:

```
Sum of all the elements of an array: 43
```

Program to Transpose a Matrix

Example:

```
m = [[11,42],[39,75],[63,12]]
for row in m :
    print(row)
rez = [[m[j][i] for j in range(len(m))] for i in range(len(m[0]))]
print("\n")
print("Transposed Matrix is:")
for row in rez:
    print(row)
```

Output:

```
[11, 42]
[39, 75]
[63, 12]
Transposed Matrix is:
[11, 39, 63]
[42, 75, 12]
```

Program to Reverse a String

Example:

```
def reverse_string(str):
str1 = ""   # Declaring empty string
for i in str:
str1 = i + str1
return str1
str = "Python Programming"
print("The original string is: ",str)
print("The reverse string is:",reverse_string(str))
```

Output:

```
The original string is:   Python Programming
The reverse string is: gnimmargorP nohtyP
```

Program to Concatenate Two Strings

Example:

```
# Defining strings
str1 = "Book on "
str2 = "Python"
str3 = str1 + str2
print("The concatenated string:", str3)
```

Output:

```
The concatenated string: Book on Python
```

Program to Append Elements to the List

Example:

```
fruits = ["Mango", "Apple", "Banana", "Tomato"]
Fruits2 = input("Please enter a name:\n")
fruits.append(Fruits2)
print('Updated List is:', fruits)
```

Output:

```
Please enter a name:

berry

Updated List is: ['Mango', 'Apple', 'Banana', 'Tomato', 'berry']
```

Program to Delete an Element of List

Example:

```
list = ['Python', 'Book', 'Computer','Science', 'Program']
print(list)
list.remove('Python')
print( "After deletion:", list )
```

Output:

```
['Python', 'Book', 'Computer', 'Science', 'Program']

After deletion: ['Book', 'Computer', 'Science', 'Program']
```

Program for Linear Search

Example:

```
def linearsearch(list, m):
    for i in range(len(list)):
        if list[i] == m:
            return i
    return -1

list1= ['p','y','t','h','o','n']
n = 'a'
x= 'o'
print("element is at index "+str(linearsearch(list1,n)))
print("Index of the element: "+str(linearsearch(list1,x)))
```

Output:

```
element is at index -1

Index of the element: 4
```

Program for Binary Search

Example:

```
def binary_search(list1, m):
 low = 0
 high = len(list1) - 1
 mid = 0
 while low <= high:

  mid = (high + low) // 2
  if list1[mid] < m:
   low = mid + 1
  elif list1[mid] > m:
   high = mid - 1
  else:
   return mid
 return 0
list = [1, 42,12,78,85]
n = 78
#calling function
result = binary_search(list, n)
if result != -1:
 print("Index of element", str(result))
else:
 print("Not present in array")
```

Output:

```
Element is present at index 3
```

In machine learning, Python is heavily used to perform different analyses. Here we introduce the time series analysis based on Python. As discussed in previous chapters there are various Python libraries that are extensively useful for machine learning tasks.

