A Journey to Core Python

Experience the Applications of Tuples, Dictionary, Lists, Operators, Loops, Indexing, Slicing, and Matrices

MR. GIRISH KUMAR
DR. AJAY SHRIRAM KUSHWAHA
MS. RAJI RAMAKRISHNAN NAIR
MS. SUBHASHINY G

A Journey to Core Python

Experience the Applications of Tuples, Dictionary, Lists, Operators, Loops, Indexing, Slicing, and Matrices

Mr. Girish Kumar

Dr. Ajay Shriram Kushwaha

Ms. Raji Ramakrishnan Nair

Ms. Subhashiny G

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-55511-249

Dedicated to

My beloved parents ***Shri. Manjit Raj*** *and* ***Shrimati. Asha Rani***
and
My wife ***Monika*** *and kids* ***Vansh*** *&* ***Romil***

— Girish Kumar

My beloved father **Late. Shriram Kirani Kushwaha**
and My family

— Dr. Ajay Shriram Kushwaha

My beloved father **Late M. N. Ramakrishnan Nair,**
My family and My dear students

— Raji Ramakrishnan Nair

My beloved family and My dear students

— Subhashiny G

About the Authors

Girish Kumar holds a B.Sc. (Computer Science), PGDCA and MIT Degrees from GNDU and is a Research Scholar currently working as an Assistant Professor at Lovely Professional University. He has more than 18 years of teaching experience. He has one patent to his credit and has published more than 20 research papers in different national as well as international conferences and journals. He has authored three books published by reputed national and international publishers. He is also a Certified Academic Associate by IBM for DB2. He is an active member of IAENG- International Association of Engineers. In the field of programming, he has a good command of Fortran, C, C++, Java, Python, VC++.

LinkedIn profile: https://www.linkedin.com/in/girish-kumar-21a62a14/

Dr. Ajay Shriram Kushwaha is an Associate Professor at the School of Computer Science & Information Technology, Jain (Deemed-to-be-University), Bengaluru, India. He has completed his Ph.D. in Computer Science & Technology in OSSDP from RTM Nagpur University, Nagpur, India, in 2017. His research interests include cyber and network security, as well as blockchain security. He has over 30 research papers published in prestigious international journals and IEEE Springer Conferences. He has authored four books published by reputed national and international publishers. He has served as the Editor in Chief in the IJEDR peer-reviewed journal and also as an editorial board member in more than 10 International Journals. He is also the

International Journal of Sensors and Sensor Networks' guest editor. In conferences, he has served as the General Chair, Session Chair, and Panelist. He is a life member of the Hong Kong-based International Association of Engineers, the Indian Science Congress, and the IAENG Societies of Computer Science, AI, Data Mining, and Software Engineering. He is now supervising two students, one of whom has been conferred a Ph.D.

LinkedIn profile: https://www.linkedin.com/in/ajay-shriram-kushwaha-7b33b758/

Raji Ramakrishnan Nair holds a BCA and MCA degree and is a Research Scholar currently working as an IQAC Coordinator at Yeldo Mar Baselios College Kothamangalam. She has 17 years of teaching experience and believes that teaching is about being a friend, philosopher, and guide to her students. She is an author and philanthropist, actively involved in many social causes, which also made her students engage in relief works in Kerala mega floods and resulted in three houses being built for the flood victims.

LinkedIn profile: https://www.linkedin.com/in/raji-nair-aa7038132

Subhashiny G holds a B.Sc.(C.S.) and MCA degree. She is passionate about technology and has 1.5 years of experience as a blockchain consultant. She is currently working as an Assistant Professor in the Department of Computer Applications at Yeldo Mar Baselios College Kothamangalam. She is also working as a research assistant (part-time) at Arab Open University, Muscat, Sultanate of Oman.

LinkedIn profile: https://www.linkedin.com/in/subhashiny-g-a5a053189/

About the Reviewer

Prashanth Raghu graduated with a Master’s from the National University of Singapore and a Bachelor’s from PES Institute of Technology, Bangalore, and has a keen interest in contributing to open source technologies. His interests lie in learning new technologies and their impact on developing scalable solutions. He currently works as a Staff Software Engineer at InMobi Technologies aiming to develop solutions at scale for the advertising domain. He has been pursuing Carnatic music initially under the tutelage of Sri. Vid. S Venkatesh and currently under the guidance of Sri. Vid. H.S. Venugopal.

Acknowledgements

The completion of this book would not have been possible without the guidance and support of so many people. First and foremost, I would like to thank my parents Shri. Manjit Raj and Shrimati. Asha Rani, my better half Monika and kids Vansh and Romil for continuously encouraging me to write this book.

I express my deep gratitude to Mr. Balraj Kumar (Assistant Professor and Assistant Dean, HOD Programming Techniques Domain, School of Computer Application, Lovely Professional University), Miss Divanshi Mehra (Project Engineer at Wipro Ltd), and Miss Sukhmanjeet Kaur (Assistant System Engineer at TCS) for their valuable co-operation and assistance.

-Girish Kumar

There are a few people I would like to thank for the continued and ongoing support they have provided during the writing of this book. First and foremost, I would like to thank my Mrs. Kismati (mother), Bindu (wife) and my daughters Anahita and Anshika, for understanding and encouraging me to write the book. I could have never completed this book without their support. Second, I would like to thank Subhashiny G. and Praveen N Kumar for providing technical support to complete the book.

I am grateful to my colleague Girish Kumar & Guru Dr. S. B. Kishor for his great ideas and also to other outstanding online content and courses provided by top universities and MOOC courses that have been developed across the globe over the past few years.

My gratitude also goes to the team at BPB Publications for being supportive enough and providing us sufficient time to complete the book.

-Dr. Ajay Shriram Kushwaha

I would like to specially acknowledge and thank a few people for their continuous support and encouragement at all stages of writing this book. Firstly, I would like to thank my family, especially my daughter Thanima, for understanding and encouraging me to write the book. I could have never completed this book without their support. I would like to thank Praveen N Kumar, Data Scientist, Django developer for providing technical support to complete the book.-

I am thankful to our beloved Chairman Chev. Prof. Baby M Varghese, and Principal Prof. K M George, Yeldo Mar Baselios College Kothamangalam, who was motivating and inspiring me throughout.

I am grateful to the excellent online courses provided by top schools across the globe over the past few years.

I am very much thankful to the Technical Reviewer of this book Mr. Prashanth Raghu for his valuable suggestions.

I would also like to express my gratitude to the team at BPB Publications for their support and assistance in creating and publishing this book.

-Raji Ramakrishnan Nair

I feel I have learned a lot from the journey of this book. This will be the great treasure I am going to cherish not only in my future career but in my whole life.

I would like to take this opportunity to express my immense gratitude to the Almighty and all those people who have given their priceless support and assistance.

I would like to thank all my teachers wholeheartedly for helping me reach this stage.

-Subhashiny G

Preface

The approach adopted for learning the programming concepts using core Python is very simple. After discussing general ideas of instructions in every chapter, we explore the same concepts using programming examples. Our main objective is to learn core python programming. So we have written sufficient programs for building programming concepts for the non-programmer. This book contains 16 chapters covering core Python with concepts of programs in detail. This book will help you understand Python from scratch and help you build a career in programming.

The key features of our book are:

The focus is provided on the in-depth treatment of the basics of Python Programming.

Each programming concept is well described with the help of suitable example programs.

The history to the core of Python has been explained well, helping any naïve programmer to understand and learn Python Programming.

It covers code reusability concepts in Python.

It contains the concepts of File handling, exceptional handling, and run time error.

It consists of basic concepts of Oops in Python like Class, constructor, Inheritance, etc.

Chapter 1 focuses on Python's past, present, and future. With this chapter, the reader will understand Python, its history, benefits, features, and various applications.

Chapter 2 focuses on the basic syntax of Python programming. All these syntaxes are well explained with lots of examples, which will boost logical thinking. With this chapter, the base of python programming gets strong in every reader.

Chapter 3 covers the variables and different data types available in python programming. The concept of data types is well explained with many examples, which helps the reader understand the concept effectively.

Chapter 4 covers arithmetic and logical operators used in python programming. All these concepts are well explained with the help of examples, which helps the reader understand the usage of these operators effectively.

Chapter 5 emphasizes decision-making statements available in python programming. Different usage styles of IF statements are well explained with suitable examples, helping students develop their logical skills.

Chapter 6 emphasizes the loop statements used in python programming. All the loop statements are neatly explained with suitable examples, helping students understand which loop statements to use while doing programming tasks.

Chapter 7 unleashes the concept of numbers. The chapter focuses on the number type conversion and various functions such as mathematical function, random number function, trigonometric function, and mathematical constants.

Chapter 8 deals with the concept of strings. The chapter focuses on the operators used with strings, various string operations, and string formatting.

Chapter 9 deals with the concept of lists and various operations that can be performed with this data structure.

Chapter 10 deals with the concept of tuples and various operations that can be performed with this data structure.

Chapter 11 covers the concept of dictionaries and various operations that can be performed with this data structure.

Chapter 12 provides a detailed explanation of defining and calling functions. It also deals with important concepts such as pass by reference v/s value, recursion, function arguments, and anonymous functions.

Chapter 13 focuses on the concept of Python's highest-level program organization unit. It also deals with the importing function.

Chapter 14 lays special focus on the concept of files. It deals with the various operations performed in files, such as opening and closing files, reading and writing files, renaming and deleting files, and other file methods.

Chapter 15 elucidates the concepts of exception handling that can be used to make your programs robust. Concepts such as raising and handling exceptions, user-defined exceptions, etc., used for handling exceptions are demonstrated in this chapter.

Chapter 16 deals with OOPS concept implementation in Python programming. It mainly focuses on various built-in attributes, types of inheritance, overriding methods, etc.

We expect fruitful learning through our book **A Journey to Core Python.**

Code Bundle and Coloured Images

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

https://rebrand.ly/b32255

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

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

errata@bpbonline.com

Table of Contents

CHAPTER 1

Introduction

Introduction

Python is a strong and straightforward programming language that delivers the power and complexity of traditional compiled languages and the ease-of-use (and then some) of simpler scripting and interpreted languages.

Structure

We will cover the following topics in this chapter:

History of Python

Features of Python

Benefits of Python

Applications of Python

Objectives

The main objectives of this chapter are to provide you with an overview of Python, and discuss its benefits, its future scope and importance, and its application in the real world.

History of Python

Python came into the picture in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in Netherlands.

At the time, Guido van Rossum was a researcher with considerable language design experience with the interpreted language ABC, also developed at CWI. Still, he was unhappy with its ability, so he came up with something more. Some of the tools he visualized performed general system administration tasks, so he also wanted access to the power of system calls, which is available with distributed operating systems such as Amoeba. Even though Amoeba-specific language gave some new thoughts, a generalized language gave more sense, and the seeds of Python were sown late in 1989.

Python was extracted from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.

Features of Python

The following are the main features of Python:

Easy-to-learn: Python hardly has any keywords or reserved words, understandable organization, and unambiguous syntax, which makes the language easy to understand for a student.

Readable: Python code is distinctly set and noticeable.

Simple-to-maintain: Python's source code is relatively simple to manage.

An extensive, conventional libraries: Python's conventional library is portable and cross-platform friendly with Windows, UNIX, and Macintosh operating systems.

Hands-on Python provides a Do-It-Yourself REPL interpreter to learn the language and library features before moving them into the application. The REPL console can also write small test scripts.

Python can work on various platforms and needs an indistinguishable interface for every platform.

Extendable: Python interpreter supports low-level modules and helps programmers add or customize their tools for efficient programming.

Databases: Python provides libraries for data-driven applications to interact with all major commercial databases.

GUI programming: Python supports GUI applications. It can be produced and ported to different system calls, libraries, and windows systems, for example, Windows MFC, Macintosh, and the X Window system of Unix operating systems.

Scalable: Python caters to a better organization and upholds wide programs than shell scripting.

Python has a list of other good features as well:

It supports functional and structured programming methods and also OOPs concepts.

It provides high-level dynamic data types and supports dynamic type checking.

It supports automatic garbage collection.

It can also be used as a scripting language or compiled to bytecode for developing a wide range of applications that perform enormous operations.

Even Python scripts that are executed are compiled to Python bytecode before execution.

It provides high-level dynamic data types and supports dynamic type checking. It can be easily unified with C, C++, COM, ActiveX, CORBA, and Java using Python APIs.

Benefits of Python

The divergent application of the Python language is a result of its combination of features. It makes Python more adaptable for a wide range of applications. Some of the perks of Python programming are:

The presence of third-party modules: The Python Package Index (PyPI) comprises numerous third-party modules, enabling Python to interact with distinct languages and platforms available.

Broad support libraries: Python provides a large standard library, including internet protocols, string operations, web services tools, and operating system interfaces. Many high-level programming tasks have already been scripted into the standard library, leading to the reduction of the length of code to be written.

Open source and community development: Python language is constructed beneath an OSI-approved open-source license, making it free to use and distribute, even for commercial purposes.

Learning ease and support available: Python offers excellent readability and accessible simple-to-learn syntax, which helps beginners adopt the language effectively. The code style guidelines, such as PEP 8, provide rules to aid code formatting. Also, the distinct users and active developers made the internet resource bank rich, leading to the development and continued usage of this programming language. The Python Package Index (PyPI) comprises numerous third-party modules and enables Python to interact with distinct languages.

Handy data structures: Python holds a built-in list and dictionary data structures and is used to construct data structures with fast runtime. Moreover, it affords dynamic high-level data types, leading to reduction in code length.

Productivity and speed: Python holds an object-oriented design structure, giving a redesigned procedure control capacities, solid consolidation, text preparing capacities, and its unit testing framework. Each one of these capacities increase profitability. Python is seen as an appropriate choice for building complex multi-protocol network applications.

Applications of Python

Here are some of the application areas of Python:

GUI-based desktop applications: Python has a basic structure, modular design, rich tools for text processing, and the capacity to work on different operating systems, making it an acceptable choice for developing desktop-based applications. Several GUI toolkits like wxPython, PyQt, and PyGtk accessible assist designers in making profoundly useful Graphical User Interface (GUI). The different applications created utilizing Python incorporates are:

Image processing and graphic design Python is used to develop 2D imaging software, for example, Inkscape, GIMP, Paint Shop Pro, and Scribus. It is also used in variable proportions like in 3D animation packages such as Blender, 3ds Max, Cinema 4D, Houdini, Lightwave, and Maya.

Scientific and Computational The high speed, productivity, and scope of various tools, like Scientific Python and Numeric Python, make Python an integral part of various applications involving computations and processing of scientific data. Various software are coded in Python, such as 3D modeling software like FreeCAD and finite element method software like Abaqus.

supports the development of games with diverse modules, libraries, and platforms. There are many examples available, and one of those is PySoy. It is a 3D game engine that supports Python 3, and PyGame contains a set of modules providing functionality and a related library for the game's development. Various games are built using Python, such as Battlefield 2, Bridge Commander, and Eve Online.

Web frameworks and web applications: Python has been used to build various web frameworks, including CherryPy, Django, Pyramid, TurboGears, Bottle, and Flask. These frameworks come up with standard libraries and modules that simplify tasks such as managing content, database interactions, and interfacing with different flavors of internet protocols like HTTP, SMTP, XML-RPC, FTP, and POP. Here are some of the popular web applications based on Python:

Plone is a content management system

ERP5 is an open-source ERP used mainly in aerospace, apparel, and banking

Odoo is a blended suite of different business applications and Google App Engine

Enterprise and business applications: Python is the most suitable language for coding and has special libraries, extensibility, scalability, and easily readable syntax. Reddit, which was initially developed in Common Lips, was rescripted in Python in the year

2005. Python also came up with different functionalities for the YouTube online video sharing platform.

Operating systems: Python is often a basic part of Linux distributions. For example, Ubuntu's Ubiquity Installer and Fedora and Red Hat Enterprise Linux's Anaconda Installer are developed in Python. Gentoo Linux used Python for its package management system known as Portage.

Language development: Python has affected the development of many languages with its design and module architecture. For example, Boo language uses an object model, syntax, and indentation equivalent to Python. Additionally, the syntax of many languages like Apple's Swift, CoffeeScript, Cobra, and OCaml share the features of the Python programming language.

Prototyping: Other than being brisk and simple to learn, Python has the open-source bit of leeway of being free with the help of a huge network. This helps it to be a favored decision for the development of prototypes. Python allows faster development of its prototype with features like agility, extensibility and scalability, and ease of refactoring code. Python is suited for prototyping primarily due to its scripting nature, ease of writing code, and library support. The availability of a developer network might not be a right pointer for creating prototypes as even JAVA/C# have solid developer networks.

Conclusion

In this chapter, we focused on basic things about Python. Python is a robust and straightforward programming language that delivers the power and complexity of traditional compiled languages and the ease of use of simpler scripting and interpreted languages. Guido van Rossum developed Python in the late 1980s, and its implementation was started in December 1989. This chapter discussed the basic features of Python. Currently, it is the most popular programming language, which is why it is used in every sector to develop web and desktop applications, AI, machine learning, and blockchain. So, we can develop user-friendly applications that are more useful to people.

In the upcoming chapter, you will learn the basic syntax of the Python programming language.

Key terms

Robust

Interpreted languages

Web frameworks

Django, CherryPy, Pyramid, TurboGears, Bottle, Flask

Python Package Index

Python libraries

Anaconda installer

GUI toolkits

Questions

Explain the features of Python.

Explain the real-world applications of Python.

What is Python, and who developed it?

Compare Python with C.

Python is written in which language?

Name the extension of Python files.. Is Python a purely object-oriented programming language?

CHAPTER 2

Basic Syntax

Introduction

To become an expert-level programmer, a learner must have a strong understanding of programming concepts. The knowledge a learner gets at the initial stages makes them an expert-level programmer later. This chapter focuses on the basic syntax of the Python programming language. Python is an interpreted language that works on popular OS platforms like Windows, Linux, and Mac, and it also works on container-based hostings like Docker and managed Kubernetes. Python has simple syntax like our English language. When compared to other programming languages, Python has a syntax that allows developers to code their programs with fewer lines. The programming language was actually designed for readability. Let's look at the basic syntax of the Python programming language.

Structure

We will cover the following topics in this chapter:

Interactive mode programming

Python program output and print statement

Script mode programming

Python identifiers

Keywords

Lines and indentation

Multi-line statements

Quotations used in Python

Comments in Python

Multiple statements on a single line

Objectives

The chapter focuses on helping you understand the basic syntax of the Python programming language. So, it helps you learn about the syntax used for specifying an identifier or a keyword and discusses how to handle multi-line statements or place comments in Python, and so on.

First Python program

Let’s run the given programs in different methodologies of programming:

Interactive mode programming

The interpreter brings up the following prompt without passing a script file while invoking:

On Linux

```
$ python

Python 3.3.2 (default, Dec

[GCC on Linux

Type more  information.
```

On Windows

```
Python 3.4.3
Feb bit (Intel)] on win32
```

Type more information.

Program output and print statement

Python uses the print statement to display output on the screen. Python contains the **printf()** function to display output to the console. Most shell script languages use the echo command for program output.

```
print
```

Output:

```
Hello, Python!
```

Note: The print statement, when paired with the string format operator behaves even more like the **printf()** function in C:

```
is number %d"
```

Output:

```
python is number 1
```

Script mode programming

The execution of the script begins after the invocation of the interpreter with a script parameter and continues till the end of the script, leading to an inactive interpreter.

Let's write a simple Python program in the script. Python files support the **.py** Type the following source code in a **test.py** file:

```
>>> print ("Hello, Python!")
```

We believe that you have the Python interpreter placed in the PATH variable. Now, try to run this program on different platforms, as follows:

On Linux

```
$ python test.py
```

Output

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result:

```
Hello, Python!
```

Let’s try another approach to execute a Python script in Linux. Here’s the modified file:

```
#!/usr/bin/python3

print
```

We presume that you have a Python interpreter available in the **/usr/bin** directory. Now, try to run this program as follows:

```
$ chmod +x test.py        # This is to make file executable
$./test.py
```

This produces the following result:

```
Hello, Python!
```

Python identifiers

Identifier is the user-defined name given to entities like class, functions, variables, and so on in Python. It helps differentiate one entity from the others.

Rules for naming identifiers

The following list mentions the rules for naming identifiers in Python:

An identifier begins with a letter A to Z or a to z or an underscore (_) suffixed by zero or more letters, underscores, and digits (0 to 9).

An identifier should not begin with a digit.

Keywords must not be used as identifiers.

An identifier must not contain special characters, that is, ‘.’, ‘$’, ‘!’, ‘@’, ‘#’, ‘%’, etc.

Uppercase letters are used to start class names, and all other identifiers start with a lowercase letter.

Python is a case sensitive programming language i.e., it treats uppercase and lowercase differently.

The identifier becomes private when an identifier is prefixed with a single leading underscore.

The identifier is said to be a strong private identifier when it is prefixed with two leading underscores.

When the identifier is suffixed with two trailing underscores, it becomes a language-defined special name.

An identifier length must not exceed 79 characters.

For example: **Hello, Learn** are valid identifiers.

while are invalid identifiers.

Keywords

These are the unique words reserved for a specific purpose and must not be used as class names, variable names, function names, or any other identifiers. All the Python keywords comprise of lowercase letters only. *Table 2.1* lists some examples of keywords in Python:

Python:
Python:
Python:
Python:
Python:
Python:
Python:

Table 2.1: *Examples of keywords in Python*

Lines and indentation

Python constitutes block structure and nested block structure with indentation, not with beginning and end brackets.

The benefits of using indentation to specify structure are as follows:

It reduces the requirement for a coding standard. The crucial point is that indentation is four spaces, and no hard tabs are required.

It decreases irregularity. Code from various other sources follows the same approach of indentation.

It diminishes work. The utmost need is to get the indentation corrected, not both the indentation and brackets.

It reduces clutter as it eliminates all the curly brackets.

If it looks correct, it is correct. Indentation cannot fool the reader.

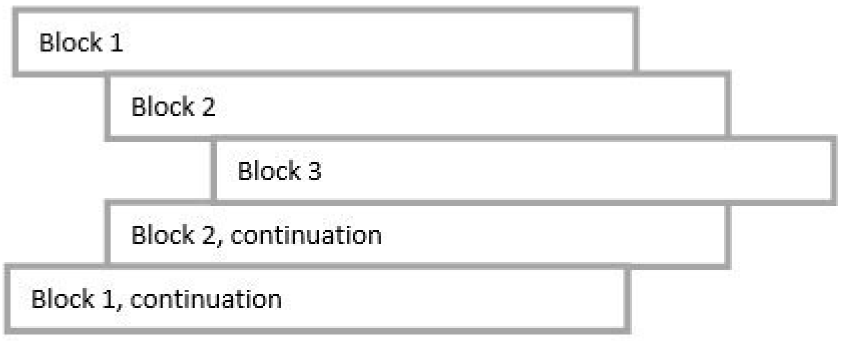

Figure 2.1: Flow of control in an indented statement

Figure 2.1: *Flow of control in an indented statement*

Multi-line statements

In Python, a new line character is used to indicate the end of the statement. However, it supports the use of the line continuation character to denote that the line should continue. Consider this example:

```
total = item_one + \

item_two + \

item_three
```

The statements contained within the or **()** brackets need not use the line continuation character. Consider this example:

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May']

Quotations used in Python

Python supports single double and triple or quotes, indicates string literals, when the same type of quote is used to begin and terminate the given string.