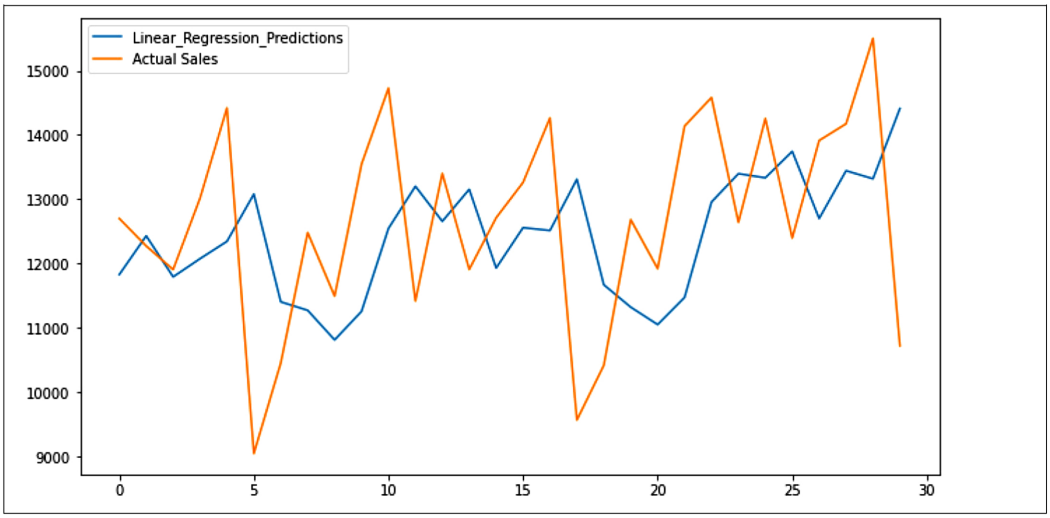
Program to Perform Time Series Analysis

Example:

```
import pandas
import numpy
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from math import sqrt

df= pandas .DataFrame()
df = pandas .read_csv('/content/Alcohol_Sales.csv',index_col='DATE',parse_dates=True)
df.index.freq = 'MS'
df.tail()
df.columns = ['Sales']
df.plot(figsize=(12,8))
df['Sale_LastMonth']=df['Sales'].shift(+1)
df['Sale_2Monthsback']=df['Sales'].shift(+2)
df['Sale_3Monthsback']=df=df.dropna()
df
df['Sales'].shift(+3)
lin_model=LinearRegression()
x1,x2,x3,y=df['Sale_LastMonth'],df['Sale_2Monthsback'],df['Sale_3Monthsback'],df['Sales']
x1,x2,x3,y= numpy .array(x1),n numpy  p.array(x2), numpy .array(x3), numpy .array(y)
x1,x2,x3,y=x1.reshape(-1,1),x2.reshape(-1,1),x3.reshape(-1,1),y.reshape(-1,1)
final_x= numpy .concatenate((x1,x2,x3),axis=1)
print(final_x)
X_train,X_test,y_train,y_test=final_x[:-30],final_x[-30:],y[:-30],y[-30:]
model.fit(X_train,y_train)
lin_model.fit(X_train,y_train)
lin_pred=lin_model.predict(X_test)
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11,6)
plt.plot(lin_pred,label='Linear_Regression_Predictions')
plt.plot(y_test,label='Actual Sales')
plt.legend(loc="upper left")
plt.show()
rmse_rf=sqrt(mean_squared_error(pred,y_test))
rmse_lr=sqrt(mean_squared_error(lin_pred,y_test))
print('Mean Squared Error for Random Forest Model is:',rmse_rf)
print('Mean Squared Error for Linear Regression Model is:',rmse_lr)
```

Output:



CONCLUDING REMARKS

In this chapter, we have demonstrated some of the important Python programs. You can also write new programs with the help of these programs. Before writing any program, it is essential to understand the nature of the problem, and then the solution can be defined accordingly.

BIBLIOGRAPHY

[1] Available at: www.python.org

[2] Available at: https://docs.anaconda.com

[3] M. Lutz, *Learning Python: Powerful Object-Oriented Programming.* 5th O'Reilly Media, Incorporated, 2009.

[4] D.M. Beazley, *Python Essential Reference.* Addison-Wesley, 2009.

[5] C.P. Milliken, *Python Projects for Beginners: A ten-week bootcamp approach to python programming.* (1st ed.) Apress, 2019.

[6] A. Harris, *Python for Beginners: Learn Computer Programming with Python Now and How to Use It with This Step by Step Guide That Gives You the Basics of Python Coding + Practical Exercises*. Independently Published, 2019.

[7] F. Romano, and H. Kruger, *Learn Python Programming: An in-depth introduction to the fundamentals of Python 3rd Edition.* (3rd ed.). Packt Publishing, 2021.

[8] A. Müller, and S. Guido, *Introduction to Machine Learning with Python: A Guide for Data Scientists*. (1st ed.). O'Reilly Media, 2016.