The triple quotes are used to stretch the string across multiple lines. Look at this example:

```
Name = 'XYZ'

Fathername = "ABC DEF"

About = """I love Python.

I can study it the whole day."""

print(Name)

print(Fathername)

print(About)
```

Output:

```
XYZ
ABC DEF
```

```
I love Python.
I can study it the whole day.
```

Note: The preceding program is in the editor, which is opened by clicking on ‘File’ and then ‘New’. After writing it, click on ‘Run’ and ‘run module’ to see the output.

Comments in Python

As with most scripting and Unix-shell languages, the hash/pound sign indicates that a comment begins right from the **#** and continues until the end of the line.

Here's an example:

```
#a is initialized to hello



#we will print hello now

print(a)
```

Output:

```
hello
```

Multiple statements on a single line

The semicolon in Python allows multiple statements on a single line with a constraint that no statement starts a new code block. Here’s an example that uses the semicolon:

a=5;   b = ’hello’;   c = “This is how we write multi statements using semicolon(;)”

Conclusion

Python can be treated as functional, procedural, or object-oriented. In this chapter, we learned about the basic syntax needed for writing code using Python. We also learned how to include multiple lines on a single line using a semicolon (;). We explored the rules used for naming an identifier in Python and understood the importance of including lines and indentation in Python.

In the next chapter, we will learn more about the types of variables used in Python. This will help decide which type of variables to use in various situations.

Questions

What is the importance of the Print function in Python?

What do you mean by interactive mode programming?

List some of the rules for naming identifiers in Python.

Explain with an example how to handle multiple line statements on a single line in Python.

Discuss the importance of lines and indentation in Python.

CHAPTER 3

Variable Types

Introduction

A **variable** is defined as a reserved memory location that holds a value that may change. It can be seen as a container to store specific values. Technically, it is a way of referring to a memory location used by a computer program and is a symbolic name for this physical location.

Based on the variable's data type, the interpreter usually allocates memory and decides what to store in the reserved memory.

Structure

We will cover the following topics in this chapter:

Variables and assignment

Multiple assignment

Built-in data types

Numbers

Strings

Lists

Tuples

Dictionary

Data type conversion

Objectives

This chapter focuses on how data is stored in a reserved memory location, assigning values to a variable and completing multiple assignments in Python. It also covers lists, tuples, and the dictionary used in the Python programming language. Additionally, it covers data type conversion in Python programming.

This chapter will help you understand the basics of variable types with the help of examples.

Variables and assignment

Python is dynamically typed, so pre-defining a variable or its type is not mandatory. The type (and value) is initialized on the assignment, and assignments are performed using the equals sign.

Here's an example:

```
is %s\nROLLNO is %d\nMARKS are %f" %(name,rollno,marks))
```

```
NAME is XYZ

ROLLNO

MARKS are 93.670000
```

The string format operator is displayed again at the end of the print statement.

Each **%x** given in the code matches the type of the argument to be printed. We have seen **%s** (for strings), **%d** (for integers), and **%f** (for floating-point values) as it is in C programming language.

Multiple assignments

Python supports the assignment of a single value to several variables simultaneously.

Consider this example:

a=b=c=1

Here, an integer object is created and assigned with the value one, and the same memory location is assigned to the three variables. Also, multiple objects can be assigned to multiple variables.

Here's an example:

a, b, c = 1, 2, "John"

Here, values 1 and 2 are assigned to two variables, a and b, whereas variable c is assigned with one string object with the value

Built-in data types

These are the standard data types that are built into the interpreter:

Numbers

Strings

List

Tuple

Dictionary

Numbers

The **number** data type stores numeric values. Number objects are created when we assign a numeric value.

```
a=5
b=27
```

The three different numerical types are as follows:

int (signed integers)

float (decimal point real values)

complex (imaginary numbers)

Note: Here, in complex numbers, j is used with an imaginary part of complex numbers. It is expressed in the form of a+bj where a and b are any real numbers.

All integers are represented as long integers, so there is no need for any separate long data type.

```
a=5
b=3.145
c=3+6j
```

Strings

Strings are identified as a continuous sequence of characters represented in single double or or triple or quotes. There is no character data type in Python, and it is a string with a length equal to 1. Triple quotes are used when we write a string in multiple lines. Subsets of strings can be seized using the slice operator or The index starts at 0 from the beginning and -1 from the end. The + sign is used for concatenation, and the * asterisk is used for repetition.

Example:

```
str = ‘Let us study

print (str)

print (str[0])

print (str[2:5])

print (str[2:])

print (str * 2)

print (str + "TEST")
```

Output:

```
Let us study Python
L
t u
t us study Python
Let us study PythonLet us study Python
Let us study PythonTEST
```

Lists

Like string, **list** is a collection. A string is a sequence of characters, not a collection by default, whereas a list can have values of any type. The values of a list are known as elements and are written in a list separated by commas in square brackets

List=[1,2.3,"list"]

The list is mutable, i.e., the value of the elements in it can be changed.

The values in the list are accessed using the slice operator or just like we did in strings.

These are some ways of accessing the list:

list =

tinylist =

print (list)

print

print

```
print

print (tinylist *

print (list + tinylist)
```

Output:

```
['abcd', 786, 2.23, 'xyz', 70.2]
abcd
[786, 2.23]
[2.23, 'xyz', 70.2]
[123, 'xyz', 123, 'xyz']
['abcd', 786, 2.23, 'xyz', 70.2, 123, 'xyz']
```

Note: Here, the value **2.3** is given as **2.2999999999998** because the values in the interpreter are actually stored as double, which contain more number of bits as compared to float. So, the approximate double value is 2.3, and there is no difference.

Tuples

The **tuple** is another sequence similar to the list. The single difference between a list and a tuple is that a list is mutable and tuple is immutable, which indicates that the elements in the tuple cannot be manipulated. Tuples can be treated as read-only lists. One more difference is that tuples are written between parentheses while lists are written in brackets Tuples are accessed just like lists but cannot be updated.

These are the ways of accessing tuples:

```
tuple =

tinytuple =

print (tuple)

print

print

print

print (tinytuple *
print (tuple + tinytuple)
```

Output:

```
('abcd', 786, 2.23, 'xyz', 70.2)
abcd
(786, 2.23)
(2.23, 'xyz', 70.2)
(123, 'xyz', 123, 'xyz')
('abcd', 786, 2.23, 'xyz', 70.2, 123, 'xyz')
```

A list is mutable and tuple is immutable; the following program illustrates this:

```
tuple =

list =

= 1000



print(list)

= 1000

print(tuple)
```

Output:

```
['abcd', 786, 1000, 'xyz', 70.2]
Traceback (most recent call last):
File "C:/Python33/LISTxx.py", line 5, in
tuple[2] = 1000
TypeError: 'tuple' object does not support item assignment
```

Dictionary

A **dictionary** is said to be a set of key-value pairs whose values are accessed by keys. We can think of it as a list, but we have integral indices in a list, while a dictionary can have any type of indices, usually numbers or strings.

Dictionaries are enclosed in curly braces and accessed through square brackets

These are the ways of performing operations on the dictionary:

```
dict = {}

= "This is three"

= "This is three"

tinydict =

print

print (dict[2])

print (tinydict)
print (tinydict.keys())

print (tinydict.values())
```

The output is as follows:

```
This is threeThis is three
{'code': 3010, 'dept': 'education', 'name': 'ymbc'}
dict_keys(['code', 'dept', 'name'])
dict_values([3010, 'education', 'ymbc'])
```

Dictionaries have no order, and they are accessed by the keys provided. The keys inside the dictionaries are given values with colon in between them.

Data type conversion

Sometimes, we need to convert the built-in types. For this, we use some built-in functions that convert one type to another, as shown in the following table:

table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table:
table: table: table: table: table: table: table: table: table: table: table: table: table: table:

Table 3.1: *List of data types*

Conclusion

In simple terms, we can say that variables are just like containers with a name or label that is used to hold the value. A variable can take data in various formats, such as a string, an integer, a number with fractional parts (float), and so on. These formats, as mentioned earlier, are called data types. Here, we learned about variables and how to assign values to variables of different types. We also learned the different ways of converting a variable of a particular data type to another data type under data type conversion.

In the next chapter, we will learn about the basic operators used in the Python programming language. The use of basic operators is explained with the help of examples for easier understanding.

Questions

Define a variable.

Explain with an example how a value is allocated to a memory location when a variable is assigned with a value.

Explain the four basic data types used in Python programming language.

Explain data type conversion in Python.

What do you mean by an implicit conversion in Python?

CHAPTER 4

Basic Operators

Introduction

Python has special symbols for operators that perform arithmetic or logical operations. The value on which the operator operates is called an operand; for example, A + B, where ‘A’ and ‘B’ are operands and ‘+’ is the operator that operates on the given operands. In this example, it is an addition operation. Several operators are used for performing different operations in Python.

Structure

We will focus on the following topics in this chapter:

Arithmetic operators

Relational operators

Python assignment operators

Bitwise operators

Logical operators

Membership operators

Identity operators

Operators precedence

Objectives

This chapter helps readers learn more about the different types of operators used in the Python programming language. A number of operators are available in Python, and each one is explained with an example. Additionally, the chapter focuses on making the concept easy to understand for learners.

Take a look at this example:

Here, **+** is the operator that performs the addition operation on the given operands **2** and The output of the operation is

Arithmetic operators

Operators are defined as constructs that can manipulate the value of operands. Arithmetic operators perform **addition, subtraction, multiplication, division, exponentiation, and modulus**

Assume that a = 25 and b = 4.

4.
4. 4. 4. 4. 4.
4. 4. 4. 4. 4.
4. 4. 4. 4. 4.
4. 4. 4. 4. 4.
4. 4. 4. 4. 4.
4. 4. 4. 4. 4. 4. 4. 4. 4. 4.

4. 4. 4. 4. 4.

Table 4.1: *Different arithmetic operators with their explanations*

Consider this example:

```
c=a+b

of %d and %d is %d" %(a,b,c))

c=a-b

of %d and %d is %d" %(a,b,c))

c=a*b

of %d and %d is %d" %(a,b,c))

c=a/b

of %d and %d is %f" %(a,b,c))

c=a%b

of %d and %d is %d" %(a,b,c))

c=a**b

of %d and %d is %d" %(a,b,c))

c=a//b

division of %d and %d is %d" %(a,b,c))
```

This is the

addition of 5 and 3 is 8
subtraction of 5 and 3 is 2
multiplication of 5 and 3 is 15
division of 5 and 3 is 1.666667
modulus of 5 and 3 is 2
exponent of 5 and 3 is 125
floor of 5 and 3 is 1

Relational operators

Relational operators compare the values on each of their two sides and decide the relation between them. They are also called comparison operators.

Assume that variable **a** holds the value five and variable **b** holds the value ten, then:

then:
then: then: then: then: then: then:
then: then: then: then:
then: then: then: then: then: then:
then: then: then: then: then:

then: then: then: then: then: then:
then: then: then: then: then:

Table 4.2: *Different relational operators with their explanations*

Here's an example:

is equal to

is not equal to

is not equal to

is equal to

is less to

is not less to

is greater to

is not greater to

is either less than nor equal to

is neither less than nor equal to

is either greater than nor equal to

is neither greater than nor equal to

This is the output:

a is not equal to b
a is not equal to b
a is less to b
a is not greater to b
a is either less than nor equal to b
a is neither greater than nor equal to b

Python assignment operators

Assignment operators are used in **Python to assign values to** For example, a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left. There are various compound operators in Python, like a += 5 that adds the value 5 to the variable a and later assigns the same to the variable a.

The following table lists the different assignment operators used in Python:

Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:

Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python:

Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:

Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:

Table 4.3: Different assignment operators with their explanations

Consider an example where variables a and b hold the values 10 and 20, respectively; then:

```
c+=a

is %d" %(c))

c-=a

is %d" %(c))

c*=a

is %d" %(c))
```

```
c/=a

is %f" %(c))

c%=a

is %d" %(c))

c**=a

is %d" %(c))

c//=a

division is %d" %(c))
```

Here’s the output:

```
addition is 8
subtraction is 3
multiplication is 15
division is 3.000000
modulus is 3
exponent is 243

floor division is 48
```

Bitwise operators

The **bitwise operator** functions on bits and performs bit-by-bit operation. Let’s take an example where a = 60 and b = 13. In binary format, they will be as follows:

a =

b =

----

a&b =

a|b =

a^b =

~a =

Python's built-in is used to get the binary representation of an integer.

integer.
integer. integer. integer. integer. integer. integer.

integer. integer. integer. integer. integer. integer. integer. integer.
integer. integer. integer. integer. integer. integer. integer. integer.

integer. integer. integer. integer. integer. integer. integer. integer. integer. integer. integer. integer. integer. integer. integer. integer.
integer. integer. integer. integer. integer. integer. integer.
integer. integer. integer. integer. integer. integer. integer.

Table 4.4: *Different bitwise operators with their explanations*

Here's an example:

```
a = 60 = 0011 1100

b = 13 = 0000 1101

print

c = 0

c = a & 12 = 0000 1100

print of AND is

c = a | 61 = 0011 1101
```

```
print of OR is
```

c = a ^ 49 = *0011 0001*

```
print of XOR is
```

The output is as follows:

```
a= 60 : 0b111100 b= 13 : 0b1101
result of AND is 12 : 0b1100
result of OR is 61 : 0b111101
result of XOR is 49 : 0b110001
result of COMPLEMENT is -61 : -0b111101
```

Logical operators

The following are the **logical operators** supported by Python language. Assume that variable **a** holds the value True and variable **b** holds the value False.

False.
False. False. False. False. False.
False. False. False. False. False.
False. False. False. False. False.

Table 4.5: *Different logical operators with their explanations*

Consider this example:

```
a(False or True):

b(False or True):

AND b is and b)

OR b is or b)

a is a)
```

This is the output:

```
enter a(False or True): True
enter b(False or True): True
a AND b is false
a OR b is true
NOT a is False
```

Membership operators

Python's **membership operators** are used to test the membership in a sequence, such as in strings, lists, or tuples. There are two membership operators:

operators:
operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators:
operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators:

Table 4.6: *Different membership operators with their explanations*

Here's an example:

a number:

a character:

python"

```
l1=

if (a in l1):

print is available in the given



print is not available in the given

if (b s1):

print is not available in the given




print is available in the given
```

The output is as follows:

```
enter a number: 34
enter a character: e
34 is not available in the given list
e is available in the given string
```

Identity operators

Identity operators differentiate between the memory locations of two objects. There are two identity operators:

operators:
operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators:
operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators: operators:

Table 4.7: *Different identity operators with their explanations*

Consider this example:

```
a = a:

b = b:

c = c:

if (a is b):


print and b have same
print and b do not have same

if (id(a) == id(b)):

print and b have same



print and b do not have same

if (a c):

print and c do not have same



print and c have same
```

The output is as follows:

```
enter a: 2
enter b: 2
enter c: 3
a and b have same identity
a and b have same identity
a and c do not have same identity
```

Operators precedence

The following table describes all the operators from the highest precedence to the lowest:

lowest:
lowest: lowest: lowest: lowest: lowest:
lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest: lowest:
lowest: lowest: lowest: lowest: lowest: lowest:
lowest: lowest: lowest:
lowest: lowest: lowest: lowest: lowest:
lowest: lowest:
lowest: lowest: lowest: lowest: lowest: lowest:
lowest: lowest:
lowest: lowest:
lowest: lowest:
lowest: lowest:
lowest: lowest:
lowest: lowest:

Table 4.8: Operators precedence

Operator precedence affects the process of an expression's evaluation.

For example, x = 5 + 3 * 2; here, x is assigned to 11, not 16. This is because the operator multiplication (*) has higher precedence than addition (+), so it first multiplies 3*2 and then adds 5.

Here, the operators at the top of the table have the highest precedence, and those at the bottom have the lowest precedence.

Here's an example:

```
a = 10

b = 15

c = 20

d = 25

ans = (a + b) * c / d

print of (a + b) * c / d is ans)
```

```
ans = ((a + b) * c) / d

print of ((a + b) * c) / d is ans)

ans = (a + b) * (c / d)

print of (a + b) * (c / d) is ans)

ans = a + (b * c) / d

print of a + (b * c) / d is ans)
```

The output is as follows:

```
Value of (a + b) * c / d is 20.0
Value of ((a + b) * c) / d is 20.0
Value of (a + b) * (c / d) is 20.0
Value of a + (b * c) / d is 22.0
```

Programs

Write a program to calculate the area and perimeter of a square:

side

area=a*a

of square

of square

The output is as follows:

```
enter side 5
area of square is 25
perimeter of square is 20
```

Write a program to calculate the average of three numbers:

1st

2nd

3rd

is:

This is the output:

```
enter 1st number34
enter 2nd number12
enter 3rd number22
average is: 22.666666666666668
```

Write the code to calculate the area of a circle:

1st

2nd

3rd

is:

Here’s the output:

```
enter radius4
area of circle is 50.24x
```

Write a program to calculate the square of a number:

```
a

sqr=a*a

of %d is
```

The output is as follows:

```
enter a number4
square of 4 is 16
```

Write a program code to convert temperature from Fahrenheit to Celsius:

```
temperature in Fahrenheit:

temp=float(temp)



in Celsius is
```

This is the output:

```
enter temperature in Fahrenheit: 98
Temperature in Celsius is : 36.666666666666664
```

Write a program to swap two numbers using a third variable:

```
# to swap 2 numbers

first number:

second number:




c=a

a=b


b=c
```

Here's the output:

```
Enter first number: 12
Enter second number: 60
Before swapping
a= 12
```

```
b= 60
After swapping
a= 60
b= 12
```

Write a program to swap two numbers without using a third variable:

```
first number:

second number:




a=a+b

b=a-b

a=a-b
```

This gives the following output:

```
Enter first number: 23
Enter second number: 56
```

```
Before swapping
a= 23
b= 56
After swapping
a= 56
b= 23
```

Write a program to convert kilometers into miles:

```
# Program to convert kilometers into miles

kilometers:

mul_value = 0.621371

miles = km * mul_value

kilometers is equal to %0.
3f miles' %(km,miles))
```

The output is as follows:

```
Enter kilometers: 34
34.000 kilometers is equal to 21.127 miles
```

Conclusion

In this chapter, we learned about the different types of operators, like arithmetic operators, relational operators, and bitwise operators. We also looked at examples of these operators for better understanding.

In the next chapter, we will cover decision-making statements.

Questions

Explain arithmetic operators in Python with examples.

What does operators precedence mean?

What is the importance of operators precedence in arithmetic operations?

Explain bitwise operators with examples.

Explain Python assignment operators with examples.

Compare identity operators and membership operators.

CHAPTER 5

Decision Making

Introduction

In a programming language, the decision-making statements determine the direction of the program execution flow.

The decision-making concept comes in a program when the program has conditional choices to execute code blocks. Let's take an example of traffic lights lit up at different conditions based on the road rules or any particular rule. The outcome that occur while executing gets evaluated with the result of either TRUE or FALSE, which is assumed as a non-zero or non-null value if it is TRUE and zero or null value if it is FALSE.

These are logical decisions, and Python provides decision-making statements to help make decisions within an application program based on the user requirements.

requirements.
requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements.
requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements.

requirements. requirements. requirements. requirements. requirements. requirements. requirements.
requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements. requirements.

Table 5.1: *This table is all about condition statements in Python*

Structure

We will focus on the following topics in this chapter:

if statement

if..else statement

elif statements

Nested if statements

Objectives

There are situations in our lives when we need to make a decision and take the next steps accordingly. Similar situations arise in a programming language, and we need to make decisions here as well. Based on the decision, we execute the next block of code.

if statement

In this, the statement starts with an **if** reserved word. The condition in the statement is a Boolean expression that determines whether or not the body will be executed. A colon must follow the condition. The block is defined as one or more statements that are executed when the condition is true. The statement in the **if** block is a Boolean expression, which determines whether or not the block of statements will be executed.

Syntax:

If expression:
statement(s)

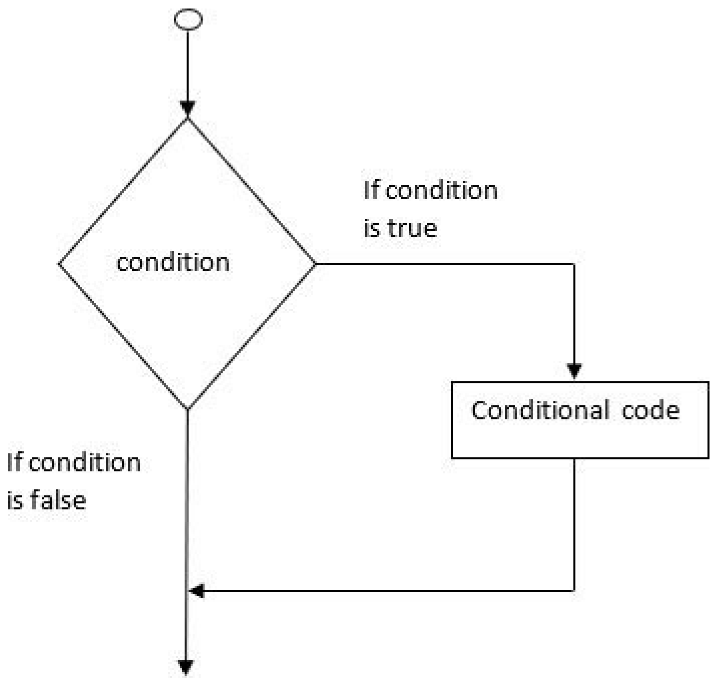

Figure 5.1: *Flow diagram of the working flow of an if-statement*

Here's an example:

value of divisor is: ", c)

This is the output:

The value of divisor is: 3

if..else statement

With **if else** statements, **if** is the reserved word, and the code block under it is executed if its boolean expression evaluates to TRUE; otherwise, the code block under the **else** statement is executed.

Syntax:

```
If expression:
statement(s)
else:
statement(s)
```

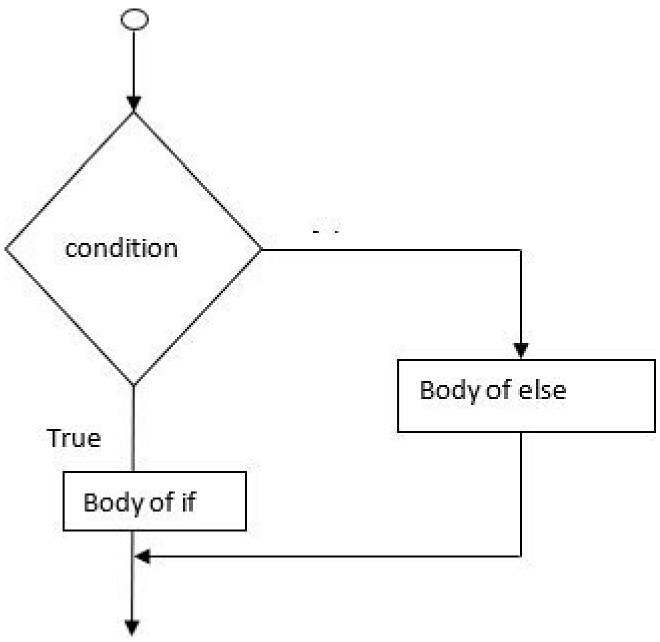

Figure 5.2: *Flow diagram of the working flow of an if-else statement*

Here’s an example:

a number:

an even

an odd

The output is as follows:

```
enter a number: 34
34 is an even number
```

elif statements

The **elif** statement helps you evaluate multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Syntax:

```
if expression1:
statement(s)
elif expression2:
statement(s)
elif expression3:
statement(s)
else:
statement(s)
```

The following figure shows the working flow of an elif statement:

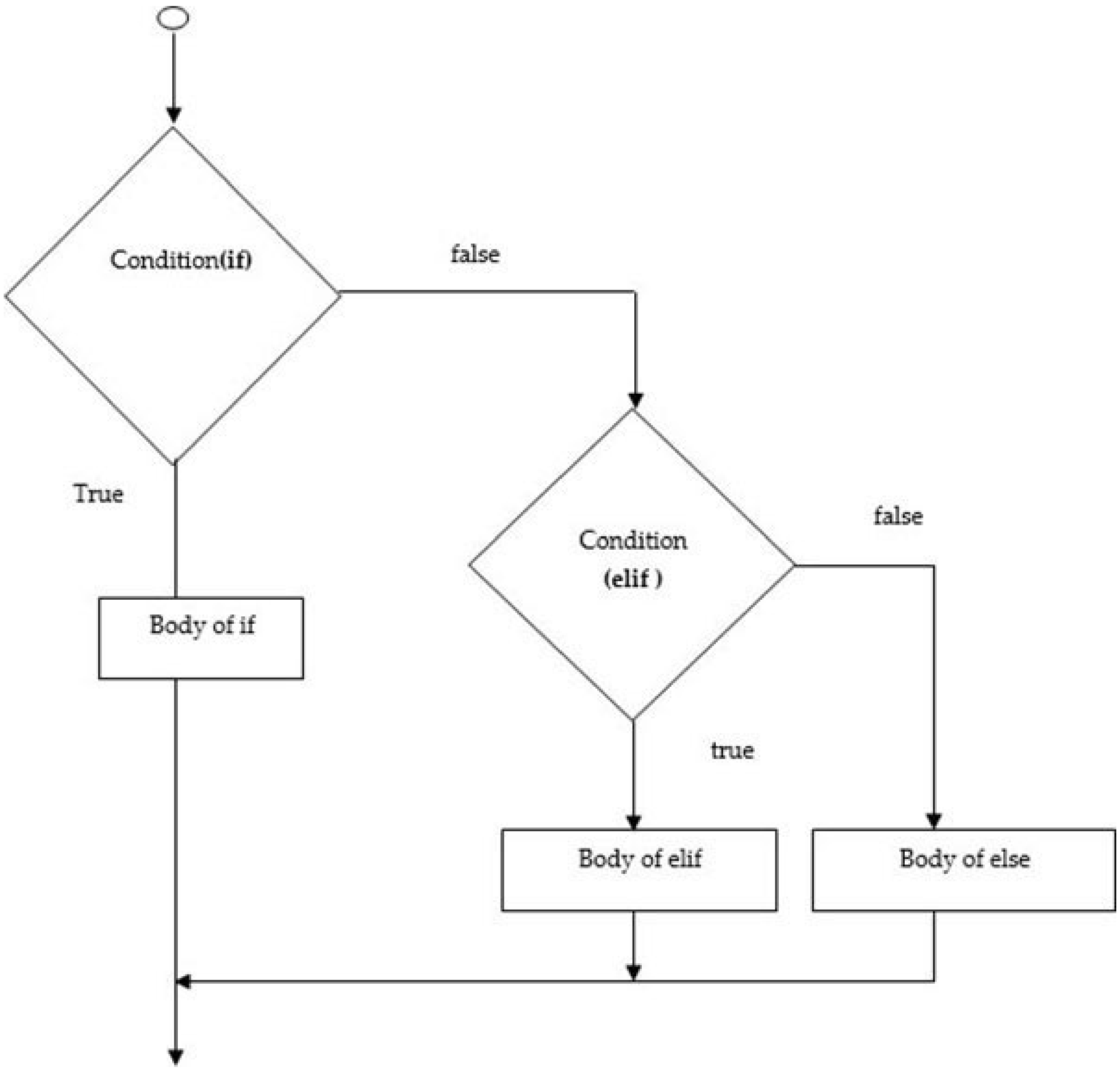

Figure 5.3: *Flow diagram of the working flow of an elif-statement*

Consider this example:

```
a = a

b = b:

if a > b:
```

is

elif a == b:

are

is

It will give the following output:

```
enter a: 22
enter b: 12

a is greater
```

Nested if statements

We require conditions inside conditions sometimes, and we use **if** under **if** or **if...elif...else** under **if...elif...else** or other nested conditions at such times.

Syntax:

```
if expression1:
statement(s)
if expression2:
statement(s)
elif expression3:
statement(s)
else
statement(s)
elif expression4:
statement(s)
else:
statement(s)
```

Take a look at this example:

```
if

if
```

```
print by 3 and


print by 2 not divisible by


if

print by 3 not divisible by


Divisible by 2 not divisible by 3"
```

The output is as follows:

```
enter number 8
divisible by 2 not divisible by 3
```

Single statement suites

When the suite of an if clause contains a single line, it may go on to the same line as the header statement.

Here's an example:

```
var = 100

if == : print of expression is

print
```

This is the output:

```
Value of expression is 100
Good bye!
```

Programs

Program to check whether a number is positive or negative:

```
a number:

if n>0:

is


is


is
```

The output is as follows:

```
enter a number: 23
positive
```

Program to check whether a number is odd or even:

```
a number:
```

an even

an odd

Here's the output:

```
enter a number: 23
23 is an odd number
```

Program to check whether or not the year is a leap year:

year to be checked:

year is a leap

year is not a leap

It gives the following output:

```
Enter year to be checked: 2018

The year is not a leap year.
```

Program to find the greatest among two given numbers:

```
the first number:

the second number:

if a>b:


elif(a==b):
```

The output is as follows:

```
enter the first number: 23
enter the second number: 12
23 is greatest
```

Program to find the greatest among three given numbers:

```
the first number:

the second number:

the third number:
```

```
if

"are

and

elif  b>a and

and
```

Here's the output:

```
enter the first number: 23

enter the second number: 56
enter the third number: 44
56 is greatest
```

Program to input marks in 5 subjects and display the grade:

```
marks in 1st subject:

marks in 2nd subject:
```

```
marks in 3rd subject:

marks in 4th subject:

marks in 5th subject:


if


and


and


and

print("Grade D")

else:

print("Grade fail")
```

The output is as follows:

```
enter marks in 1st subject: 89
enter marks in 2nd subject: 78
enter marks in 3rd subject: 60
enter marks in 4th subject: 90
enter marks in 5th subject: 87
80.8
```

Conclusion

The if statement executes a block of statements if a condition is true and does not execute them if the condition becomes false. But what if we want to perform some other actions if the condition is false. This is where the else statement comes in. The else statement with the if statement is mainly used to run a block of code whenever the condition becomes false. A user can decide multiple options, but the if statement is executed from top to bottom. If one of the conditions controlling the if condition is true, the statement associated with it is executed, and the else statements are bypassed. If none of the conditions is true, the final nested statement is executed.

In the next chapter, we will learn about repeating code using loops in Python.

Key terms

Condition statement

Nested statement

Block of code

Elif

Decision making

Syntax

CHAPTER 6

Repeating Code Using Loops

Introduction

Generally, statements execute in sequence. However, you may sometimes need to run a block of code several times, and you can use loops at such times.

Loops are statements that repeat an action over and over.

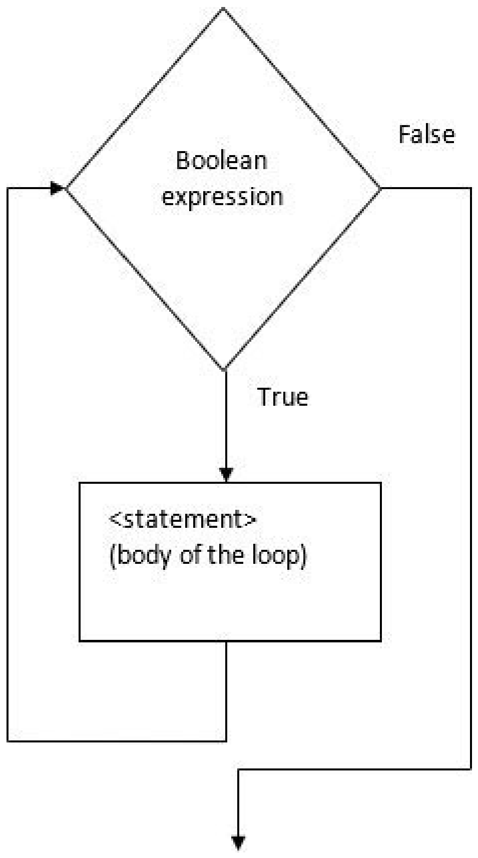

Figure 6.1: *Flow of the primary loop*

The control flow goes to the loop's body and executes the statement if the Boolean expression is True in the preceding figure; otherwise, it exits from the loop.

Structure

We will cover the following topics in this chapter:

While loop

For loop

Nested loop

Infinite loop

Objective

Python for loops iterate over an object until it is complete. For example, you can iterate over the contents of a list or a string. For loops use the syntax for an item in an object, where the object indicates the iterable over which you wish to iterate. You can repeat similar operations in your code by using loops. In this chapter, we will discuss looping concepts and working flow.

While loop

Python supports the **while** loop, which is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax:

while expression>: # as long as the expression evaluates to True
block of code>

Note that this syntax is parallel to an **if** statement. The **while** statement contains a Boolean expression followed by a colon, and there is an indented block of code after the **while** statement. The block of code is referred to as the body of the loop, and the **while** statement and the indented block together are called a while loop. The statements in the loop's body are repeated until the Boolean expression in the while statement evaluates to True. If the Boolean expression evaluates to False when executing the while statement, the execution continues with the next immediate statement after the body of the loop.

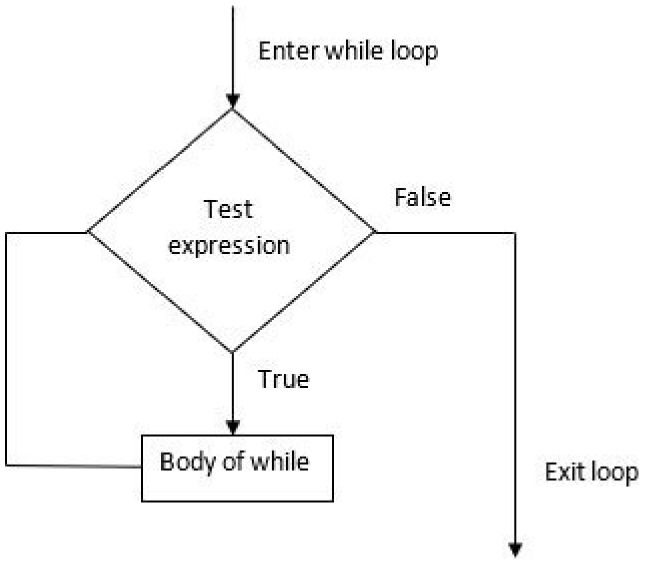

Figure 6.2: *Block diagram of while loop*

Example:

Output:

```
Hello
Hello
Hello
Hello
Hello
```

For loop

The **for** loop is a common iterator in Python. It can step through the items in an ordered sequence or other iterable objects. The **for** loop statement supports strings, lists, tuples, and other built-in iterables as well as new user-defined objects.

Syntax:

For iterating_var in sequence:
Statements(s)

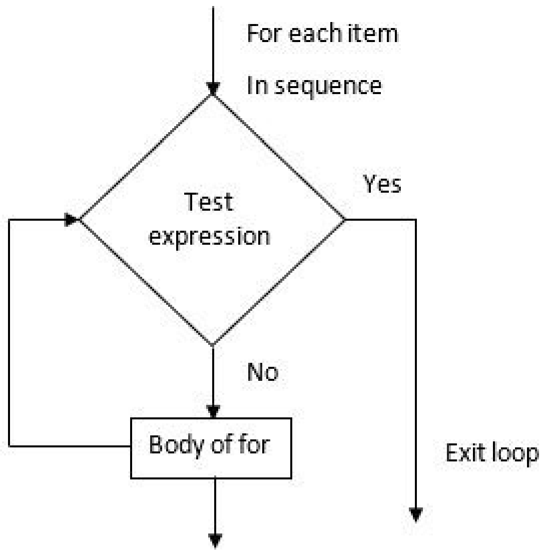

Figure 6.3: *Block diagram of for loop*

Example:

# List of numbers

```
numbers = [6,5,3,8,4,2,5,4,11]

# variable to store the sum

= 0

# iterate over the list

in numbers:



= sumtotal+totalval

# print the sum print("The sum is",sumtotal)

",sumtotal)
```

Output:

```
sum is 48
```

Range() function

We can generate a sequence using the **range()** function. The range (10) will generate numbers from 0 to 9 (10 numbers). The common format of the range function call is as follows:

range(begin,end,step)

Here,

begin is the initial value given in the range; the default value becomes zero if this is not included.

end is the value that comes after the last value in the range; the end value is not deleted.

step indicates the amount to increment or decrement; it defaults to 1 (increments by one) if the change parameter is omitted.

The values in begin, end, and step must be integer values; floating-point values and other types are not allowed.

Example:

```
for i in
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

In the preceding example, the loop will begin from 0 and go up till n. The **range()** function excludes the upper range, so we need to write (n+1) instead of n.

Break, continue, pass, and the loop else

In Python:

Break: Allows you to jump out of the closest enclosing loop (past the entire loop statement)

Continue: Jumps to the top of the closest enclosing loop (to the ' 'loop's header line)

Pass: Does nothing at all; ' 'it's an empty statement placeholder

Loop else block: Runs if the loop is exited normally (i.e., without hitting a break)

Break

Python supports the break statement to implement middle-exiting control logic. The break statement leads to an immediate exit from a loop.

Example:

```
n=1




break



    n+=1
```

Output:

```
1
2
3
4
```

The **break** statement is used sparingly because it introduces an exception to the normal control logic of the loop. Preferably, every loop should have a single entry point and a single exit point. Programmers generally use **break** statements within **while** statements with conditions that are not always true. In such a **while** loop, including a **break** statement provides an additional exit point (the one point is the top of the loop where the condition is checked, and the break statement is another). Using multiple break statements within a single loop is particularly dubious and should be avoided.

Continue

Python supports the **continue** statement, which returns the control to the beginning of the current loop. When the continue is found, the loop starts the next iteration without executing the remaining statements in the current iteration. The **continue** statement is used in both **while** and **for** loops.

Example:

```
n=0



    n+=1



continue
```

Output:

```
1
2
3
4
6
```

7
8
9
10
11

Pass

Python supports which is a null statement. The interpreter differentiates a comment and passes by completely ignoring a comment and supporting However, nothing happens when it is executed, and it results in no operation (NOP).

Pass statements are used when your code will eventually execute but has not been drafted yet, that is, in stubs. The usage of pass statements for stubs is an excellent example within Python.

Syntax:

```
pass
```

Example:

```
for i in

if

is a pass
```

Output:

```
h
this is a pass statement
e
l
l
o
```

Loop else

Python supports the inclusion of an **else** statement in a loop statement.

If the **else** statement is included in a **for** loop in Python when the loop has exhausted iterating the list, the else statement is executed then.

If the **else** statement is included in a **while** loop, it is executed when the condition becomes false.

Example:

```
n=1




    n+=1
```

Output:

```
1
```

2

3

4

5

hello

Nested loops

Python programming language supports the use of one loop inside another.

Syntax:

```
For iterating_var in sequence:
For iterating_var in sequence:
Statements(s)
Statements(s)
```

Example:

```
for i in

for j in
```

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

The **print()** function’s inner loop includes end =' which adds a space rather than a default new line. Hence, the numbers will pop up in one row.

The last **print()** executes at the end of the inner for loop.

Infinite loop

An **infinite loop** exists when the condition in the loop never becomes FALSE. You must be careful when using while loops as there is a risk that this condition never resolves to a FALSE value, which leads to a situation where the loop never ends. Such a loop is an infinite loop.

Example:

```
#this constructs an infinite loop

a

entered
```

Output:

```
Enter a number12
you entered 12
Enter a number 34
you entered 34
Enter a number12
you entered 12
Enter a number34
```

```
you entered 34
Enter a number55
you entered 55
Enter a number
Traceback (most recent call last):
File "infinite_loop.py", line 3, in
a=int(input("Enter a number"))
KeyboardInterrupt
```

The preceding example leads to an infinite loop, and you need to manually exit the program using CTRL + C or operating system exit controls.

Iterator and generator

An **iterator** is an object that permits any programmer to visit all the elements of a collection, regardless of its specific implementation. Python allows an iterator object to implement two methods: **iter()** and

String, list, or tuple objects are used to create an iterator.

A generator is a function that uses the yield method to produce or yield a series of values.

The **generator** function returns a generator object without even starting execution. When the **next()** method is called for the first time, the function begins executing until the yield statement is reached, which returns the yielded value. The **next()** method remembers the last execution due to this, so the second execution proceeds from the last execution.

Iterating examples

Program to find factorial of a number:

a number:

```
for i in

    fact=fact*i

of %d is
```

Output:

```
Enter a number: 5
Factorial of 5 is 120
```

Program to input a number and print its table:

```
a number:

limit:

of

for i in



X %d =
```

Output:

```
Enter a number: 4
Enter limit: 10
```

```
Table of  4
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
4 X 10 = 40
```

Program to determine whether a number is prime:

```
a number:


for i in


break


is a Prime


is not a Prime
```

Output:

```
Enter a number: 23

23 is a Prime number
```

Program to print all prime numbers between two numbers m and n:

```
the lower limit:

the upper limit:

numbers between %d and %d are:

for i in


for j in



break
```

Output:

```
Enter the lower limit: 2
Enter the upper limit: 23
Prime numbers between 2 and 23 are:
2
3
5
7
11
13
17
19
23
```

Program to determine whether the number is Armstrong:

```
a number:

l=len(a)

a=int(a)

orig=a

sum=0
```

```
while(a>0):

    temp=a%10

    sum=sum+(temp**l)

    a=a//10

if(sum==orig):



else:

an Armstrong
```

Output:

```
Enter a number: 153
Armstrong number
```

Program to print the Fibonacci series:

```
a
```

Fibonacci series

```
    t=f+s

    f=s

    s=t
```

Output:

```
Enter a number 5
The Fibonacci series is
0 1 1 2 3
```

Code to print the sum of the digits of a number:

a number

```
    temp=a%10
    sum=sum+temp

of the digits is
```

Output:

```
Enter a number :345
Sum of the digits is   12
```

Program to determine whether a number is a palindrome:

```
a number

orig=a

sum=0

while(a>0):

    temp=a%10

    sum=sum*10+temp

    a=a//10

if(sum==orig):
```

```
else:
```

Output:

```
Enter a number :343
Palindrome
```

Program to print the following pattern:

```
*
* *
* * *
* * * *
```

```
for i in

for j in
```

Program to print this pattern:

```
1
2 2
```

```
3 3 3
4 4 4 4
5 5 5 5 5
```

```
for i in

for j in
```

Program to print the pattern as follows:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
for i in

for j in
```

Program to print the following pattern:

```
*
**
***
****
*****
****
***
**
*
```

```
for i in

for j in



for i in

for j in
```

Program to print the pattern shown here:

```
*
```

```
* * *
* * * * *
* * * * * * *
```

```
for i in

for k in


for j in
```

Program to print this pattern:

```
1
23
456
78910
```

```
for i in

for j in
```

Program to print the Fibonacci series using iterator and generator:

```
import sys

#generator function

    a, b, counter = 0


if (counter > n):

return

yield a

        a, b = b, a + b

        counter += 1

f = #f is iterator object




print (next(f),
except Stop Iteration:

sys.exit()
```

Output:

```
0 1 1 2 3 5
```

Conclusion

The manual executes instructions until it gets the accurate outcome, using loops to provide a shortcut to minimize the code, increase its readability, and reduce complexity. Loops and statements hold various components in any programming language. Loops help alter the conventional flow of the program, execute specific sets of instructions with developer-specified conditions, and produce optimal results. The working of a loop depends upon the condition specified. In loops, a group of instructions tends to repeat itself until the termination condition occurs. After that, the program revises the already devised relevant action. So, loops provide an opportunity to bring similar executable instructions together. This technique makes the code reusable for the same tasks.

In the next chapter, we will look at numbers in Python.

Multiple Choice Questions

What will be the output of the following Python program?

```
For(I in range(0,2,-1)
Print(“Hello”)
```

No output

Hello

Hello Hello

Error

Which type of iteration is carried out in Python with a *while* loop?

Definite

Indefinite

Discriminant

Indeterminate

Predict the output of the following code:

```
value = "abcdef"

i = "i"
while i in value:
print(i, end=" ")
```

iiiii

a b c d e f

abcdef

No output

Python for loop is:

Entry Control Loop

Exit Control loop

Simple Loop

None of the above

Which of the following is an Infinite loop

While(1)

While(infinite)

For(1)

None of the above

CHAPTER 7

Numbers

Introduction

Number data types store numeric values and are used in almost every program. Favorably, Python has several different types of numbers to fit in all your game or application programming needs.

Structure

We will cover the following topics in this chapter:

Types of numbers

Type conversions

Mathematical functions

Objectives

We will discuss the different types of numbers used in Python, learn how to convert one type into another, and understand the mathematical operations supported by Python.

Types of numbers

The three types of numbers are:

Integers

Floating-point numbers (or floats)

Complex numbers

Integers (signed integers): These are whole numbers or numbers with no fractional part. Another way to think about them is that they can be written without a decimal point. Python 3 supports integers of unconditional sizes but does not support 'long integers'. Python 2 has the *int* and *long* integer types. The numbers 1, 27, -100, and 0 are all examples of integers.