[9] Availabel at: https://www.nltk.org/ For a complete reference to download and usage to NLTK.

[10] D. Paper, *Hands-on scikit-learn for machine learning applications. Data Science Fundamentals with Python.* Apress: Berkeley, CA, 2020.[http://dx.doi.org/10.1007/978-1-4842-5373-1]

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

SUBJECT INDEX

A

Anaconda 7, 8
navigator 8
open-source distribution 7
Applications 1, 7, 17, 43, 142, 182, 184, 210, 238, 250, 251
banking 43
basic machine learning 251
data science 250
desktop 1
Arguments, positional 136
Arrays 20, 115, 116, 138, 239, 240, 241, 251, 254
large multi-dimensional 239
one-dimensional 240
two-dimensional 241
Associativity, operator's 34
Attributes 121, 198, 209, 210, 215, 216, 219, 233, 236
class's 216
module's 198
particular 233
Automatic memory management allocation 4

B

Benevolent dictator for life (BDFL) 1, 3
Binary 30, 31, 165, 177, 178
files 165, 177, 178
left shift 30
right shift 31
Bitwise operator 25, 30, 31
Boolean 18, 22, 27, 77, 91
data type 22
value 27, 91
Break statement 12, 56, 57

C

Characters 36, 37, 38, 39, 77, 78, 79, 88, 90, 91, 93, 100, 101, 122, 164, 169, 170
alphabetical 39
alphanumeric 38, 101
non-alphanumeric 101
Class(s) 99, 216, 233, 234
attributes 216, 233
declaration 234
string module 99
Code 2, 3, 12, 13, 15, 184, 185, 198, 208, 219, 220
block 12, 13, 184, 185, 208
malicious 184
platform-independent 3
portable 2
program's 219
readability 12
redundancy 15
reutilization 220
source 2, 3, 198
Collection 4, 20, 21, 23, 77, 115, 121, 138, 153, 163, 192, 195, 209, 247
garbage 4, 192
ordered 77, 121
stable 121
Commands 9, 10, 11, 28, 169, 182, 184, 247
bash 9
raise 184
Computer 2, 7, 17, 115, 138, 139, 163, 177, 195, 208, 209, 237
hardware 2
languages 115, 139
programming 17
program's components 208
Concatenating components 81
Condition 31, 32, 38, 44, 46, 53, 73, 182, 192
false 32
termination 73
Conditional processing 43
Constructor(s) 216, 217, 218, 236

Krishna Kumar Mohbey & Malika Acharya
All rights reserved-© 2023 Bentham Science Publishers

default 216, 218
function executes 216
non-parameterized 217
Control 43, 56, 60
flow alterations 43
structures, sequential 60
Control statements 43, 44, 60
decision 43, 44
Conversion 23, 24, 30, 134, 161, 201
angle 201
binary 30

D

Data 122, 209, 215, 238, 240, 241
abstraction 215
analysis 241
encapsulation 209
extraction 238
formats 122
frames 240
mining 241
storage 241
Data elements 122, 208
heterogeneous 122
Data types 121, 201, 241
container 201
heterogeneous 241
immutable 121
Debugging assertions 11
Deep learning algorithms 244
Dictionaries, employee's 158
Dictionary 121, 140, 147, 152, 155, 158, 159, 161
comparison 159
components 147
creation 140
elements 158
iteration 152
key 140
lookups 161
values 121, 155
Digits, decimal 101
Directory operations 178
Download 5, 7, 8, 9, 247
anaconda 8
automatic 5

E

Elements 116, 122, 124, 128, 135, 237
access 124
container 135
core 237
nested 116
searching 128
varied 122
EMP dictionary 155
Exception(s) 4, 12, 182, 183, 184, 185, 186, 188, 189, 190, 191, 192, 193
blocks 186
class 193
custom 189
explicit 189
Execution 5, 12, 43, 72
function call's 72
path 5, 43
sequence 12
Exemption provisions 186
Expressions 25, 42
evaluating 42
straightforward arithmetic 25