Float (floating point real They are also known as floats and mean real numbers. They are written with a decimal point that divides the integer and fractional parts. Floats may also be represented in scientific notation, with E or e that indicates the power of 10 (3.5e2 = 3.5 x = 350). Numbers like 2.376, -99.1, 1.0, and -32.54e100 are examples of floats.

Complex (complex Complex numbers are of the form x + yZ, where x and y are floats, and Z (or z) represents the square root of -1 (which is an imaginary number). The real part of the number is x, and the imaginary part is y. Python rarely uses complex numbers. Some examples are 3.14z, -.6545+0z, and 3e+26Z.

Numbers are immutable (unchangeable) values that support numeric operations. This implies that changing the value of a number of particular data type results to a newnumber.Number type conversion

Python internally converts the numbers in expressions from mixed types to a common type for evaluation. In order to satisfy the requirements of an operator or function parameter, you sometimes need to explicitly convert a number from one type to another.

Type **int(m)** to convert m to a plain integer.

Type **long(m)** to convert m to a long integer.

Type **float(m)** to convert m to a floating-point number.

Type **complex(m)** to convert m to a complex number with real part m and imaginary part zero.

Type **complex(m, n)** to convert m and n to a complex number with real part m and imaginary part n, where m and n are numeric expressions.

Mathematical functions

Python programming contains a math module that provides most of the familiar mathematical functions to implement. We have to import the module to use it in programming:

```
import math
```

To access any of the available functions, you must provide the module name and the function name separated by a dot (also known as a period). This format is called dot notation.

```
Math.function_name
```

Number abs() method

The **abs()** method gives the absolute value of that is, the positive distance from **x** to zero, as output.

Syntax:

The **abs()** method has the following syntax:

abs(x)

Parameters

x - Indicates a numeric expression.

Return value

This method provides output as the absolute value of x.

Example:

```
>>>
```

```
34.55
```

```
>>>

45

>>>

467

>>>

33.55
```

Number ceil() method

The **ceil()** method provides output as the ceiling value of that is, the smallest integer, which should not be less than

Syntax:

The syntax for the **ceil()** method is given as follows:

```
import math.ceil(m)
math.ceil
```

Note: The ceil() function is not directly accessible. In order to access the ceil(), we must import the math module and then use the math static object to call this function.

Parameters

m – This refers to a numeric expression.

Return value

This method gives an output of the smallest integer not less than m.

Example:

The following example explains the usage of the **ceil()** method:

```
math

>>>

13

>>>

-34

>>>

279
```

Number exp() method

The **exp()** method gives the output as e raised to the power of m.

Syntax:

The syntax for the **exp()** method is as follows:

```
import math
math.exp(m)
```

Note: The exp() function is only accessible using the exp static module object.

Parameters

m - This is a numeric expression.

Example:

The use of **exp()** is shown as follows:

```
>>>
```

```
3.4934271057485095e+19

>>>

2.4150062132629406e-20

>>>

23.140692632779267
```

Number fabs() method

The **fabs()** method provides the output of the absolute value of float It seems similar to the **abs()** function, but there are some differences between the two:

The **abs()** method is a built-in function, whereas fabs() is defined in the math module.

The **fabs()** function only takes on float and integer values, whereas abs() works with complex numbers as well.

Syntax:

The following is the syntax for the **fabs()** method:

```
import math
math.fabs(m)
```

Note: The fabs() function is only accessible using the fabs static module object.

Parameters

m - This is a numeric value.

Return value

This method returns the absolute value of m.

Example:

The use of the **fabs()** function is shown here:

```
>>>




>>>



>>>





3.141592653589793
```

Number floor() method

The floor() method returns the floor of m as an integer, that is, the largest integer not greater than m.

Syntax:

The following is the syntax for the **floor()** method:

```
import math
math.floor(m)
```

Note: The floor() function is only accessible using the floor static module object.

Parameters

m - This is a numeric expression.

Example:

```
>>>
```

```
>>>

39

>>>

-37

>>>

-57
```

Number log() method

The **log()** function returns the logarithm of **x** to the given base. If the base is not specified, it returns the natural logarithm (base e) of m.

Syntax:

The following is the syntax for the **log()** method:

```
import math
math.log(m)
```

Note: The log() function is only accessible using the log static module object.

Parameters

m - This is a numeric expression.

Return value

This method returns the natural logarithm of m, for m > 0.

Example:

The **log()** method is used as follows:

```
>>>

3.1210424645194377

>>>

4.609660091260944

>>>

1.1447298858494002
```

Number log10() method

The **log10()** method returns base-10 logarithm of m for m > 0.

Syntax:

Here's the syntax for the log10() method:

```
import math
math.log10(m)
```

Note: The log10() function is only accessible using the log10 static module object.

Parameters

m - This is a numeric expression.

Return value

This method returns the base-10 logarithm of m for m > 0.

Example:

The use of the log10() method is shown here:

```
>>>

1.3554515201265174

>>>

2.001949941084268

>>>

0.49714987269413385
```

Number modf() method

The **modf()** method outputs the fractional and integer parts of m in a two-item tuple. Both parts have the same sign—m. The output of the integer part is a float.

Syntax:

The following is the syntax for the **modf()** method:

```
import math
math.modf(m)
```

Note: The modf() function is only accessible using the modf static module object.

Parameters

m - This is a numeric expression.

Example:

The following example explains the use of the **modf()** method:

```
>>>
```

```
>>>
```

```
>>>
```

```
>>>
```

Number pow() method

The **pow()** function in Python returns the power of a given number. It computes In this function, the arguments are converted to floats before the power is calculated.

Syntax:

pow(u, v, w)

Return value

This method provides the value of **u**v** (with two arguments) or **u**v % w** (with three arguments).

Example:

The use of the **pow()** method is as follows:

```
>>>

64

>>>
```

```
128
>>>
2
>>>
0
```

Number sqrt() method

The **sqrt()** method gives the square root of **m** for m > 0.

Syntax:

Here's the syntax for the **sqrt()** method:

```
import math
math.sqrt(m)
```

Note: The sqrt() function is only accessible using the sqrt static module object.

Parameters

m - This is a numeric expression.

Example:

The use of the **sqrt()** method is shown here:

```
>>>
```

```
4.0
```

```
>>>

4.795831523312719

>>>

1.7724538509055159
```

Number round() method

The **round()** method is a built-in function in Python. It rounds a number to a given precision in decimal digits (default 0 digits). This returns an int when called with one argument; otherwise, it returns the same type as the number. Additionally, n-digits may be negative.

Syntax:

The following is the syntax for the **round()** method:

round(m, n)

Parameters

m - It indicates an expression that must be numeric.

n – It indicates the number of digits from the decimal point up to which m is to be rounded. The default is 0.

Return value

This method provides m rounded to n digits from the decimal point.

Example:

The use of the **round()** method is shown as follows:

```
56

23.34

100.0

30.0

23.345

23.3
```

23.0

Random number functions

Python supports random numbers. Many applications, such as games, simulations, testing, security, and privacy applications, use random numbers.

Number choice() method

The **choice()** method outputs a random item from a list, tuple, or string.

Syntax:

The following is the syntax for the **choice()** method:

```
import random
random.choice(sq)
```

Note: The choice() function is only accessible using the choice static module object.

Parameters

sq – It could be a list, tuple, or string.

Return value

This method provides a random item as an output.

Example:

The following example shows the usage of the **choice()** method:

```
world"


'l'


2


'prom'


'e'
```

This function doesn't work for sets because sets do not have indices.

Number randrange() method

The **randrange()** method gives a randomly selected element from range **stop,**

Syntax:

randrange ([start,] stop [,step])

Note: The randrange() method is not directly accessible. In order to access the randrange(), we must import the random module and use the random module object to call this function.

Parameters

start -Start point of the range. This would be included in the range; the default is 0.

stop -Stop point of the range. This would be excluded from the range.

step -Value with which the number is incremented. The default is 1.

Example:

random

2) :

: 11

:

: 86

Number random() method

The **random()** method provides a random floating point number in the range **[0.0, 1.0]** as output.

Syntax:

random ()

The **random()** method is not accessible directly. In order to access it, we must import the random module and then use the random static object to call this function.

Parameters

start – indicates the starting position of the range. The range includes the start; the default value is zero.

stop – indicates the stop position of the range. The given range doesn't include a stop.

step – indicates the value with which the number is incremented. The default value is one.

Result

This method provides a random item from the given range as the output.

Example:

```
>>> import random

>>> print :

: 0.9279919237961513

>>> print :

: 0.9800448891204537
```

Number seed() method

The **seed()** method loads the basic random number generator. You must call this function before calling any other random module function.

Syntax:

seed ([x], [y])

Note: The seed() function initializes the basic random number generator.

Parameters

The next random number will be generated by this seed. When omitted, the next random number has to be generated by the system; X is used directly if it is an integer.

The version number defaults to 2. A string, byte, or byte array object will be converted to an integer if this value is set. Version 1 uses **hash()** of x.

Result

This method does not return any value.

Example:

```
import random iop



print number with default seed



print number with int
```

{when seeded with a string “hello”, the
hash() of the string is used as the seed value during
randomization.}

```
print numbers with string
```

Output

```
random number with default seed 0.2524977842762465
random number with int seed 0.5714025946899135
random number with string seed 0.3537754404730722
```

Number shuffle() method

The **shuffle()** function randomizes the items of a list in place.

Syntax:

shuffle (lst,[random])

Note: The shuffle() method is only accessible using the shuffle static module object.

Parameters

lst - indicates a list or a tuple.

random - indicates an optional 0 argument function returning float between 0.0 - 1.0; the default value is none.

Result

This method returns the reshuffled list.

Example:

```
import random
```

```
number_list = [7,14,21, 28, 35, 42, 49, 56, 63, 70]

random.shuffle(number_list)
```

Output

```
14, 21, 70, 28, 49, 35, 56, 7, 42, 63
```

Number uniform() method

The **uniform()** method provides as output a random float r, such that **m** <= **r** and **r** <

Syntax:

uniform(m, n)

Note: The uniform() method is only accessible using the uniform static module object.

Parameters

x - fixes the lower limit of the random float

y - fixes the upper limit of the random float

Result

This method provides as output a floating-point number r, such that **x** <=**r** <

Example:

```
import random

>>> print Float uniform(5, 10) :

Random Float : 8.3159929051343O8
```

Trigonometric functions

Python includes the following functions that perform trigonometric calculations:

calculations:
calculations: calculations: calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations:
calculations: calculations: calculations: calculations: calculations: calculations: calculations:

calculations: calculations: calculations: calculations: calculations: calculations: calculations:

Table 7.1: *Trigonometric functions*

Number acos() method

The **acos()** method provides the arc cosine of x (in radians) as output.

Syntax:

acos(x)

Note: The acos() method is only accessible using the acos static module method.

Parameters

x - Indicates a numeric value that comes in the range -1 to 1. If x > 1, it will develop 'math domain error'.

Example:

```
import math

print :

print :
```

Output

```
acos(0.46)  :  0.8762980611683406
acos(0)  :  1.5707963267948966
```

Number asin() method

The **asin()** method provides the arc sine of **x** (in radians) as output.

Syntax:

asin(x)

Note: The asin() method is only accessible using the asin static module object.rameters:

x – indicates a numeric value ranges from -1 to 1. If x > 1, it will produce 'math domain error'.

Example:

```
import math

print :

print :
```

Output

```
: 0.694498265626556

: -1.5707963267948966
```

Number atan() method

The **atan()** method provides the arc tangent of x (in radians) as output.

Syntax:

atan(x)

Note: The atan() method is only accessible using the atan static module object.

Parameters

x – indicates a numeric value that ranges from -1 to 1. If x > 1, it will produce 'math domain error'.

Example:

```
import math

>>> print :

: 0.0
```

Number atan2() method

The **atan2()** method provides **atan(y / x)** (in radians) as output.

Syntax:

atan2(y, x)

Note: The atan2() method is only accessible using the atan2 static module object.

Parameters

y – indicates a value that must be a numeric value.

x – indicates a value that must be a numeric value.

Example:

```
import math

print :
```

Output

```
: -2.356194490192345
```

Number cos() method

The **cos()** method provides the cosine of x (in radians) as output.

Syntax:

cos(m)

Note: The cos() method is only accessible using the cos static module object.

Parameters

m - This indicates a value that must be numeric.

Return value

This method provides a numeric value that ranges from -1 and 1, and it represents the cosine of the angle.

Example:

```
>>> import math

>>> print :
```

: -0.9899924966004454

Number hypot() method

The **hypot()** method provides the Euclidean norm, **sqrt(x*x +** as output. This indicates the length of the vector from the origin to point

Syntax:

hypot(x, y)

Note: The hypot() method is only accessible using the static hypot module object.

Parameters

x – This indicates a value that must be numeric.

y - This indicates a value that must be numeric.

Example:

math

2) :

```
: 3.605551275463989
```

Number degrees() method

The **degrees()** method transforms angle x from radians to degrees.

Syntax:
degrees(x)

Note: The degrees() method is only accessible using the degrees static module object.

Parameters:

x – This indicates a value that must be numeric.

Return value

This method provides an output in the degree value of an angle.

Example:

math

:

: 171.88733853924697

:

: -171.88733853924697

Number radians() method

The **radians()** method transforms angle x from degrees to radians.

Syntax:

radians(x)

Note: The radians() method is only accessible using the static radians module object.

Parameters

x - This indicates a value that must be numeric.

Return value

This method returns the radian value of an angle.

Example:

math

:

: 0.05235987755982989

:

: -0.05235987755982989

Mathematical constants

Constants, which are constant numbers whose value is unambiguously defined, appear in many areas of mathematics, with constants like e and * appearing in such diverse contexts as geometry, number theory, and calculus.

The module also defines two mathematical constants, as follows:

follows:
follows: follows: follows: follows:
follows: follows: follows: follows:

Table 7.2: *Mathematical constants*

Conclusion

A number in Python refers to a numeric data type. Numeric types in Python are int, float, and complex. In this chapter, we learned how to use numbers and understood their significance in Python programming.

In the next chapter, we will discuss strings.

Key terms

Floating points

Complex numbers

Random(), hypot(), degrees(), radians()

seed(), acos(), asin(), atan()

Constant

CHAPTER 8

Strings

Introduction

When characters like letters, digits, and symbols are arranged in a sequence, it is known as a string. The usual string object is an immutable (unchangeable) sequence of characters accessed by offset (position).

Strings are immutable, but you can update the value of a string by assigning a new string to the same string variable.

Structure

We will cover the following topics in this chapter:

Creating a string

Accessing values in string

Changing or deleting a string

String operations

Build-in string methods

Programs

Objectives

Strings can be worked on within Python using a variety of operators, functions, and methods. After going through the chapter, you will learn how to access and extract strings, and you will also be able to manipulate and modify string data using the methods available. Additionally, you will learn about the built-in string functions in Python.

Creating a string

We can create strings by enclosing characters within a single quote or double quotes. Python supports triple quotes, but these are generally used to represent multiline strings and docstrings.

Single quoted and double-quoted strings are identical.

In Python strings, single- and double-quote characters are interchangeable, i.e., string literals can be expressed in two single or two double quotes. Here, the two representations work in the same way and return the same type of object.

The reason for supporting the two representations is that it allows programmers to enclose a quote character of the other type within a string without a backslash. It also helps programmers to enclose a single-quote character in a string enclosed in double-quote characters, and vice-versa.

Example:

said

Output

```
hello
hello
doesn’t
He said “Hello”
```

Accessing values in a string

Python does not allow a character type; these are considered as strings of length one, so they are also considered substrings.

We can access individual characters using indexing and with different characters using the process of slicing. The index always starts from zero, but an IndexError may arise while trying to access a character out of the index range. The index is always an integer; we cannot use float or other data types; this will lead to TypeError.

Python allows negative indexing for its sequences. For example, the index of -1 indicates the last item, -2 the second last item, and so on. Python also allows you to access a range of items in a string using the slicing operator (colon).

Slice expressions support the optional third index, which is used as a step (sometimes called stride). Steps are added to the index of each item extracted. Slices are now X[I:J:K], which means that all items in X are removed from offsets I through J*1, by K. K is generally set to +1, which is why all items in a slice are generally extracted from left to right. However, the third limit can be used to skip items or reverse their order if you specify an explicit value. X[1:10:2] will fetch every other item in X from offsets 1–9. It will collect items at offsets 1, 3, 5, 7, and 9. Since the first and second limits are set to 0 and the length of the sequence, respectively, X[::2] receives all items from the beginning to the end of the sequence.

Example:

world"

Reversing items

Skipping items

Output

```
'h'
'llo world'
'hello world'
'hello '
'lo wo'
'wo'
'olleh'
'dlrowolleh'
```

```
'hlowr'
```

Changing or deleting a string

Strings are immutable, which implies that elements of a string once assigned cannot be changed. Python allows you to set the same name to different strings.

```
Traceback (most recent call last):

  File line in

object does not support item assignment

a

'world'
```

We cannot delete or remove characters from a string, but we can delete the string entirely using the **del** keyword.

```
Traceback (most recent call last):
  File line in

del

TypeError: 'str' object support item deletion

>>> del a

>>> a

Traceback (most recent call last):

  File "", line 1, in

    a

NameError: name is not defined
```

Escape characters

Escape sequences or non-printable characters are characters that are not printed on the screen as is. They convey a different meaning, and backslash notation is used to represent them.

An escape character is interpreted with single quoted as well as double quoted strings.

strings.
strings. strings. strings.
strings.
strings.
strings.
strings.
strings.
strings. strings. strings. strings. strings. strings. strings. strings. strings. strings.
strings. strings. strings. strings.
strings. strings. strings.
strings. strings. strings.
strings. strings. strings. strings.

strings. strings. strings. strings.
strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings. strings.

Table 8.1: *Backslash notation*

String_operations

Concatenation

You can concatenate strings using the + operator. Formally, adding two string objects with the + symbol creates a new string object with the contents of its operands joined.

```
c=a+b

c

'helloworld'
```

Repetition

You can repeat a string using the * operator. Repetition with * is related to the addition of a string to itself a number of times.

"

b

'hello hellohellohello

Membership

in - Returns true if a character exists in the given string.

not in - Returns true if a character does not exist in the given string.

world"

b in a

True

c in a

False

b not in a

False

Raw string

Raw strings consider escape characters to be part of the complete string. The syntax for raw strings and normal strings are the same, except that the letter "r" precedes the quotation marks in the raw string operator. The raw string operator is case insensitive, which means, "r" can either be lowercase (r) or uppercase (R) and should be included immediately preceding the first quote mark.

Example:

hello

world

r

hello\nworld

String formatting

Python also supports a leading method to blend various string processing tasks; string formatting permits you to do multiple type-specific substitutions on a string in one step. It’s not strictly needed, but it can be convenient, especially when formatting text to be displayed to a program’s users.

The initial expression (all Python versions), coded with the % operator: fmt % (values)

The latest method (3.0, 2.6, and later), coded with **call** fmt.format(values)

Example:

```
'%s, %s,

'23, hello, 2.130000'



'23,hello,2.130000'
```

Triple quote code

Python also supports triple-quoted string literal format, also known as a block string. It is a syntactic accessibility for performing various multiline coding with different text data. This format begins with three quotes (of either the single or double variety), is followed by one or more lines of text, and ends with the same type of three quotes that opened it. It's not necessary to embed single and double quotes in the string's text, escaped— the Python does not end the string until it sees three unescaped quotes of the same type as the ones used to start it.

Example:

```
how're you!!!

I am

hello

you!!!

I am good
```

Unicode string

All text is unicode text, including the text encoded with one character per byte (8 bits) in the ASCII scheme. Python supports richer character sets and encoding schemes with unicode, that is, strings that may use multiple bytes to represent characters in memory, and which translate text to and from various encodings on files.

Built-in string methods

String capitalize() method

This method provides us a copy of the string with the first letter capitalized and the rest lowercase as the output.

Syntax:

```
str.capitalize()
```

Parameters

NA

Return value

string

Example:

```
a=”hello world”
```

```
'Hello world'
```

String center() method

The **center()** method returns the string at the center. Padding is done using the specified fillchar, and the default filler is a space.

Syntax:

str.center(width[, fillchar])

Parameters

width - This returns the total width of the given string

fillchar - This acts as the filler character to fill the gap

Example:

```
world"
```

```
'*********hello  world**********'
```

String count() method

The **count()** method gives the number of occurrences of the substring sub within the range **[start, end]** as the output. Arguments start and end are optional, and they are interpreted the same as in slice notation.

Syntax:

```
str.count(sub, start= 0,end=len(string))
```

Parameters

sub - This is the substring needed for searching

start - Search always starts from this index, and the first character usually starts from the index, so search starts from the index by default.

end - Search ends at this index; the first character always starts from index, and search ends at the last index in the default case

Return value

Centered in a string with length width

Example:

```
world  hello"
```

```
2
```

String decode() method

The **decode()** method is used to decode the string using the codec registered for encoding. It defaults to the default string encoding.

Syntax:

```
str.decode(encoding='UTF-8',errors='strict')
```

Parameters

encoding - This is used for encoding.

errors – It provides different error handling schemes. ‘Strict’ is the default value for errors, which indicates a Other available values for errors are **backslashreplace** and any other name registered via

Return value

It returns a decoded string.

Example:

String = ‘Python’

```
Print('The String)

String_utf=String.encode()

Print('The encoded version

Print('The decode version
```

Output

```
The String is: Python
The encoded version is b'pyth\xc3\xb6n!
The decoded version is Python
```

String encode() method

The purpose of the **encode()** method is to provide the encoded version of the string. The default encoding is the current default string encoding. The errors produced are given to set a different error handling scheme.

Syntax:

```
str.encode(encoding='UTF-8',errors='strict')
```

Parameters:

encoding - This is used for encodings.

errors - It provides different error handling scheme. **Strict** is the default value for errors, which indicates a UnicodeError, and other available values for errors are **backslashreplace** and any other name registered via

Return value

It returns an encoded string

Example:

```
String = 'Python'

Print('The string is:',



Print('The encoded version is:',String_utf)
```

Output

```
The String is: Python
The encoded version is b'pyth\xc3\xb6n!
```

String endswith() method

The boolean value true is returned when the string ends with the given suffix; otherwise, it returns false. Also, it restricts the matching with the given indices start and end, which are optional.

Syntax:

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

Parameters

suffix – It can be a string or a tuple of suffixes to look for

start – It represents the start of the slice

end – It indicates the end of the slice

Example:

```
is a test of endswithfunction....!!'



print (Str.endswith(suffix))

print (Str.endswith(suffix,25))
print (Str.endswith(suffix))

print (Str.endswith(suffix, 0, 24))
```

Output

```
True
True
False
False
```

String expandtabs() method

This method provides an exact copy of the string in which the tab character, that is, uses spaces to expand, using the given tabsize (default

Syntax:
str.expandtabs(tabsize=8)

Parameters

This indicates the number of characters needed to replace for the given tab character **\t**

Return value

An expanded string is returned in which tab character expansions have been performed using spaces, for example, **/t**

Example:

```
str = "this is expandtabsfunction....!!"

print string: " +

print tab: " +
print expanded tab: " +
```

Output

```
Original String: this is expandtabsfunction....!!
Defaultexpanded tab: this is expandtabsfunction....!!
Double expanded tab: this is expandtabsfunction....!!
```

String find() method

The **find()** method determines whether the given string **str** occurs in the string or in a substring of the string if the starting index **beg** and ending index **end** are given.

Syntax:

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

Parameters

str - It is the string needed to be found

beg - It is the starting index of the string; by default, it is zero

end - It indicates the ending index; by default, it is equal to the length of the string.

Return type

Index if found and -1 otherwise

Example

```
str1 = "this is string find function..!!"

str2 =

print

print

print
```

Output

```
15
15
-1
```

String index() method

The **index()** method determines whether the string str occurs in string or in a substring of the string, given the starting index beg and ending index end. This method is similar to but it raises an exception if sub cannot be found.

Syntax:

str.index(str, beg=0 end=len(string))

Parameters

Str: It specifies the string to be searched

It is the starting index; by default, it is 0

The ending index, by default, is equal to the string length

Return value

Index if and raises an exception if str is not found

Example:

```
str1 = "this is string index function...!!"

str2 =

print (str1.index(str2))

print (str1.index(str2, 10))
```

Output

```
15
15
```

String isalnum() method

The **isalnum()** method checks whether the string contains alphanumeric characters.

Syntax:

```
str.isalnum()
```

Parameters

NA

Return type

This method returns true if all characters in the string are alphanumeric and there is at least one character; otherwise, it returns false.

Example:

```
str = "thisisnot1234"

print
```

```
str = "this is string alphanumfunction..!!"

print
```

Output

```
True
False
```

String isalpha() method

The **isalpha()** method checks whether the string consists of alphabetic characters only.

Syntax:
str.isalpha()

Parameters

NA

Return type

The method returns true if there is at least one character in the string and all characters are alphabetic; otherwise, it returns false.

Example:

```
str =
```

```
print
```

```
str = "this is string isalphafunction..!!"
```

print

Output

```
True
False
```

String islower() method

You can check whether all case-based characters (letters) in a string are lowercase by calling the **islower()** method.

Syntax:

```
str.islower()
```

Parameters

NA

Return type

The method returns true if all cased characters in the string are lowercase

Example:

```
str = "THIS is string islowerfunction..!!"

print

str = "this is string islowerfunction..!!"
```

print

Output

False

String isnumeric() method

The **isnumeric()** method checks whether a string contains only numeric characters. It is only available on unicode objects.

Syntax:

str.isnumeric()

Parameters

NA

Return type

This method returns true if all characters are numeric; otherwise, it returns false.

Example:

```
str = "this2016"

print

str = "23443434"
```

```
print
```

Output

```
False
True
```

String isspace() method

The **isspace()** method checks whether the string consists of whitespace.

Syntax:

str.str.isspace()

Parameters

NA

Return type

This method returns true if there is at least one character in the string and there are only whitespace characters; otherwise, it returns false.

Example:

```
str = "  "

print
```

```
str = "This is string isspace function!!"

print
```

Output

```
True
```

String istitle() method

The **istitle()** method checks whether all case-based characters in a string following non-case-based letters are uppercase and whether all other case-based characters are lowercase.

Syntax:

```
str.istitle()
```

Parameters

NA

Return type

As an example, uppercase characters can only follow uncased characters, and lowercase characters can only follow cased characters. So, this method returns true if the string is a title cased string and there is at least one character; otherwise, it returns false.

Example:

```
str = "This Is String Function..!!"
print (str.istitle())
```

```
str = "This is string function..!!"
print (str.istitle())
```

Output

```
True
```

String isupper() method

The **isupper()** method determines whether all the case-based characters (letters) in the string are uppercase.

Syntax:

str.isupper()

Parameters

NA

Return type

The method returns true only if all characters in the string are uppercase

Example:

```
str = "THIS IS STRING ISUPPER FUNCTION..!!"

print

str = "THIS is string upper function..!!"
```

print

Output

True

String join() method

The **join()** method returns a string containing all the string elements of a sequence separated by an str separator.

Syntax:

str.join(sequence)

Parameters

sequence - In this case, the elements will be joined in a sequence

Return type

The method returns a string that is the concatenation of the strings in the sequence seq. The string providing this method acts as a separator between elements.

Example:

```
s = ">"

seq = # This is sequence of strings.

print (s.join(seq))
```

Output

```
3>2>1
```

String len() method

This **len()** method returns the length of the string.

Syntax:

```
len(str)
```

Parameters

NA

Example:

```
str = "This Is String lenFunction..!!"

print of the string:
```

Output

```
Length of the String: 30
```

String ljust() method

The **ljust()** method returns the left-justified string in a string of length width. The padding is calculated using the specified fillchar (by default, a space is used). The original string is returned if width is smaller than len (s).

Syntax:

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

Parameters

width - After padding, this is the total length of the string

fillchar - This is a filler character; the default is a space

Example:

```
str = "This Is String ljustFunction..!!"

print
```

Output

```
This  Is  String  ljustFunction..!!******************
```

String lower() method

The **lower()** method returns a copy of the string if all case-based characters have been lowercased.

Syntax:

str.lower()

Parameters

NA

Example:

```
str = "This Is String lower Function..!!"

print
```

Output

```
this is string lowerfunction..!!
```

String lstrip() method

The **lstrip()** method returns a copy of the string that has been stripped of all characters at the beginning (default whitespace characters).

Syntax:

str.lstrip([chars])

Parameters

chars - You can specify which characters need to be trimmed.s

Example:

```
str = "This Is String lstripFunction..!!"

print

str = "*****this is string lstripfunction..!!*****"

print
```

Output

```
This Is String lstripFunction..!!
this is string lstripfunction..!!*****
```

String maketrans() method

Using each character in the intabstring is mapped into the character in the outtabstring at the same position. **Translate()** then takes this table as input.

Syntax:

```
str.maketrans(intab, outtab]);
```

Parameters

intab - The actual characters are in this string

outtab - There are corresponding mapping characters in this string

Return type

The method returns a translate table to be used with the **translate()** function

Example:

Intab=”aeiou”

```
outtab = "12345"

trantab = str.maketrans(intab, outtab)

str = "this is string translate function...!!"

print (str.translate(trantab))
```

Output

```
Th3s 3s str3ng tr1nsl2t3 f5ct34n...!!
```

String max() method

The **max()** method returns the maximized alphabetical character from the string

Syntax:

```
max(str)
```

Parameters

str - From this string, the maximum alphabetical character should be returned

Example:

```
str = "this is a string example..!!"

print character: " +

str = "this is a string max function..!!"

print character: " +
```

Output

Max character: x

String min() method

The **min()** method returns the minimum alphabetical character in the string str.

Syntax:

min(str)

Parameters

str - The string from which the minimum alphabetical character needs to be returned.

Return type

This method returns the minimum alphabetical character from the **str** string.

Example:

```
str = "thisisastringexample..!!"

print character: " +
str = "thisisastringmaxfunction..!!"

print character: " +
```

Output

```
Min character: !
```

String replace() method

A string returned by **replace()** has all occurrences of the old replaced with the new, optionally, with a maximum number of replacements.

Syntax:

str.replace(old, new[, max])

Parameters

old – This is the old substring to be replaced

new - This is a new substring that replaces the old substrings

max - When the optional argument max is given, only the first count occurrences are replaced

Return type

This method returns a copy of the string with all instances of substring old replaced by the new one. By specifying the optional argument max, only the first count occurrences are replaced.

Example:

```
str = "this is a replace function"

print
```

Output

```
this was a replace function
```

String rfind() method

The **rfind()** method returns the last index in which the **str** substring is found or **-1** if no such index exists; optionally, the search is restricted to **String**

Syntax:

str.rfind(str, beg=0 end=len(string))

Parameters

str – It specifies the string to be searched

beg - It is the starting index; by default, it is 0

end - By default, the ending index is equal to the length of the string

Example:

```
str1 = "this is a string rfindfunction..!!"

str2 = "is"
```

```
print  (str1.rfind(str2))

print  (str1.rfind(str2,

print  (str1.rfind(str2,

print

print

print
```

Output

```
5
5
-1
2
2
-1
```

String rindex() method

When **rindex()** is called, it returns the last index where the str substring has been found or raises an exception if no index exists; optionally the search is restricted to

Syntax:

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

Parameters

str - It specifies the string to be searched

beg – The starting index; by default, it is 0

len – The ending index; by default, it is equal to the length of the string

Example:

```
str1 = "this is a string rindex function...!!"

str2 = "is"
```

```
print (str1.rindex(str2))

print (str1.rindex(str2,10))
```

Output

```
5
Traceback (most recent call last):
File "C:/Users/Admin/Desktop/python files/srindex.py", line 4, in
print (str1.rindex(str2,10))
ValueError: substring not found
```

String rjust() method

A string of length width is returned by the **rjust()** method with the string right justified. The padding is calculated using the specified fillchar (by default, a space is used). The actual string is returned if the width is less than len (s).

Syntax:

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

Parameters

width - After padding, this is the length of the string in total

fillchar - It is the filler character; by default, it is a space

Example:

```
str = "this is string rjustfunction..!!"

print
```

Output

```
*****************this  is  string  rjustfunction..!!
```

String rstrip() method

Whitespace characters are stripped from the string using rstrip() (default).

Syntax:

```
str.rstrip([chars])
```

Parameters

chars - You can give what chars have to be trimmed

Example:

```
str = " this is a string rstripfunction....!! "

print
```

Output

```
this is a string rstripfunction....!!
```

String split() method

The **split()** function splits a string into words using str as a separator (breaks on whitespace if not specified), there is an option limiting the number of splits to num.

Syntax:

```
str.split(str="", num=string.count(str))
```

Parameters

str - It is any delimiter; by default, this is space

This is the number of lines to be made

Return type

This method returns a list of lines

Example:

```
str = "this is a string split function..!!"

print
```

```
print

print
```

Output

```
['this', 'is', 'a', 'string', 'split', 'function..!!']
['th', 's is a string split function..!!']
['this is a string sp', 'it function..!!']
```

String splitlines() method

The **splitlines()** method returns the lines of string, optionally including the line breaks (if the num parameter is true).

Syntax:

```
str.splitlines(num=string.count('\n'))
```

Parameters

num - If present, it would be assumed that the line breaks need to be included in the text

Return type

This method returns true if found matching with the string; otherwise, it returns false

Example:

```
str = "this is \nstringsplitlines \nfunction..!!"
```

Output

```
['this is ', 'string splitlines ', 'function..!!']
```

String startswith() method

The **startswith()** method checks whether the string starts with str, optionally restricting the matching with the given indices start and end.

Syntax:

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

Parameters

str - String to be checked

beg - This is the optional parameter to set start index to matching boundaries

end - This is the optional parameter to set end index of the matching boundary

Return type

This method returns true if found matching with the string; otherwise, it returns false.

Example:

```
str = "this is string startswithfunction....!!"

print

print

print
```

Output

```
True
True
False
```

String strip() method

The **Strip()** method returns a copy of a string in which all whitespace characters have been stripped (default).

Syntax:

```
str.strip([chars])
```

Parameters

chars - Characters will be trimmed at the start or the end with the selected characters.

Example:

```
str = "*****this is a string strip function....!!*****"

print
```

Output

```
this is a string strip function....!!
```

String swapcase() method

Swapping the case of the characters in a string is accomplished by the **swapcase()** method.

Syntax:

str.swapcase()

Parameters

NA

Return type

If all the case-based characters are swapped, then this method returns a copy of the string

Example:

```
str = "this is string swapcasefunction....!!"

print

str = "This Is String Swapcase Function....!!"
```

print

Output

```
THIS IS STRING SWAPCASE FUNCTION....!!
```

String title() method

The **title()** method returns a copy of the string in which the starting characters of all the words are capitalized.

Syntax:

```
str.title()
```

Parameters

NA

Example:

```
str = "this is string title function....!!"

print
```

Output

```
This Is String Title Function....!!
```

String translate() method

A copy of the string is returned by the **translate()** method, in which the characters have been translated using table (created by **maketrans()** in the string module), optionally removing any characters found in the string

Syntax:

str.translate(table[, deletechars]);

Parameters

table can use the **maketrans()** function to create translate table

deletechars - Several characters in the source string need to be removed

Return type

This method returns a copy of the translated string

Example:

from string import maketrans #
Required to call maketrans function.

```
input = "aeiou"

output = "12345"

traslt = maketrans(input, output)

str = "this is string

print (str.translate(traslt))
```

Output

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

String upper() method

The **upper()** method returns a copy of the string in which all case-based characters have been uppercased.

Syntax:

```
str.upper()
```

Parameters

NA

Return type

Returns the string with all characters converted to uppercase

Example:

```
str = "this is string upper function....!!"

print :
```

Output

```
str.upper : THIS IS STRING UPPER FUNCTION....!!
```

String zfill() method

The **zfill()** method pads string on the left, with zeros to fill the width.

Syntax:

str.zfill(width)

Parameters

width - This is the final width of the string; it is the width we would get after filling in zeros.

Return type

This method returns padded string

Example:

```
str = "this is string zfill function...!!"

print :
```

Output

```
str.zfill : oooooothis is string zfill function...!!
```

String isdecimal() method

You can use the **isdecimal()** method to determine whether a string contains only decimal characters; only unicode objects have this method.

Note: Unlike in Python 2, all strings are represented as Unicode in Python 3. The following example illustrates it.

Syntax:

str.isdecimal()

Parameters

NA

Return type

This method returns true if all the characters in the strings are decimal; otherwise, it returns false.

Example:

```
str = "thisisnot01234"
```

```
print
```

```
str = "2334567434"
```

```
print
```

Output

```
False
True
```

Programs

Program to find the index of a character in a string:

```
a

a


for i in range(len(a)):




found at


not
```

Output

```
Enter a string -hello
Enter a character- e
character found at index 1
```

Program that counts the number of characters in a string:

```
a string: ')s


in str:

in str:

is i:


print(i,count)
```

Output

```
Enter a string: python
p 1


y 1
t 1
h 1
o 1
n 1
```

Program to reverse a string:

a string: ")

in a:

Output

```
Enter a string: hello world
dlrowolleh
```

Program to find a substring in the string:

a string:

substring:

and

Output

```
enter a string: hello how are you
enter substring: are

found
```

Program to count vowels in a string:

```
a string:



for i in str:

or (i or
(i or (i or (i
```

Output

```
Enter a string: hello world
3
```

Program to check whether a string is palindrome:

```
a string:

for i in a:
    b=i+b
```

Output

```
enter a string: Malayalam
palindrome
```

Program to find the occurrence of a word in a string:

```
a string:

s2=[]
```

```
count=0


for i in range(0,len(s2)):

    if(word==s2[i]):

        count+=1

of word is
```

Output

```
enter a string: abcfghhjabcabcfwabcrabchhabc
enter word: abc
count of word is  3
```

Program to sort a string:

```
a

b=a.split()
```

Output

```
Enter a string- hello how are you
['are', 'hello', 'how', 'you']
```

Program to check whether the two strings are anagram:

```
# check whether strings are anagram or not

first string:

second string:

if(sorted(s1)==sorted(s2)):

strings are

else:

are not
```

Output

```
enter first String: hello
enter second String: eholl
both strings are anagram
```

Program to calculate the number of digits and characters in a string:

```
a string:

countN=0

countC=0

for i in s1:

      if(i.isdigit()):

countN+=1

countC+=1

=

=
```

Output

```
enter a string: hello 123 hows you 678
digits = 6
characters = 22
```

Conclusion

String manipulation is easy with Python's built-in functions. Strings in Python are immutable, so these functions return a new string and leave the original string unchanged. You can do a lot with strings in Python, but it's not possible to remember them all. In this chapter, we discussed the different operations of string and the built-in functions of string.

In the next chapter, we will take a look at Python

Questions

If a string contains words, how can you check if they begin with a capital letter?

Which method can be used to verify whether two strings are the same?

Check whether a string contains any substring.

Count the characters in a string.

Find a string that contains numbers and any uppercases.

CHAPTER 9

Lists

Introduction

The primary data structure of Python is the sequence. Every element of a sequence is assigned a number—its position or index. The first index will be zero, the second index will be one, and so on.

There are six types of sequences in Python, but lists and tuples are the most common among these.

Structure

We will cover the following topics in this chapter:

Lists in Python

Accessing values in the list

Updating list

Deleting list elements

Built-in list functions and methods

Objectives

Python lists have a bunch of methods to performdifferent tasks. For example, we can sort records with eliminate the last element from a list with count the quantity of things we're searching for with and remove elements with After going through the chapter, readers will learn about the key operations on a Python list.

Lists in Python

Lists are heterogenous data structures created by elements separated with commas and enclosed within square brackets.

Example:

List1 =

List2 =

List3 =

Accessing values in a list

The **[]** index operator can be used on a list to access elements by the index.

Example:

```
list1[0])

print list2[1:5])
```

Output

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

Updating lists

The variable on the left-hand side of the assignment operator can be used to update or add to elements in lists. The **append()** method can be used to add to the elements in a list.

Example:

```
list1 =

print available at index 2:

= 2000

print value available at index 2:
```

Output

```
Value available at index 2: 1994
New value available at index 2: 2000
```

Deleting list elements

When deleting a list element, you can use either of the del statements to know exactly which element(s) you are deleting. You can use the **remove()** method if you don't know exactly which items to delete. Rundown can be replaced with the collection as the development community is familiar with this term rather than the name rundown.

Example:

```
list =

print

del list

print deleting value at index 2:
```

Output

```
['maths', 'chemistry', 1994, 1997]
After deleting value at index 2: ['maths', 'chemistry', 2000]
```

Built-in list functions and methods

List len() method

The **len()** method returns the total number of components in the rundown.

Syntax:

len(list)

Parameters

list - This is the list in which the elements are to be counted.

Return value

This returns the number of elements in the list.

Example:

list1 =

print

#creates list of
numbers between 0-4

```
print
```

Output

```
3
5
```

List max() method

This **max()** method returns the components from the rundown with the most noteworthy worth.

Syntax:
max(list)

Parameters

list - The number of elements in this list is to be counted.

Return value

The max() method returns the components in the rundown with the most elevated worth.

Example:

```
list1, list2 =

print value element:

print value element:
```

Output

```
Maximum value element : Python
```

List min() method

The **min()** method returns the components from the rundown with the least worth.

Syntax:

```
min(list)
```

Parameters

list - This is the list in which the number of elements is to be counted.

Return value

This **min()** method returns the elements from the list with minimum value.

Example:

```
list1, list2 =

print value element :
```

```
print value element :
```

Output

```
min value element : C++
min value element : 100
```

List list() method

The **list()** method converts sequence types into lists. For example, it is used to convert a tuple into a list.

Note: Tuples are similar to records, the difference being that component upsides of a tuple can't be changed, and tuple components are put between enclosures rather than square sections. Similarly, this method changes characters in a string into a rundown.

Syntax:

list(seq)

Parameters

seq- This is a tuple or string to be converted into a List.

Return value

This method returns the list.

Example:

Tuple1 =

```
list1 =

print elements : list1)




print elements : list2)
```

Output

```
List elements : [897, 'C', 'Java', 'Python']
List elements : ['W', 'e', 'l', 'c', 'o ', 'm', 'e']
```

Python includes the list methods discussed here.

List append() method

The append() method appends a passed obj to the existing list.

Syntax:

list.append(obj)

Parameters

The object to be appended to the list.

Return value

There is no return value, but the updated list is printed.

Example:

```
list1 =


print list : list1)
```

Output

```
updated list : ['C++', 'Java', 'Python', 'C#']
```

List count() method

The **count()** method returns the count of how frequently **obj** occurs in the list.

Syntax:

list.count(obj)

Parameters

obj - Object to be counted in the list.

Return value

The **count()** method returns the count of how many times **obj** occurs in the list.

Example:

```
aList = [123, 123];

print for 123 : aList.count(123))

print for zara :
```

Output

```
Count for 123 : 2
Count for zara : 1
```

List extend() method

The **extend()** method appends the contents of seq to list:

Syntax:

list.extend(sequence)

sequence is the list's element.

Return value

The **extend()** method does not return any value but adds the content to an existing list.

Example:

list1 =

#creates list of numbers between 0-4

List list2)

print (list1)

Output

```
Extended List : ['physics', 'chemistry', 'maths', 0, 1, 2, 3, 4]
```

List index() method

The index() method returns the lowest index in list that obj appears.

Syntax:

list.index(obj)

Parameters

obj – obj is the object to be determined.

Return value

This method returns the index of the found object or raises an exception indicating that the value was not found.

Example:

```
list1 =

print of

print of
```

Output

```
Index of chemistry 1
Traceback (most recent call last):
File "test.py", line 3, in
print ('Index of C#', list1.index('C#')) ValueError: 'C#' is not in list
```

List insert() method

The **insert()** method inserts object into list at offset index.

Syntax:

```
list.insert(index, obj)
```

Parameters

index - This is the index; the object to be inserted

obj - This is the object to be inserted into the list

Return value

It does not return a value; rather, it inserts the given element at the given index.

Example:

```
list1 =



print list : list1)
```

Output

```
Final list : ['physics', 'Biology', 'chemistry', 'maths']
```

List pop() method

The **pop()** method eliminates the last item and returns the last item from the list.

Syntax:

list.pop(obj=list[-1])

Parameters

obj – This is the index of the object to be removed from the list, and it is an optional parameter.

Return value

This method returns the removed object from the list.

Example:

list1 =

list1.pop()

print now : list1)

```
list1.pop(1)

print now : list1)
```

Output

```
list now : ['physics', 'Biology', 'chemistry']
list now : ['physics', 'chemistry']
```

List remove() method

The **remove()** function is an in-built function in the list collection that removes a given object from the list.

Parameters

obj - This is the object to be removed from the list.

Return value

The **remove()** method does not return any value but removes the given object from the list.

Example:

```
list1 =



print now : list1)



print now : list1)
```

Output

```
list now : ['physics', 'chemistry', 'maths']
list now : ['physics', 'chemistry']
```

List reverse() method

The reverse() method reverses the objects of the list in place.

Syntax:

```
list.reverse()
```

Parameters

NA

Return value

The **reverse()** method does not return any value; rather, it reverses the given object from the list.

Example:

```
list1 =


print now : list1)
```

Output

```
list now : ['maths', 'chemistry', 'Biology', 'physics']
```

List sort() method

The Python list **sort()** capacity can be utilized to sort a list in climbing, plummeting, or client characterized request.