F

Fibonacci series 253
File 122, 164, 165, 167, 174, 176, 177
handling capabilities 164
music 122
path 164
pointer 165, 167
positions 174
processing tasks 176
type, binary 177
File name 164, 165, 173, 178
absolute 178
relative 178
File object 164, 165, 166, 173
attributes 173
File systems 164, 166
local 164
First python program 9
Floating 18, 62, 192
point errors 192
point numbers 18, 62
Flow diagram 47, 49, 57
of break statement 57

of if-elif-else statement 49
of if-else statement 47
Format 20, 80, 81, 99, 122, 161, 163, 165, 171
binary 165
decimal 122
string syntax 99
strings 81
Formatting dictionaries 161
Function(s) 38, 61, 62, 65, 66, 73, 76, 90, 113, 119, 123, 128, 132, 135, 158, 159, 164, 166, 176, 192, 200, 201, 229
linked 176
logarithmic 201
mathematical 201
parameters and arguments 66
polymorphism 229

H

Handy explanation 13
Hash tables 21, 121
High-level language program 3
Hyperparameter tuning 238

I

If 46, 47, 49
elif-else statement 49
else statements 46, 47
Imaginary components 18
Immutable strings 84
Implementation, standard 3
Indentation rules 12
Index 36, 38, 39, 88, 103, 104, 114, 124, 125, 128, 135, 138, 192, 241
negative 124
number 103
offset 114
out-of-bound 192
positions 138
positive 124
row 241
Indexing 103, 125, 142
negative 103
Information 17, 20, 132, 138, 139, 163, 165, 184, 209, 230, 233, 238
modeling 238
Inheritance 15, 193, 209, 219, 220, 221, 222, 226, 231, 236
hierarchical 220, 226
hierarchy 226
multi-level 222
process 209, 231
single 220, 221
Inheriting properties 219
Install 6, 247
dictionaries 247
NLTK 247
python 6
Installation 1, 5, 6, 8, 9, 17, 247
destination, default 5
instructions 6
of mint 6
of supporting software 6
Window 247
Installing 4, 7
Anaconda on windows 7
Python 4
Installing NLTK 247
on Windows 247
Integer 17, 18, 23, 26, 37, 62, 77, 85, 90, 103, 122, 123, 125, 127
hexadecimal 37, 85
number 62
octal 37, 85
signed decimal 37, 85
unsigned decimal 37, 85

J

Jupyter 9, 10
home 9
notebook 9, 10

L

Lambda 14, 72, 73, 148, 149
calculus 14
functions 72, 73, 148, 149
Language(s) 2, 3, 19, 61, 62, 82, 163, 208, 247, 251
flexibility 61
object-oriented 1
object-oriented system 3
portable 3
processing 247
Learning 237, 238, 239, 245
deep 245

supervised 238, 239
unsupervised 238, 239
Libraries, text processing 247
Linux systems 9
Literals 17, 35, 140
byte 35
floating-point 35
imaginary 35
Local namespace 12

M

Machine 1, 2, 238
language 2
virtual 1
Machine code 2, 3
instruction 2
Machine learning 236, 237, 241
algorithms 241
concepts 236
paradigm 237
Memory 4, 77
allocation 4
computer's 77
growth 4
management 4
Modifying dictionary 143
Module(s) 205, 207
hierarchy of 205
implementations 207

N

NASA's jet propulsion laboratory 1
Natural language processing 246

O

Object-oriented 207, 208, 219, 236
framework 219
methodology 208
programming concepts 207, 236
Open 9
Anaconda navigator 9
Open-source 3, 245
framework 245
software 3
Operands 25, 26, 28, 30, 32, 92
numeric 25
single 30
Operating system 1, 4, 5, 7, 16, 164, 178, 202
Operations 11, 14, 30, 77, 78, 104, 116, 120, 121, 127, 130, 161, 164, 166, 178, 181, 192
arithmetic 116
bitwise 30
clearing 104
Operator 19, 25, 30, 37, 103
arithmetic 25
formatted string 37
index 103
repetition 19
unary 25, 30
Ordered sequences 77, 148