Syntax:

```
list.sort([func])
```

Parameters

NA

Example:

```
list1 =


print now : list1)
```

Output

```
list now : ['Biology', 'chemistry', 'maths', 'physics']
```

Programs

Program to find the largest number in a list:

```
a=[]

number of elements:

for i in

element:



a.sort()

element is:
```

This is the output:

```
Enter number of elements: 4
Enter element: 12
Enter element: 78
Enter element: 23
Enter element: 45
Largest element is: 78
```

Program to categorize odd and even elements in a list into two different lists:

```
a=[]

number of

for i in

elements:


even=[]

odd=[]

for j in a:







even

odd
```

The output is as follows:

```
Enter number of elements:5
Enter elements: 34
Enter elements: 67
Enter elements: 12
Enter elements: 98
Enter elements: 55
The even list [34, 12, 98]
The odd list [67, 55]
```

Program to merge two lists and sort it:

```
a=[]

c=[]

number of

for i in




number of elements:

for i in
```

element:

list is:

Here's the output:

```
Enter number of elements:5
Enter element34

Enter element22
Enter element67
Enter element45
Enter element11
Enter number of elements: 4
Enter element: 23
Enter element: 66
Enter element: 77
Enter element: 99
Sorted list is: [11, 22, 23, 34, 45, 66, 67, 77, 99]
```

Program to sort a list according to the length of the elements:

```
a=[]

number of elements:
```

```
for i in

element:
```

The output is as follows:

```
Enter element: 1
Enter element: 234
Enter element: 65
Enter element: 8834
['1', '65', '234', '8834']
```

Program to swap the first and last values in a list:

```
a=[]

the number of elements in list:

for x in
```

```
list is:
```

Here’s the output:

```
Enter the number of elements in list: 4
Enter element1:12
Enter element2:67
Enter element3:33
Enter element4:45
New list is:
[45, 67, 33, 12]
```

Program to remove duplicate items from a list:

```
a=[]

the number of elements in list:

for x in




b=set()

unique=[]
```

```
for x in a:

if x not in b :



b.add(x)

items:
```

It gives the following output:

```
Enter the number of elements in list: 7
Enter element1:23
Enter element2:56
Enter element3:43
Enter element4:23
Enter element5:67
Enter element6:98
Enter element7:67
Non-duplicate items:
[23, 56, 43, 67, 98]
```

Program to remove the ith occurrence of the given word in a list:

```
a=[]
```

```
the number of elements in list:

for x in

the elements :



c=[]


word to remove:

the occurrence to remove:

for i in a:







not
```

```
for i in a:
```

```
number of repetitions is:
```

```
distinct elements are:
```

Here’s the output:

```
Enter the number of elements in List: 5
Enter the elements : hello
Enter the elements : bye
Enter the elements : world
Enter the elements : bye
Enter the elements : Python
['hello', 'bye', 'world', 'bye', 'python']
Enter word to remove: bye
Enter the occurrence to remove: 2
The number of repetitions is:
The distinct elements are: ['hello', 'world', 'python']
```

Program to search the number of times a particular number occurs in a list:

```
a=[]
```

```
number of elements:

for i in

element:

the number to be counted:

for j in a:

of
```

This is the output:

```
Enter number of elements: 5
Enter element: 23
Enter element: 34
Enter element: 23
Enter element: 78
Enter element: 56
Enter the number to be counted: 23
Number of times 23 appears is 2
```

Program to generate random numbers from 1 to 20 and append them to the listl:

```
import random

a=[]

number of elements:

for j in



list is:
```

The output is as follows:

```
Enter number of elements: 5
Randomized list is: [2, 15, 20, 1, 7]
```

Conclusion

This chapter covered the basic list operations with examples. The list is quite possibly the most widely used data type in Python. A Python List can be effectively recognized by square sections [] and stores objects of all types. The next chapter covers tuples in detail.

Questions

Write a Python function to interchange the first and last elements in a list.

Input: [12, 35, 9, 56, 24]

Output: [24, 35, 9, 56, 12]

What is the difference between append and extend?

What is the difference between and

Write a program to concatenate two lists.

How can you remove duplicate elements in a list?

CHAPTER 10

Tuples

Introduction

A **tuple** is a sequence of immutable or unchangeable Python objects, like lists. The difference between tuples and lists is that tuples cannot be changed. Additionally, lists use square brackets, whereas tuples use parentheses.

Structure

We will cover the following topics in this chapter:

Creating a tuple

Accessing values in tuples

Updating tuples

Deleting tuple elements

Basic tuple operations

Indexing, slicing, and matrices

Built-in tuple functions

Objective

This chapter aims to make the learner understand the concept of tuples and the various operations performed upon tuples. Several examples are also included in the chapter to help the learner understand the concept clearly and easily.

Creating a tuple

Creating a tuple is as simple as separating values using commas. You may also put these comma-separated values between parentheses.

Consider this example:

t

tuple

tuple

Comma is not mandated to create a single element tuple:

tup1 = (50,)

Accessing values in tuples

Use the square brackets to access values in a tuple and the index or indices to obtain a value of that index. Take a look at this example:

56.8

56.8)

Updating tuples

A tuple cannot be changed once it is created, so tuples are immutable.

However, there is a workaround. We can convert a tuple into a list, change it, and then convert it back into a tuple.

```
t1=(65,90,89.7,'abc',78)

t1[3]=88

Traceback (most recent

line in



TypeError: does not support item assignment



t1=t1+t2

print(t1)
```

Deleting tuple elements

Individual elements from a tuple cannot be removed, but there is nothing wrong with constructing another tuple without the unwanted elements.

Use the del statement to explicitly remove a tuple. Consider this example:

```
recent call last):

line 1, in

'tuple' object doesn't support item deletion

del(t)

t

recent call last):

line 1, in
```

```
t
```

```
name 't' is not defined
```

Note: An exception occurs. Tuple does not exist anymore after del tup, so we cannot delete a particular element from it.

Basic tuple operations

Len

This function is used to determine the length of the tuple.

Syntax:
len(tuple)

```
6
```

Concatenation

This function adds values of tuples on either side of the operator:

a=(1,2,3)

a+b

(1, 2, 3,

Repetition

This function concatenates copies of the tuple to create a joined tuple with the duplicate elements.

Membership

in - returns true if a character exists in the given string.

String to tuple

not in - returns a true value if a character does not exist in the given string.

```
'h' in t1

True

'h' in t2

False

'e' not in t1

False

'e' not in t2
```

```
True
```

Iteration

It indicates the repetition of a statement or group of statements while a given condition is TRUE.

:

```
print  (x,
```

Indexing, slicing, and matrices

Since tuples are sequences, indexing and slicing work the same way for tuples as they do for strings, assuming the following input:

```
90
```

```
'xyz'
```

```
'xyz'
```

Built-in tuple functions

Python includes the tuple functions listed in this section.

Tuple len() method

This method returns the number of elements in the tuple similar to the list.

Syntax:

len(tuple)

Parameters

tuple - This is a tuple of elements to count.

Return value

This method returns the number of elements in the tuple.

```
tuple1, tuple2 =

print tuple length :

print tuple length :
```

First tuple length

Second tuple length

Tuple max() method

The **max()** method returns the elements from the tuple with maximum value.

Syntax:

max(tuple)

Parameters

tuple - This is a tuple from which the max valued element is to be returned.

Return value

This method returns the elements from the tuple with maximum value.

```
tuple1, tuple2 =

print value element :

print value element :
```

```
Max value element : phy
Max value element : 700
```

Tuple min() method

The **min()** method returns the elements from the tuple with minimum value.

Syntax:

min(tuple)

Parameters

tuple - This is a tuple from which min valued element is to be returned.

Return value

This method returns the elements from the tuple with minimum value.

```
tuple1, tuple2 =

print value element :

print value element :
```

```
min  value  element  :  Python
min  value  element  :  100
```

Tuple tuple() method

The **tuple()** method converts a list of items into tuples.

Syntax:

```
tuple(seq)
```

Parameters

seq - This is a tuple to be converted into tuple.

Return value

This method returns the tuple.

```
list1=

tuple1=tuple(list1)

print elements : tuple1)
```

```
tuple elements : ('maths', 'che', 'phy', 'bio')
```

Programs

This is the program to remove all tuples in a list of tuples with the USN outside the given range:

```
lower range (starting with 12CS):

upper range (starting with 12CS):

for i in y:

if and
```

Output:

```
(starting with 123

(starting with 300
```

Here's the program to sort a list of tuples in increasing order by the last element in each tuple:

```
def

return

def

return
```

Output

```
Sorted:
[(20, 10), (23, 12), (12, 34), (22, 45), (45, 67)]
```

This program creates a list of tuples with the first element as the number and the second as the square of the number:

```
the lower range:
```

```
the upper range:

for x in
```

Output:

```
Enter the lower range: 2
Enter the upper range: 8
[(2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64)]
```

Conclusion

This chapter taught us tuples and the various operations performed upon them, and each operation was explained with the help of an example. You understood the basic concepts related to tuples, from creating a tuple and accessing values in tuples to updating tuples, using built-in functions in tuples, and so on.

In the next chapter, you will understand the dictionaries used in Python.

Questions

Create a tuple with a single item 5.

What are tuples used for in Python?

Are tuples mutable or immutable?

What is the importance of the len function in Python?

Explain the min() method with an example.

Compare the tuple max() and tuple min() methods.

How can you access values in tuples?

Explain the concatenation function used in tuples.

Explain any three built-in functions used in tuples in Python.

How is indexing done in tuples?

CHAPTER 11

Dictionaries

Introduction

A **dictionary** is entirely different for Python as it is not a sequence but a mapping. A mapping is the same as an object collection, but it stores objects using keys instead of relative positions. Moreover, mappings don't maintain any reliable left-to-right order; they map keys to the associated values.

Structure

We will cover the following topics in this chapter:

Creating a dictionary

Accessing the elements in a dictionary

Updating a dictionary

Deleting dictionary elements

Built-in dictionary functions

Objectives

The main objective of this chapter is to know more about dictionaries in Python. A dictionary is similar to a list in that it allows you to refer to a collection of data by a single variable name. However, it differs from a list in one fundamental way. Order is important in a list, and the elements in a list never change (unless you explicitly make changes to them). Since the order of elements in a list is important, you refer to each element using its index (its position within the list).

Note:

You can use only immutable objects like strings, numbers, or tuples for the keys of a dictionary. A tuple can contain any mutable or changeable object but cannot be modified post creation.

The key must be unique.

Syntax:

{:, :, ..., :}

Creating a dictionary

Curly brackets represent dictionary entries. Key-value pairs are separated by a colon in the dictionary.

An empty dictionary without any items is written with just two curly braces, i.e.,

```
# empty dictionary

my_dict = {}

# dictionary with integer keys

my_dict =

# dictionary with mixed keys

my_dict =

# using dict()

my_dict =

# from sequence having each item as a pair
```

```
my_dict =
```

Accessing the elements in a dictionary

You use the square brackets along with the key to access the elements of a dictionary.

Example:

```
is

is

name is Girish

is

age

are

Traceback (most recent call last):

  File line in

are
```

```
KeyError: 'marks'
```

Updating a dictionary

Dictionary entries can be changed. We can add new items or change the values of the existing ones using the assignment operator. The value is updated if the key already exists; otherwise, a new key-value pair is added to the dictionary.

dict

24, 90}

Deleting dictionary elements

The **pop()** method is used to remove an item from a dictionary. It returns the value of an item removed with the provided key. You can use **popitem()** to remove and return an arbitrary item (key, value) from a dictionary. The **clear()** method is used to remove all items at once. The del keyword can also be used to remove individual items or the entire dictionary.

create a dictionary

remove a particular item 16

16

remove an arbitrary item (1, 1)

delete a particular item

squares

remove all items

```
squares
```

```
{}
```

delete the dictionary itself

```
squares
```

```
squares
```

Properties of dictionary keys

There are no restrictions on dictionary values. A Python object can either be a standard object or a user-defined object, but this is not true for dictionary keys.

The two important properties of keys are as follows:

There will be no more than one entry per key, so duplicate keys are not allowed. If duplicate keys are encountered during assignment, the last assignment **wins.**

A key must be immutable type, such as a dictionary key. You can use strings, numbers, or tuples, but a list, such as ['key'], is an invalid key.

```
Traceback (most recent call last):

  File line in



unhashable 'list'
```

Built-in dictionary functions

Dictionary len() method

The **len()** method returns the total length of the dictionary. The number of items in the dictionary would be equal to this.

Syntax:

Here’s the syntax for the **len()** method:

len(dict)

Parameter

dict - Calculate the length of the dictionary

Return value

This returns the length of the dictionary.

Example:

Take a look at the following example:

is

length

Dictionary str() method

The **str()** method returns a printable string representation of a dictionary.

Syntax:
str(dict)

Parameter

dict - Dictionary

Return value

This method returns the string representation.

Example:

This example shows this method’s usage:

```
str(dict)

"{'name': 'Girish', 'age': 34}"
```

Dictionary_type()_method

The **type()** function returns the type of the passed variable. It will return a dictionary type if the passed variable is a dictionary.

Syntax:
type(dict)

Parameter

dict - This is a dictionary

Return value

The method will return the passed variable type.

Example:

Here's an example of the usage of the **type()** method:lang = 'Python'

digits =

digits_dict =

```
print(type(digits))

print(type(lang))

print(type(digits_dict))
```

Methods on dictionary

Methods that are available with a dictionary are explained as follows:

Here, we have considered **d** to be a dictionary.

Dictionary clear() method

The **clear()** method removes all items from the dictionary.

Syntax:

```
d.clear()
```

Parameters

NA

Return value

This method does not return any value.

Example:

This example shows the usage of the **clear()** method:

```
d.clear()

d
```

{}

Dictionary copy() method

The **copy()** method returns a copy of the dictionary.

Syntax:

dict.copy()

Parameters

NA

Return value

It returns a shallow copy of the dictionary.

Example:

This example shows the usage of the **copy()** method:

x>>>

Dictionary fromkeys() method

The **fromkeys()** method creates a new dictionary with keys from seq and values set to value.

Syntax:

dict.fromkeys(seq[, value]))

Parameters

seq – This parameter is used for preparing dictionary keys

value - If provided, the value will be set to this value

Return value

It returns the list.

Example:

This example shows the usage of the **fromkeys()** method.

Dictionary get() method

The **get()** method returns a value for the given key; it returns default value None if the key is not available.

Syntax:

None, None}

None, None, None}

The syntax for the **get()** method is as follows:

dict.get(key, default=None)

Parameters

key – The key to be searched in the dictionary

default – The default value to be returned if the key does not exist

Return value

This method returns a value for the given key; it returns the default value None if the key is not available.

Example:

The following example shows the usage of the **get()** method:

34

male

Dictionary items() method

The **items()** method returns a list of dictionary (key, value) tuple pairs.

Syntax:

```
dict.items()
```

Parameters

NA

Return value

It returns a list of tuple pairs.

Example:

This example shows the usage of the **items()** method:

```
print(d.items())
```

Dictionary keys() method

The **keys()** method returns a list of all the keys in the dictionary.

Syntax:

dict.keys()

Parameters

NA

Return value

This function will return a list of all the keys in the dictionary.

Example:

Here's an example that shows the usage of the **keys()** method:

Dictionary setdefault() method

The **setdefault()** method is similar to but it will set if the key is not already in dict.

Syntax:

dict.setdefault(key, default=None)

Parameters

key – The key to search

Default - The value to be returned if the key cannot be found

Return value

The method returns the value of the given key if it is present in the dictionary; it returns the default value if it isn't present.

Example:

This example shows the usage of the **setdefault()** method:

```
>>>
```

```
>>>

34

>>>

None

>>> d
```

Dictionary update() method

Dictionary dict2's key-value pairs are added to dict using the **update()** method; there is no return from this function.

Syntax:

dict.update(dict2)

Parameters

dict2 - Dictionary is to be added into the dict.

Return value

It does not return any value.

Example:

The following example shows the usage of the **update()** method:

d

Dictionary values() method

The **values()** method returns a list of all the values available in a given dictionary.

Syntax:

dict.values()

Parameters

NA

Return value

This method returns an index of the values in the dictionary.

Example:

The following example shows the usage of the **values()** method:

print :

Values

Programs

Program to add a key-value pair to the dictionary:

a key to be added:

its value:

d.update({k:v})

dictionary is

```
enter a key to be added: vegetable
enter its value: tomato
updated dictionary is :
{'fruits': 'apple', 'color': 'pink', 'vegetable': 'tomato'}
```

Program to concatenate two dictionaries into one:

d.update(d2)

dictionary is

concatenated dictionary is :
{'fruits': 'apple', 'color': 'pink', 'name': 'girish', 'age': 34}

Program to generate a dictionary that contains numbers (between 1 and n) in the (x,x*x) form:

a number:

d={x:x*x for x in

dictionary is

enter a number: 6

the dictionary is {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

Program to find the sum of all values of a dictionary:

sum of values in the dictionary is

```
the sum of values in the dictionary is
100
```

Program to take a string as a parameter and generate a frequency of characters contained in it:

```
a string:

d=dict()

for i in a:

d):




of each character is
```

```
enter a string: dictionary
frequency of each character is
{'d': 1, 'i': 2, 'c': 1, 't': 1, 'o': 1, 'n': 1, 'a': 1, 'r': 1, 'y': 1}
```

Program to count the frequency of words in a dictionary:

```
a string:

l=list()

d=dict()


for i in l:


d):




frequency of words is
```

```
enter a string: abc xyz mno xyz pqrabcabc
the frequency of words is  {'abc': 3, 'xyz': 2, 'mno': 1, 'pqr': 1}
```

Program to get a minimum and a maximum value in a dictionary:

```
value is

value is
```

```
maximum value is d
minimum value is a
```

Program to find the number of positive and negative numbers from a list of numbers:

```
l1=[23,-33,90,-3,-12,-90,37,6,-7]

d=dict()



for i in l1:
```

```
{'pos': 4, 'neg': 5}
```

Program to convert octal to binary using a dictionary:

```
a




for i in num:




in binary is
```

```
Enter a number32
number in binary is  0b011010
```

Program to replace dictionary values with their sum:

```
result=1

for key in d:

    result=result*d[key]
```

result is

the result is
240000

Conclusion

In this chapter, you learned about dictionaries in Python. You also understood the basic concepts related to creating a dictionary, accessing the values in it, etc. We discussed built-in functions in the dictionary elaborately and looked at examples to help learners understand the concept.

In the next chapter, we will study more about the functions in Python. We will discuss built-in functions and user-defined functions with examples.

Questions

How can you create a dictionary in Python?

How can you get the number of items in a dictionary?

Find the output of the following program:

```
D = dict()

for x in

=

=

print(D)
```

Compare the setdefault() and get() methods.

What is the significance of the dictionary items() method?

CHAPTER 12

Functions

You can write more complicated programs in extended, complex modules, but detailed modules are harder to write and difficult to understand. A better approach is to divide a complicated program into smaller, simpler, more focused modules and functions.

A function in Python is a named block of code. It performs a particular task. We can break down our program into smaller, modular chunks using parts. As our program grows, functions make it more manageable and organized. Additionally, they avoid repetition and make code reusable.

Structure

We will cover the following topics in this chapter:

Built-in functions

User-defined functions

Defining a function

Function calling

Docstring

Return statement

Pass by reference vs value

Function arguments

Required arguments

Default arguments

Keyword arguments

Arbitrary arguments

Anonymous functions

Use of lambda function

Scope of variable

Recursion

Objectives

This chapter explores functions in Python. There are mainly two types of functions: built-in functions and user-defined functions. We will focus on explaining such concepts with examples. The current chapter emphasizes the core concepts related to functions like pass by reference vs value, recursion, and different types of arguments.

Built-in functions

Just like Python has built-in operators (such as plus and minus for addition, asterisk for multiplication, and so on), it has built-in functions. You can use them in your programs as pieces of code. A built-in function can be used by specifying its name and, typically, by providing some information. With that information, the function performs some work and returns the results. For example, and so on.

User-defined functions

These are functions that we define ourselves to accomplish specific tasks.

User-defined functions have the following advantages:

Using user-defined functions, a large program can be broken up into smaller parts that are easier to understand, maintain, and debug.

When repeated code appears in a program, you can include it and execute it when needed by calling a function.

Defining a function

The def keyword marks the beginning of the function header, followed by the function name and parentheses

The function name identifies it. In Python, function names also follow the same rules that are followed by identifiers.

Parameters and arguments should be enclosed in parentheses; parameters can also be defined inside these parentheses.

It describes what the function does with an optional documentation string (docstring).

A colon marks the end of the function header.

At least one valid Python statement must appear in the body of the function; statements must be indented equally (usually 4 spaces).

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return

Syntax:

```
deffunction_name(parameters):
"""docstring"""
statement(s)
```

Example

```
"""This is the first function"""

        print(name)

return
```

Function calling

Defining a **function** gives it a name, which specifies the parameter that will be included within the function and the structure of the blocks of code in it.

When we define a function, we can call it from another function, program, or even the Python prompt. The name of the function and its parameters are typed to call it. Here’s an example to call the **first()** function:

Docstring

The **docstring** is the first string after the function header, and it stands for documentation string. It is used for briefly describing a function; documentation is a good programming practice, even if it is optional.

There is a docstring immediately below the function header in the preceding example. Usually, triple quotes are used as docstrings often span multiple lines. The string is available in the **__doc__** attribute of the function.

Example:

```
b):

''' Adding two numbers '''

return a + b
```

The docstring of the function can be printed using the following code:

```
print(sum.__doc__)
```

Return statement

The **return** statement is used to exit the function and return to its starting point.

Syntax:

```
return [expression_list]
```

The statement can contain expressions that are evaluated, after which the value is returned. The function returns None if there is no expression in the statement or if the **return** statement itself is absent inside the function. Here's an example:

```
"""This is the first function"""
```

```
hello world

None
```

Example:

```
def

if




return ans

a number:


a number:
```

Output

```
enter a number: 24

even
enter a number: 65
odd
```

Pass by reference vs. value

Consider this example for better understanding:

```
"changing the original list using function"

print of list inside function before change: l1)



print of list inside function after change: l1)

return

l1 =

change(l1)

of list are: l1)  #values outside function
```

Python passes parameters (arguments) by reference, which means when you change what a parameter refers to within a function, the change also reflects in the calling function.

Output

```
Values of list inside function before change: ['apple', 'mango',
'litchi', 'orange']
Values of list inside function after change: ['apple', 'mango',
'banana', 'orange']
Values of list are: ['apple', 'mango', 'banana', 'orange']
```

Function arguments

These types of formal arguments can be used to call a function:

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

Required arguments

Positional arguments are known as **required arguments** and are passed to a function in the correct order. The arguments in the function call should match the number in the function definition.

Example:

```
defcal(a,b):

"""program to calculate sum of 2 numbers"""

    sum=a+b

return sum

1st

2nd

print(cal(a,b))
```

Output

```
enter 1st number23
enter 2nd number22
45
```

Here, **cal()** has two parameters. This function runs smoothly and without any errors since it is called with two arguments. If we call it with different arguments, the interpreter will complain. This function is called with one argument and without one argument, along with the respective error messages.

```
print(cal(a))

Traceback (most recent

line in

    print(cal(a))

TypeError: cal() positional argument: 'b'
```

Default arguments

A **default argument** is an argument that assumes a default value if a value is not provided in the function call for that argument.

The user may not want to provide values for some parameters for some functions, so you may want to make them optional and use default values. Default argument values are used to accomplish this. Add the assignment operator followed by the default value to the parameter name in the function definition if you want to specify default argument values for parameters.

A constant value should be used for the default argument; it should be an immutable value by default.

Example:

```
"""program to calculate sum of 2 numbers"""

        sum=a+b

return sum
```

```
print(cal(a,b))
```

```
22
```

```
print(cal(a))
```

```
30
```

Note: We cannot have a parameter with a default argument value preceding a parameter without a default argument value in a function's parameter list.

The parameters are assigned values based on their positions. For example, **defcal(a, b=5)** is valid, but **defcal(a=5, b)** is not valid. Take a look at this:

```
a=10



b=12
```

```
print(cal(a,b=5))
```

```
15
```

```
print(cal(a=5,b))
```

SyntaxError: positional argument follows keyword argument

Keyword arguments

Arguments related to function calls are known as **keyword** In function calls, parameter names are used to identify keyword arguments.

If you have some functions with many parameters and only want to specify some of them, you can call them keyword arguments because we now use the name (keyword) instead of the position (which we have been using all along) as the parameter name for the arguments to the function.

You can skip arguments or place them out of order because the Python interpreter can match the values with the parameters using the keywords provided. There are two advantages of Keyword arguments:

The function is easier to use since we don't have to worry about the order of the arguments.

Only the parameters we wish to set values for can be given values, provided that the others have default arguments.

Example:

defcal(a,b):

```
"""program to calculate sum of 2 numbers"""

        sum=a+b

return sum

# 2 keyword argument

36

# 2 keyword argument(out of order)

36


1 positional, 1 keyword

28


SyntaxError: positional argument follows keyword argument
```

During a function call, we can mix keyword arguments with positional arguments. However, keyword arguments should follow positional arguments; positional arguments after keyword arguments will result in errors.

Arbitrary_arguments

They are also known as variable length arguments.

In some cases, we do not know in advance how many arguments will be passed to a function. This kind of a situation can be handled in Python through function calls with **arbitrary** For this kind of an argument, we use an asterisk before the parameter name in the function definition.

Example:

```
*vartuple):

"Arbitrary arguments"

print is:

print (arg1)

for var in vartuple:

print (var)

return
```

Output

```
Output  is:
34
Output  is:
10
20
30
40
```

Anonymous functions

An **anonymous function** in Python is one with no name. The def keyword is used to define normal functions, while the lambda keyword is used to define anonymous functions. So, anonymous functions are also called lambda functions. Here's what you need to know about them:

Lambda forms can take any number of arguments, but they will return only one value in the form of an expression. Neither can they contain commands nor can they contain multiple expressions.

Lambda requires an expression, so an anonymous function cannot be a direct call to print.

In lambda functions, variables can only be accessed from the parameter list and from the global namespace, and they have their own local namespace.

Although lambdas appear to be one-line versions of functions, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function during invocation for performance reasons.

Syntax:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Example:

lambda function

```
print(double(5))
```

```
10
```

Use of lambda function

We use lambda functions when we need a nameless function for a short period of time. They are used as arguments to a higher-order function (a function that takes other functions as arguments). Lambda functions are commonly used as arguments to built-in methods like and so on.

Example:

A function and a list are both arguments for the **filter()** function in Python. With all the items in the list, the function is called and a new list is returned; it contains the items for which the function evaluates to True.

Here's an example showing the use of the **filter()** function to filter out even numbers from a list:

```
print(list2)
```

Output

```
[4, 6, 8, 12]
```

Python's **map()** function takes a function and list. A list containing the items returned by the function for each list element is returned when the function is called with all items in a list.

For example, the use of the **map()** function to double all the items in a list:

```
print(list2)
```

Output

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Scope of variable

A variable's scope is the area of a program where it is recognized. A function's parameters and variables are not visible from outside, so they have local scope.

The life time of a variable is the period during which it remains in memory. Variables inside a function exist for the duration of the function; they are destroyed once we return from the function. As a result, a function does not remember the value of a variable from its previous calls.

Variable scope defines which part of the program can access a particular identifier. In Python, variables fall into two basic scopes:

Global variables

Local variables

The variables inside a function body have a local scope, while the variables defined outside have a global scope.

Local variables can only be accessed within the function in which they are declared, while global variables can be accessed by all functions throughout the program body. The variables declared inside a function are brought into scope as soon as you call them.

Here’s a simple example:

```
deffunc():

    str = "hello"

of str inside

str = "world"


of str outside
```

Output

```
Value of str inside function: hello
Value of str outside function: world
```

Recursion

Recursion is the process of defining something in terms of itself. This refers to the ability to save its state or value in register or stack before calling itself.

A **recursive function** is a function that calls itself till it is terminated by an exit condition. A recursive function usually returns the value of the function call.

Example:

```
"""This is a recursive function

to find the factorial of an integer"""

if x == x ==




return (x *

n = a number:
```

```
factorial  n,  fact(n))
```

Output

```
Enter a number: 5
The factorial of 5 is 120

# While calling

fact(5)
5 * fact(4)
5 * 4 * fact(3)
5 * 4 * 3 *fact(2)
5 * 4 * 3 * 2 *fact(1)

# While returning

5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2

5 * 4 * 6
5 * 24
120
```

Advantages of recursion

Recursive functions make the code look clean and elegant

A complex program can be broken down into simpler sub-problems using recursion

Sequence generation is easier with recursion as compared to using some nested iteration

Disadvantages

The logic behind recursion becomes very complex sometimes

Recursive calls are expensive (inefficient) as they take up a lot of memory and time

Recursive functions are hard to debug

Comparisons between iteration and recursion

In recursion, a function calls itself until the base condition is reached. On the other hand, iteration means the repetition of code until a condition fails.

Iteration involves four steps: initialization, condition, execution, and updation; recursion only has a base condition.

Recursion is slower than iteration and takes more memory than iteration due to the overhead of maintaining stack.

Programs

Program to find the maximum of three numbers:

```
if a>b and

elif and b>a:

elif and

a:

b:

c:

number is
```

Output

```
enter a: 24
enter b: 45
enter c: 33
largest number is  45
```

Program to add all the numbers in a list:

```
for i in l1:

        sum+=i

return sum



of list
```

Output

```
sum of list is 75
```

Program to reverse a string:

```
def
```

```
for i in str:

        rev=i+rev

return rev

a string:

reverse of string is
```

Output

```
enter a string: hello
the reverse of string is olleh
```

Python function to calculate the factorial of a number; the function accepts the number as an argument:

```
for i in

        fact=fact*i
```

```
return fact

a number:

of %d
```

Output

```
enter a number: 5
factorial of 5 is 120
```

Program that accepts a list and returns a new list with the unique elements of the first list:

```
in l:

not in x:

x.append(a)


print(unique([1,2,4,6,7,6,5,3,5,6,6]))
```

Output

[1, 2, 4, 6, 7, 5, 3]

Program to accept a string and calculate the number of uppercase and lowercase characters:

```
def count(str):

for i in str:

a

characters=

characters=
```

Output

```
enter a string: Hello World
```

```
Uppercase characters= 2
lowercase characters= 8
```

Program to calculate the sum of a series:

1/2 – 3/4 + 5/6 .... n

```
n:




for i in




of series
```

Output

```
enter n: 2
sum of series is
-0.25
```

Program to check whether a number is a perfect number:

```
#perfect number


for i in n):

% i ==

            sum1 = sum1 + i

return sum1

n = any number:

sum1 = 0

ans=perfect(n,sum1)

if (ans == n):

number is a Perfect


number is not a Perfect
```

Output

```
Enter any number: 6
The number is a Perfect number!
```

Python program to evaluate a binomial expression:

```
def



for i in

        f=f*i

return f

value of n:

value of r:

ans=fact(n)/fact(r)/fact(n-r)

coefficient is
```

Output

```
enter value of n: 8
enter value of r: 3
binomial coefficient is 56.0
```

Program to find the HCF of two numbers:

```
y):

if x > y:

        small = y


        small = x

for i in +

% i == and (y % i ==

  hcf = i

return hcf

a = first number:

b = second number:

H.C.F. hcf(a, b))
```

Output

```
Enter first number: 128
Enter second number: 36
The H.C.F. of 128 and 36 is 4
```

Program to generate a Fibonacci series using recursion:

```
# fibonacci

deffibo(n):

if



if or






number of terms:


is


for i in
```

Output

```
enter number of terms: 6
fibonaccies is
0 1 1 2 3 5
```

Program to reverse a string using recursion:

```
def



return s1




a string:

string is
```

Output

```
enter a string: hello python
reversed string is
nohtypolleh
```

Conclusion

In this chapter, we learned about functions, their importance in Python, and their types. We also looked at the different types of arguments used in Python, along with their examples. Finally, we compared pass by reference vs value.

In the next chapter, we will focus on the concept of modules in Python. Modules are files containing Python code. They can define functions, classes, and variables.

Key terms

Built-in function

User-defined function

Function calling

Pass by reference

Pass by value

CHAPTER 13

Modules

Introduction

A Python module is the highest level of program organization, packing code, and data for reuse and easy interoperability at the functional level and for providing self-contained namespaces to prevent variables from clashing across different programs.

Modules are files containing Python code that define functions, classes, and variables. Runnable code can also be included in a module. These pieces should be shared, so Python allows a module to "bring in" and use attributes from other modules to take advantage of work that has been done, maximizing code reusability. This process of associating the attributes from other modules with your module is called importing.

Structure

We will cover the following topics in this chapter:

The import statement

The from..import statement

The from..import * statement

Executing modules as scripts

The module search path

The dir() function

The globals() and locals() functions

The reload function

Packages in Python

Objective

This chapter aims to help the reader understand the Python module. It also explains the use of import statements in Python and focuses on executing modules as scripts in Python. Finally, the chapter explores the packages in Python.

The import statement

Any Python source file can be used as a module by executing an **import** statement in another Python source file.

Syntax:

```
import module1[, module2[,... moduleN]
```

When the interpreter finds an import statement, it imports the module if it is present in the search path.

Search The search path refers to a list of directories that the interpreter searches before importing a module.

For example, this is a **test.py** module:

```
defprint_func(p):

: p

return
```

To import the **test.py** module, you need to put the import test at the top using the following command:

```python
import support
```

If the code is executed, the result will be as follows:

```
Hello: Sukhman
```

The from...import statement

You can import specific attributes from a module into the current namespace using Python's **from** statement.

Syntax:

```
from modname import name1[, name2[, ... nameN]]
```

For example, import function Fibonacci from the **fib** module:

```
    r=[]

    a, b = 1

while (b

r.append(b)

        a, b = b, a+b
```

Use the following to import the **fib** function from the **Fibonacci.py** module:

```
fib(50)
```

The statement does not import the entire Fibonacci module into the current namespace; it just introduces the fib item from the Fibonacci module into the global symbol table of the importing module.

The from...import * statement

All the names in a module can be imported into the current namespace using the following import statement.

Syntax:

```
from modname import *
```

A module can be imported into the current namespace using this statement, but the information should be used sparingly.

Executing modules as scripts

Module names (as strings) are available as the value of a global variable **__name__** within a module. The code in the module is executed as you imported it, but the **__name__** will be set to

Consider that you run a Python module with:

Python fibo.py

Then, the module code will be executed just as it had been imported, but **__name__** will be set to When we add this code at the end of your module, you can execute modules as scripts:

If

Import sys

Fib(int(sys.argv[1]))

Since the code that parses the command line only runs when the module is executed as the **main** file, it can be used as both a script and an importable module:

The code is not run if the module is imported:

```
import fibo
```

The module search path

The interpreter first searches for the built-in module named spam when importing a module named spam. If **spam.py** is not found, the **sys.path** variable is used to search a list of directories for a file named These locations are used to initialize

The directory that contains the input script (or the current directory)

This is the Pythonpath (a list of directory names with the same syntax as the shell variable PATH)

Installation-dependent defaults

Python programs can modify **sys.path** during initialization. The directory containing the script being run is placed at the start of the search path, before the standard library path. This means when scripts are loaded from that directory, modules from the library directory of the same name are not loaded. Unless the replacement is intended, this is an error.

Note: Here is a typical PYTHONPATH from a Windows system.

```
set PYTHONPATH=c:\python34\lib;
```

The dir() function

Directory() returns a sorted list of strings containing the names defined in a module. The list consists the names of all the variables, and functions defined in a module.

Here’s an example:

```
import math

con =

print (con)
```

When the preceding code is executed, the result will be as follows:

```
['__doc__', '__loader__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

In this example, **__name__** is the module's name, and **__file__** is the filename from which the module was loaded.

The globals() and locals() functions

You can use **globals()** and **locals()** to return names in the global and local namespaces depending on where they are called.

If the **locals()** method is called from within a function, it returns all the names that can be accessed locally.

When the **globals()** method is called from within a function, it will return all the names that can be accessed globally.

Both of these functions return dictionaries, so names can be extracted using the **keys()** function.

The reload() function

The code in the top portion of a module is only executed once when imported into a script. So, you can use the **reload()** function if you want to rerun the top-level code in a module; it imports a previously imported module again.

Syntax:

```
reload(module_name)
```

In this case, module_name is the name of the module you want to reload, not the string containing the module name.

For example, do the following to reload fibonacci module:

```
reload(fibonacci)
```

Packages in Python

Modules and sub packages make up a package, which is a hierarchical file directory structure for defining a single Python application environment. Packages were added to Python 1.5 to overcome various problems, including:

Adding hierarchical organization to flat namespace

Allowing developers to group related modules

Allowing distributors to ship directories versus a bunch of files

Helping resolve conflicting module names

Along with classes and modules, packages use the familiar attribute/dotted attribute notation to access their elements. Importing modules within packages needs the standard import and from-import statements.

Suppose a **Pots.py** file is available in phone directory. It has the following line of source code:

#!/user/bin/python3

```
print Pots
```

Similarly, we have two other files with different functions but the same name as above. They are as follows:

The **Phone/Isdn.py** file with the **Isdn()** function

The **Phone/G3.py** file with the **G3()** function

Now, create the **__init__.py** file in the **Phone** directory:

Phone/__init__.py

You need to add explicit import statements in **__init__.py** to make all of your functions available when you have imported Phone, as shown here:

```
from Pots import Pots
from Isdn import Isdn

from G3 import G3
```

You have all these classes available when you import the Phone package after you add these lines to

```
#!/usr/bin/python3

# Now import your Phone Package.
```

```
import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()
```

If the preceding code is executed, the result will be as follows:

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

This example uses a single function per file, but you can keep multiple functions in your files. Additionally, you can define different Python classes in those files and then create packages based on those classes.

Programs

Program to import the value of pi using math module:

```
import math
```

value of pi is ",

Program to create a file called num.py containing the square and cube of a number and import to print the same:

Num.py

```
return x*x
```

```
return
```

Python interpreter

27

Program to print the current time:

9 2018'

Program to print the current working directory:

```
import os
```

Conclusion

In this chapter, we learned about the concept of Python modules. We also discussed the import statements and their various usages in Python with examples. Additionally, we explored the packages in Python.

In the next chapter, we will learn about files I/O.

Questions

What is a module in Python?

What is an import statement in Python?

Explain the dir() function with an example.

Compare the globals() function and the locals () function.

What is the reload() function?

Explain packages in Python.

CHAPTER 14

Files I/O

Introduction

You have been reading and writing to the standard input and output until now. Now, we will see how to use actual data files.

Python provides the basic functions and methods necessary to manipulate files by default. However, you can perform file manipulation using a **file** object.

Python provides numerous built-in functions that are readily available to us at the Python prompt. Some of the functions, like **input()** and are widely used for standard input and output operations, respectively. Let's take a look at the output section first.

Python output using the **print()** function: We use the **print()** function to output data to the standard output device (screen). We can also output data to a file, but we will discuss this later.

Here's an example of its use:

```
print('This sentence is output to the screen')
```

Our programs were static until now, and the value of variables was defined or hard coded into the source code.

To allow flexibility, we might want to take the input from the user. In Python, we have the **input()** function to allow this. The syntax for **input()** is as follows:

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Structure

We will cover the following topics in this chapter:

Open and close files

Modes of opening a file

The file object attributes

Reading and writing files

Renaming and deleting files

File methods

Objectives

The main objective of this chapter is to help the learner understand all I/O functions available in Python. It intends to explain the concept of opening and closing files in Python and reading, writing, renaming, and deleting files. Finally, it explores the file methods available in Python.

Open and close files

So far, your input and output have been through the standard channels. We will now see how to use actual data files.

The open function

Before you can read or write a file, you have to open it using Python's built-in **open()** function. This function creates a file object.

Syntax:

file object = open(file_name [, access_mode][, buffering])

Parameters:

The **file_name** argument contains the name of the file you wish to access.

Using you can determine how the file should be opened; for example, reading, writing, or appending. As far as file access modes are concerned, read (r) is the default.

Buffering is not performed if the buffering value is set to 0. When the buffering value is 1, line buffering is performed when accessing a file. If you specify more than one as the buffering value, the buffer size will be indicated. The default buffer size (default behavior) is negative.

Modes of opening a file

The following table lists all the different modes of file operations:

operations:
operations: operations:
operations: operations:
operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations:
operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations:
operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations: operations:

operations: operations: operations: operations: operations: operations: operations:
operations: operations:
operations: operations:

operations: operations:
operations: operations:

operations: operations: operations: operations: operations: operations:
operations: operations:
operations: operations:

Table 14.1: *Modes of opening a file*

The file object attributes

Once a file is opened and we have one file object, we can use some attributes to get information about it.

The following are the attributes:

Returns true if file is closed; returns false otherwise

Returns the access mode with which the file was opened

Returns the name of the file

Example:

```
fo =

print of the file: fo.name)

print or not : fo.closed)

print mode : fo.mode)
```

Output:

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
```

The close() method

A file object's **close()** method flushes any unwritten information and closes it.

The reference object of the file is reassigned to another file. Python automatically closes the file, but it is still a good practice to close a file using the **close()**

Syntax:

```
fileObject.close();
```

Reading and writing files

We will look at how the **read()** and **write()** methods are used to read and write files.

The write() method

The **write()** method writes any string to an open file. It does not add a newline character to the end of the string.

Syntax:

fileObject.write(string);

Parameter:

String: Content to be written into the opened file.

file =

hard to achieve success.\nKeep

If you open the file named it will contain the following content:

Work hard to achieve success.
Keep moving!!

The read() method

The **read()** method reads a string from an open file.

Syntax:

```
fileObject.read([count]);
```

Parameter:

Count: This is the number of bytes to read from an opened file. If count is missing while trying to read as much data as possible, and perhaps until the end of the file, then this method starts reading from the beginning of the file.

Example:

```
file =

str =

print String is : str)
```

Output:

Work hard

Note: Python strings can have binary data and not just text.

Renaming and deleting files

The rename and delete operations are performed by the OS module of Python. To use it, we need to import it and then call related functions.

The rename() method

The **rename()** method changes the current filename to the new filename.

Syntax:

```
os.rename(current_file_name, new_file_name)
```

Example:

```
Import os
```

The remove() method

The **remove()** method is used to delete files.

Syntax:

os.remove(file_name)

Example:

File methods

File objects are created using the open function, and the following functions can be called on this object.

File flush() method

As with stdio'sfflush, **flush()** flushes the internal buffer; this may be a no-op on some file-like objects.

Python automatically flushes files when they are closed. Before closing a file, flush the data if necessary.

Syntax:

fileObject.flush()

Parameters:

NA

Return value:

It does not return any value.

Example:

file =

File fileno() method

Using the underlying implementation can request I/O operations from the operating system based on a file descriptor.

Syntax:

fileObject.fileno()

Parameters:

NA

Return value:

Integer file descriptors are returned by the method.

Example:

```
file =

fid = file.fileno()

print Descriptor: fid)
```

Output:

```
File Descriptor:  3
```

File isatty() method

The isatty() method returns true if the file is connected (to a secondary device) to a tty(-like) device, otherwise it returns false.

Syntax:

fileObject.isatty()

Parameters:

NA

Return value:

This method returns true if the file is connected (is associated with a terminal device) to a tty(-like) device, otherwise it returns false.

```
file =

ret = file.isatty()

print value : ret)
```

Output:

```
Return value : False
```

File next() method

There is no **next()** method available in the Python 3 file object. By calling its **__next__()** method, Python 3's **next()** function retrieves the next item from an iterator. The default is returned if the iterator is exhausted; **StopIteration** is raised. You can use this method to read the next input line from the file object.

Syntax:

next(iterator[,default])

Parameters:

iterator : The file object from which the lines are to be read

default : This is returned if the iterator is exhausted; StopIteration is raised if not given

Return value:

Returns the next input line.

Example:

Let's assume that a test.txt file contains the following lines:

Computer Networks
Discrete Structures
Functional English

Consider that we write the following program:

```
file

in

    line

print No %d - %s" % line))

file.close()
```

Output:

```
Line No 0 – Computer Networks

Line No 1 -  Discrete Structures
Line No 2 – Functional English
```

File readline() method

The **readline()** method reads a single line from the file. The string contains a trailing newline character.

Syntax:

fileObject.readline(size)

Parameters:

size - This is the number of bytes to be read from the file.

Return value:

Lines can be read from a file with this method. A non-negative size argument represents the maximum number of bytes, including the trailing newline. An incomplete line may be returned if the size argument is not given. An empty string is returned when EOF is encountered immediately.

Example:

Let's assume that a **test.txt** file contains the following lines:

Computer Networks

```
Discrete Structures
Functional English
```

Consider that we write the following program:

```
file =

line = file.readline()

print Line: %s" % (line))

line = file.readline(3)
```

Output:

```
Read Line – Computer Networks
Read Line -  Discrete Structures
Read Line – Functional English
```

File readlines() method

The reads until EOF using **readline()** and returns a list containing the lines.

Syntax:

fileObject.readlines(sizehint)

Parameters:

sizehint - The number of bytes to read from the file

Return value:

The method returns a list of lines. An optional sizehint argument causes whole lines (possibly rounding up to an internal buffer size) to be read instead of reading up to EOF. An empty string is returned when EOF is encountered immediately.

Example:

Let's assume that a **test.txt** file contains the following lines:

Computer Networks
Discrete Structures

Functional English

Consider that we write the following program:

```
file =

line = file.readlines()

print Line: %s" % (line))

line =

print Line: %s" % (line))

file.close()
```

Output:

```
Read Line: [Computer Networks\n', Discrete Structures\n',
Functional English\n']
Read Line:
```

File seek() method

The **seek()** method sets the file's current position at the offset. There are two values for the whence argument: 0, which means absolute file positioning, and 1, which means seeking relative to the current position and 2 means seeking relative to the file's end. There is no return value.