P

Pin code 139
Polymorphism 15, 210, 229, 230, 231, 236
in class methods 230
Probabilistic reasoning 238
Program 237, 253, 257
for fibonacci series 253
for linear search 257
machine learning algorithms 237
Program's syntax 4
Programmer splits 238
Programming 1, 237
efficient 1
traditional 237
Programming languages 1, 3, 7, 36, 61, 132, 153, 163
object-oriented 1, 3
Programming paradigm 208
object-oriented 208
Python 1, 2, 3, 4, 5, 6, 7, 16, 21, 35, 43, 54, 139, 148, 149, 163, 181, 182, 183, 194, 195, 196, 239, 245, 247, 250, 258, 259
based library 245
codes execute 1
dictionary 139
environment 245
exception handling in 182, 194
expressions 35
file management 181
installation 6, 16
libraries 239, 250, 258
modules 195
object 21

package manager 7
processes files 163
program executes 43
programming language's syntax 54
programs 2, 183, 196, 250, 259
software foundation 1
tools 4
universe 1
usage 245
versions 1, 3, 4, 5, 6, 148, 149, 247
Python source 195, 200
code 195
file 200
Python's string 19
PyTorch 245

R

Random module 201
Recursive functions 61, 73, 74

S

Slice 88, 120
object 88
operation 88
string methods 120
Slicing 4, 77, 88, 104, 109, 125
operations 125
operator 104
process 88, 109
Software, building 208
Sony entertainment 1
Sorting 147, 150, 254
dictionaries 147, 150
elements of array in ascending order 254
Spyder platforms 8
Statements 11, 12, 13, 14, 35, 43, 44, 45, 46, 47, 48, 49, 50, 57, 193
expression 11
jump 43
Statistics module 201
String 37, 80, 85, 86, 88, 97, 98
capwords 98
constants 97
conversion 37, 85
formatting operations 85
functions 86, 88
method 80
module constants 97
Support 1, 3, 4, 245, 246
effective interface 246
polymorphism 3

T

Time series analysis 258
Tools, software development 3
Troubleshooting work 183
Tuple 19, 124, 125, 127, 130, 137
datatypes 19
elements 127, 130, 137
indexes 124
slicing 125
sorting 137
Tuple's beginning 124

U

Ubuntu 6
distribution 6
buntu system 6

V

Voice recognition 247

W

Windows search bar 7

In a landscape where programming resources can be overwhelming, Basics of Python Programming: A Quick Guide for Beginners is an accessible, user-friendly, and effective learning companion. It's an indispensable resource for anyone stepping into the vast realm of coding. It is highly recommended for beginners eager to begin their Python programming journey.

Dharmendra Singh Rajput
Vellore Institute of Technology
India

Krishna Kumar Mohbey

Dr. Krishna Kumar Mohbey is an assistant professor of Computer Science at the Central University of Rajasthan, India. He received his Bachelor's in computer application from the MCRPV Bhopal in 2006, Master's in computer application from the Rajiv Gandhi Technological University Bhopal in 2009, and Ph.D. from the Department of Mathematics and Computer Applications from the National Institute of Technology Bhopal, India, in 2015. His areas of interest are data mining, mobile web services, big data analysis, and user behaviour analysis. He has published more than 50 research articles in various journals and conferences of international repute.

Malika Acharya

Ms. Malika Acharya is a research scholar at the Central University of Rajasthan, India. She holds a bachelor's degree in Technology (B.Tech) from Amity University Noida (2019) and a master's degree in Technology (M.Tech) from Rajasthan Technical University (2021). She is currently a research scholar at Central University of Rajasthan, Department of Computer Science. Her research areas include big data analysis, data mining, machine learning, social networking, and recommendation systems. She has published research articles in various journals and conferences of international repute.