Note:

Using either 'a' or 'a+' for opening the file for appending will undo any **seek()** operation at the next write.

This method is essentially no-op when an append mode file is opened, but it remains useful for apps opened in reading mode (mode 'a+').

If the file is opened in the text mode with 't', only offsets returned by tell() are valid. A different offset causes undefined behavior.

Note that not all objects are seekable.

Syntax:

```
fileObject.seek(offset[, whence])
```

Parameters:

offset- This is the position of the read/write pointer within the file.

By default, 0 indicates that the position of the file is absolute, 1 means that the position is relative to the current position, and 2 means that the position is relative to the file's end.

Return value:

It does not return any value.

Example:

Let’s assume that a **test.txt** file contains the following lines:

```
Computer Networks
Discrete Structures
Functional English
```

Consider that we write the following program:

```
file =

line = file.readlines()
```

```
print Line: %s" % (line))



line = file.readline()

print Line: %s" % (line))

file.close()
```

```
Read Line: [Computer Networks\n', Discrete Structures\n',
‘Functional English\n']
Read Line: Computer Networks
```

File tell() method

The tell() method returns the current position of the file read/write pointer within the file.

Syntax:

```
fileObject.tell()
```

Parameters:

NA

Return value:

The method returns the current position of the file read/write pointer.

Let's assume that a **test.txt** file contains the following lines:

```
Computer Networks
Discrete Structures
Functional English
```

Consider that we write the following program:

```
file =

line = file.readline()

print Line: %s" % (line))



print position :

file.close()
```

Output:

```
Read Line: Computer Networks
Current Position: 18
```

[Programs]

Program to write content in a file:

fo =

real man spends time with his

added to the

Output:

Data added to the file.

Program to read the contents of a file:

fo =

real man spends time with his

added to the

Output:

```
Enter the name of the file with .txt extension: file.txt
A real man spends time with his family.
```

Program to count the number of words in a text file:

```
file name:



with as f:

for line in f:

        words=line.split()



of words:
```

Output:

```
Enter file name: file.txt
Number of words:
8
```

Program to count the number of lines in a text file:

```
file name:

with as f:

for line in f:

of lines:
```

Output:

```
Enter file name: file.txt
Number of lines:
1
```

Program to read a string from the user and append it into a file:

```
a file:

string to append:
```

```
of append file:

line1=file4.readline()

          line1=file4.readline()
```

Output:

```
A real man spends time with his family.
```

Program to count the number of occurrences of a word in a text file:

```
file name:

word to be searched:

with as f:
```

```
for line in f:

           words=line.split()

for i in words:




of the words:
```

Output:

```
Enter file name: file.txt
Enter word to be searched: a
Occurrences of the words:
4
```

Program to copy the content of one file into another:

```
as f:

as f1:

for line in f:

                        f1.write(line)
```

Program that reads a text file and counts the number of times a certain letter appears in it:

```
file name: ")

letter to be searched: ")


as f:

in f:



in words:

in i:



the letter: ")

print(k)
```

Output:

```
Enter file name: file.txt
Enter letter to be searched: t
Occurrences of the letter:
6
```

Program to read a text file and print all the numbers present in it:

```
file name: ")

as f:

in f:


in words:

in i:


print(letter)
```

Output:

```
Enter file name: file.txt
No digit.
```

Program to append the contents of one file to another:

file to be read from: ”)

file to be appended to: ”)

Program to count the number of blank spaces in a text file:

file name: ")

as f:

in f:

in words:

```
in i:



the letter: ")

print(k)
```

Output:

```
Enter file name: file.txt
Occurrences of the letter:
96
```

Conclusion

In this chapter, we learned about different I/O functions in Python. We also learned about the opening and closing of files in Python and understood different file methods with the help of examples. Additionally, we covered reading, writing, renaming and deleting files with examples.

In the next chapter, we will learn about exception handling in Python.

Key terms

open()

close()

rename()

read()

write()

remove()

CHAPTER 15

Exception Handling

We might have encountered several kinds of run-time errors while writing programs, such as integer division by zero, accessing a list with an out-of-range index, using an object reference set to None, and attempting to convert a non-number to an integer. To this point, all of our run-time errors have resulted in the program's termination. Python provides an exception handling mechanism that allows programmers to deal with run-time errors and much more. Rather than permanently terminating execution, a program can detect the problem and execute code to correct the issue or manage it in other ways.

Instead, it raises an exception, indicating that something exceptional has happened. Then, Python stops what it's doing and displays an error message detailing the exception if nothing is done about it.

Structure

We will cover the following topics in this chapter:

Standard exceptions

Python assertion

Handling an exception

The except clause with else clause

With no exceptions except close

The except clause with multiple exceptions

Try-finally clause

With statement

Argument of an exception

Raising an exception

User-defined exceptions

Objectives

The main objective of this chapter is to help the learner understand the use of exceptions in Python. It explains the concept of handling exceptions in Python with examples and walks learners through the idea of raising an exception in Python. Several examples are included in the chapter for a practical understanding of the concepts.

Let's take a simple example of Python raising an exception:

```
s="hello"

int(s)

Traceback (most recent

line in



ValueError: invalid literal with base 'hello'
```

Python tries to convert the string "Hello" to an integer. Since it cannot do this, Python raises an exception and displays the details.

Using Python's exception handling functionality, we can intercept and handle exceptions so that our program doesn't end abruptly (even if a user enters "Hello" when we ask for a number). At the very least, we can have our program exit gracefully instead of crashing awkwardly.

Python provides two important features to handle any unexpected error in your program and add debugging capabilities in them:

Exception handling

Assertions

Standard exceptions

Here’s the list of standard exceptions available in Python:

Python:
Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python:

Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:

Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:

Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python:

Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python: Python:

Table 15.1: *Standard exceptions available in Python*

Python assertions

The assert statement performs debugging checks. If it encounters an **assert** statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, it raises an Assertion Error and passes the message as its constructor argument, if provided.

Syntax:

```
assert Expression[, Arguments]
```

Python uses **ArgumentExpression** as an argument for the **AssertionError** when the assertion fails. The try-except statement can be used to catch and handle AssertionError exceptions. The program will terminate and a traceback will appear if they are not handled.

Example:

```
defavg(list1):


    assert is empty"
```

```
for i in list1:

        sum+=i

return


of list is

l2=[]

of list is
```

Output

```
average of list is 30.0

Traceback (most recent

line in


of list is

line

   assert is empty"
```

AssertionError:

Handling an exception

We can write programs that handle selected exceptions. If you have any code written in your program that can raise an exception, you can prevent your program from crashing by placing the suspicious code in a **try:** block. Put an **except:** statement after the try: block, followed by a block of code that handles the problem as elegantly as possible.

Syntax:

```
try:
Suspicious code
except ExceptionI:
Statements
except ExceptionII:
Statements
......................
else:
If there is no exception then execute this block.
```

Here are a few important points about the preceding syntax:

There can be more than one except statement in a try statement. The try block contains statements that throw exceptions, which is useful.

We can also provide a generic except clause that handles any exception.

We can also add a else clause that executes if no exception is raised.

Example:

```
        x = enter a small positive integer:

entered

except ValueError:

is not an integer value

test()

test()
```

Output

```
Please enter a small positive integer: 23
you entered 23
Please enter a small positive integer: 'hello'
```

```
This is not an integer value
```

Note: If the try block raises an exception at any statement, the code following that statement will not be executed. The control will shift to the except block.

The except clause with else clause

In a try statement, you can add a single else clause after all the except clauses. The else block executes if the code in the try block does not raise any exceptions.

The else block is a good place for code that does not require the try: block's protection.

Syntax:

```
try:
Suspicious code
except ExceptionI:
Statements
except ExceptionII:
Statements
......................
else:
If there is no exception then execute this block.
```

Example:

an integer:

```
except ValueError:

is not an integer



of %d is %d" %(a,a*a))

test()

test()
```

Output

```
Enter an integer: 34.4
This is not an integer value

Enter an integer: 22
Square of 22 is 484
```

With no exceptions except close

Use the except statement as follows if no exceptions are defined:

Syntax:

```
try:
Suspicious code
......................
except :
Statements
else:
If there is no exception then execute this block.
```

You will catch every exception that occurs if you use this type of statement. Since this practice will handle all exceptions in the same way, it is not a good programming practice. An except clause can specify which exceptions it will catch. There can be any number of except clauses within a try clause, but only one will be executed when an exception occurs.

Example:

```
try:

1st number:
```

2nd number:

ans=a/b

dividing first by second we get:

except:

went

Output

```
enter 1st number: 2
enter 2nd number: ab
Something went wrong
```

The except clause with multiple exceptions

A single line of code can result in different types of exceptions. For this, we can use the same except statement to handle multiple exceptions. We can use a tuple of values to specify multiple exceptions in a single **except** clause in the form of a comma-separated group enclosed in a set of parentheses.

Syntax:

```
try:
Suspicious code
......................
except(Exception1[, Exception2[,...ExceptionN]]]):
If there is any exception from the given exception list,
then execute this block
```

Example:

```
an integer


ans=float(a)


of float is
```

break

went

Output

```
enter an integer 'e'
something went wrong
enter again

enter an integer None
something went wrong

enter again

enter an integer 34
value of 34 in float is 34.0
```

Multiple except clauses can also be used to catch multiple exceptions. Following a single try statement, you can list as many except clauses as you like:

Syntax:

try:

Suspicious code

....

except ExceptionI:

Statements

except ExceptionII:

Statements

....

Example:

```
an integer:

except ValueError:

have not entered an

except KeyboardInterrupt:

have interrupted the

except EOFError:

have done end of file


entered
```

```
test()



test()


test()
```

Output

```
enter an integer:
You have interrupted the keyboard
enter an integer: a
You have not entered an integer
enter an integer: 23
You entered 23
```

Note: This method for handling multiple exceptions is better than the other one because we can have different blocks for each except clause so that we can print different statements for each one. On the other hand, we can only print a generalized statement for all the exceptions in the other method.

The Try-finally clause

You can use a finally: block with a try: block. Whether or not the try: block occurs as an exception, the finally: block is a place to add any code that must be executed.

Syntax:

```
try:
Suspicious code
......................
except ExceptionI:
Statements
finally:
This code always executes
```

Example:

```
try:

fh =

is my test file for exception

finally:
```

```
print can\'t find file or read
```

Output

```
Error: can't find file or read data
```

The same example can be written more clearly as follows:

```
fh =

is my test file for exception

print to close the

fh.close()

except

print can\'t find file or read
```

When an exception is thrown in the try block, it immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

With statement

Try blocks are commonly used to acquire resources, and finally blocks are used to release the resources. This is achieved with the help of the **with** statement.

Consider the following example with the finally clause:

```
f = None


    f =


        line = f.readline()

if len(line) ==

break

        print(line)

except IOError:
```

```
print not find file

except KeyboardInterrupt:

print You cancelled the reading from the



if f:

f.close()



print up: Closed the
```

Output

```
hello

i am learning Python

today is thursday
Cleaning up: Closed the file
```

Now, the difference here is that we are using the open function with the **with** statement; we leave the closing of the file to be done automatically with open.

Argument of an exception

When an exception occurs, it may have an associated value, i.e., the Exception's argument. The argument is usually an official message from Python describing the You can receive the argument if you specify a variable after the exception type, preceded by the **as** keyword. When we specify except Exception as e:, the variable e is not an argument but an instance of the Exception itself.

Syntax:

```
try:
You do your operations here
......................
except ExceptionType as Argument:
You can print value of Argument here...
```

In an **except** command, you can have a variable follow the name of the Exception if you are handling a single exception. On the other hand, a variable can track the exception tuple if you are trapping more than one Exception.

The value of this variable is the value of the Exception, mainly containing the Exception clause. Variables can accept one or more values. The tuple usually contains an error number, an error string, and the location of the error.

Example:

```
a number:

entered

except ValueError as Argument:

argument does not contain numbers\n ",Argument)
```

Output

```
Enter a number: abc
The argument does not contain numbers
invalid literal for int() with base 10: 'abc'
```

Raising an exception

We can forcefully raise exceptions using the **raise** keyword in Python programming when related errors occur at runtime. Exceptions can also, optionally, be populated with value to clarify why they occurred.

Syntax:

```
raise [Exception [, args [, traceback]]]
```

Arguments is a value for the exception argument, and Exception describes the type of Exception that occurred (for example, NameError). Exception arguments are optional, and NoException is returned if they are not provided.

The final argument, traceback, is optional (and seldom used in practice), and it is the traceback object used in the Exception if present.

Note: To catch an exception, the "except" clause must refer to the same Exception thrown as a class object or as a simple string.

Example:

```
a number between 25 and 40:

if

raise of

entered

except ValueError as ve:

    print(ve)

test()



test()
```

Output

```
enter a number between 25 and 40: 22
Out of range
enter a number between 25 and 40: 40
you entered 40
```

User-defined exceptions

Users can define their own Exception by creating a new class in Python. This exception class has to be derived, either directly or indirectly, from the Exception class. Most built-in exceptions are also derived from this class.

Example:

```
pass

pass

pass

any
```

```
if n

raise ValSmallError

elif n>up:

raise ValLargeError

break

except ValSmallError:

        value is very



except ValLargeError:

        value is very



guessed number between the
```

Output

```
enter any number1
```

```
this value is very small
------------------------------
enter any number56
this value is very large
------------------------------
enter any number34
this value is very large
------------------------------
enter any number22
this value is very large
------------------------------
enter any number16
you guessed number between the range
```

Conclusion

Exception handling is an essential concept in Python. In this chapter, we learned about exceptions and their handling in Python. We also looked at examples for easy understanding of the concepts.

In the next chapter, we will learn about object-oriented programming.

Key terms

Exception

ZeroDivisionError

Exception handling

Except clause

With statement

Raising an exception

CHAPTER 16

Object-Oriented Programming

We have designed our program around functions in all the programs we wrote, i.e., blocks of statements manipulating data; this is called procedure-oriented programming. Another way of organizing your program is to combine data and functionality and wrap it inside something called an object; this is called object-oriented programming. If you're writing large programs or have a problem better suited to object-oriented programming, you can use procedural programming most of the time.

Structure

We will cover the following topics in this chapter:

An overview of OOP terminology

Creating classes

Creating instance objects

Accessing attributes

Built-in class attributes

Destroying objects

Class inheritance

Types of inheritance

Super() keyword

Functions to check a relationship of two classes and instances

Overriding methods

Objectives

The main objective of this chapter is to help learners understand the concept of OOP from scratch. It also focuses on classes and objects and looks at accessing attributes and built-in class attributes. The chapter uses examples to help learners understand the concepts of inheritance. We will also look at overriding methods at the end of the chapter.

An overview of OOP terminology

Object: Instances of a data structure that are unique which are defined by its class Object is simply a collection of data (variables) and methods (functions) that act on that data.

Class: Class is a blueprint for the object or a user-defined prototype for a thing that defines a set of attributes that characterize any entity/object of the

Class variable: Each instance of a Class has access to the class variables. Class variables exist only once, and any change to a class variable will be seen by all the instances of the Class.

Object variable (data member): Every object/instance of a Class has access to the object variables. Hence, each object has its own copy of the field, and they are not related to the field with the same name in another instance. This will be easier to understand with an example.

Function overloading: A function may be assigned more than one behavior. Different operations are performed depending on the type of object or argument.

Instance variable: These are the variables defined inside a method and belonging only to the instance of a class.

Inheritance: This means that the characteristics of one Class are transferred to another class derived from it.

This is a single object of a class. Instances of a class Circle include objects that belong to the Class, for example.

Instantiation: It refers to the creation of an instance/object of a class.

Method: This is a function that is defined in a class definition.

Object: Class-defined data structures are unique instances. Objects consist of both data members (class variables and instance variables) and methods.

Operator overloading: This is the assignment of more than one function to a particular operator.

Creating classes

Create a class using the """**class**""" keyword. docstring is the first string and contains a brief description of the Class. It is recommended, but it is not compulsory.

```
'''This is a docstring. I have created a new class'''

pass
```

All the attributes of a class are defined in the class namespace. Data and functions can be attributes, and special attributes begin with double underscores A docstring for a class can be found by looking up its **__doc__** attribute. Each time we define a class, a new class object is created with the same name. The object of this class allows us to access its attributes and instantiate new objects from it.

Syntax:

```
class ClassName:
    ....
```

The self

Unlike ordinary functions, class methods have only one difference —an extra first name must be added to the beginning of the parameter list. However, you don't need to provide a value for this parameter when calling the method; Python will do it for you. **Self** refers to the object itself, and it is called self by convention.

You can give this parameter any name, but self is strongly recommended; any other name would be highly discouraged. Standardizing the name of your program has many advantages, including immediate recognition by any reader.

The __init__ method

Functions in the class that begin with double underscore are called special functions since they have special meaning. The **__init__()** function is of particular interest. This special function is called if a new object of that class is created. In **Object Oriented Programming** this type of function is called a constructor. The initialization of all variables is usually done with it.

Example:

'This is my first program using classes'

=

=

Creating instance objects

You can create instances of a class by calling its **__init__** method with the class name and the arguments it accepts.

#creating objects

Accessing attributes

You can access an object's attributes using the dot operator and access class variables by using the class name, as follows:

Now, we get the following by putting them together:

'This is my first program using classes'

=

=

#creating an object

```
ob1.display()

ob2.display()
```

Output

```
name = Rohit
age = 21
name = Divanshi
age = 20
```

The following functions are alternative methods to the regular statements for accessing attributes:

getattr(obj, name[, to access an attribute of an object

to check whether an attribute is present

Use the **setattr(obj,name,value)** method to set an attribute; the attribute will be created if it does not exist.

The **delattr(obj, name)** method deletes an attribute.

Take a look at these examples:

hasattr(ob1, Returns attribute exists

```
getattr(ob1, Returns value of 'age' attribute

setattr(ob1, # Set attribute 'age' at 17 of ob1

delattr(ob2, Delete attribute 'age' of ob2
```

Built-in class attributes

Every Python class keeps the following built-in attributes, and they can be accessed using the dot operator, like any other attribute:

Namespaces for the classes in a dictionary

If undefined, the documentation string will be used

Name of the class

The name of the module where the class resides; this attribute is "**__main__**" in interactive mode

A possible empty tuple containing the base classes in the order in which they appear in the base class list

Example:

```
'This is my first program using classes'


        self.name=name
self.age=age

=

=

#creating an object

print Student.__doc__)

print Student.__name__)

print Student.__module__)

print Student.__bases__)

print
```

Output

```
Student.__doc__: This is my first program using classes

Student.__name__: Student
Student.__module__: __main__

Student.__bases__:

Student.__dict__: 'This is my first program using Student.__init__
at Student.display at '__dict__' of 'Student' objects>, '__weakref__'
of 'Student' objects>}
```

Destroying objects (garbage collection)

Python automatically deletes unnecessary objects (built-in types or class instances) to free up memory. During garbage collection, Python periodically reclaims blocks of memory that are no longer in use. This occurs when an object's reference count reaches 0 in Python's garbage collector. The reference count of an object changes as the number of aliases pointing to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary), and its reference count decreases when it is deleted with del, reassigned, or when it goes out of scope. Python automatically collects reference counts when they reach zero.

```
x = 50        # Create object <40>
y = x         # Increase ref. count  of <40>
z = [y]       # Increase ref. count  of <40>
del x         # Decrease ref. count  of <40>
y = 100       # Decrease ref. count  of <40>
z[0] = -1     # Decrease ref. count  of <40>
```

You will normally not notice when the garbage collector destroys an orphaned instance and reclaims its space. An instance of a class may implement a special **__del__ ()** method, called a destructor, that is invoked when the instance is about to be destroyed. An instance might use this method to clean up any non-memory resources it has used.

Class inheritance

We can create a class by deriving it from an existing class by listing the parent class in parentheses after the new class name.

A child class inherits the attributes of its parent class, and you can use these attributes as if they were defined in the child class. A child class can also override data members and methods from its parent.

Syntax:

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
'Optional class documentation string'
class_suite
```

Example:

```
parentAttr = 100


print parent

defparentMethod(self):
```

```
print parent

defsetAttr(self, attr):

Parent.parentAttr = attr

defgetAttr(self):

print attribute Parent.parentAttr)

# define child class


print child

defchildMethod(self):

print child

c = Child()

c.childMethod()


c.parentMethod()
```

```
c.getAttr()
```

Output

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Types of inheritance

There are two types of inheritance:

Multilevel inheritance

Multiple inheritance

Multilevel inheritance

Like other object oriented programming languages, Python supports **multilevel** This type of inheritance occurs when a derived class inherits from another derived class. There is no maximum depth for multilevel inheritance in Python.

Example:

```
print

print

d=BabyDog()

d.eat()
```

```
d.bark()

d.weep()
```

Output

```
Eating...
Barking...
Weeping...
```

Multiple inheritance

Multiple inheritance is supported in Python, so we can inherit from multiple parents. A child class can be derived from more than one base (parent) class.

Example:

Output

```
first
second
```

third

The super () keyword

The **super()** method is most commonly used with the **__init__** function in the base class. This is usually the only place where we need to do some things in a child class and then complete the initialization in the parent.

Example:

```
are those whole eat both animals and plants')
```

```
obj=human()
```

Output

```
omnivores are those whole eat both animals and plants
humans are omnivores
```

Functions to check the relationship between two classes and instances

issubclass(sub, function: It returns True if the given subclass sub is indeed a subclass of the superclass sup.

isinstance(obj, function: It returns True if obj is an instance of class or if it is an instance of a subclass of class.

Overriding methods

When overriding a parent method, you may want to add unique or different functionality to your subclass.

Example:

```
defmyMethod(self):

print parent


defmyMethod(self):

print child

c = Child()

c.myMethod()
```

Output

```
Calling child method
```

Base overloading methods

__init__ (self [,args...]): Constructor (with any optional arguments)

Sample obj= className(args)

__del__(self): Destructor, deletes an object

Sample del obj

__repr__(self): Evaluatable string representation

Sample repr(obj)

__str__(self): Printable string representation

Sample str(obj)

__cmp__ (self, x): Object comparison

Sample cmp(obj, x)

Programs

Program to find the area of a rectangle using classes:

length of rectangle:

breadth of rectangle:

obj=rectangle(a,b)

of

print()

Output

```
Enter length of rectangle: 3
Enter breadth of rectangle: 4
```

```
Area of rectangle: 12
```

Program to create a class in which one method accepts the string and the other displays it:

```
string:

obj=print1()

obj.get()
```

Output

```
Enter string: hello
String is:
hello
```

Program to append, delete, and display the elements of a list using classes:

```
return

obj=perform(l1)

number to append:

obj.add(n)

print(obj.display())

element to be removed:

obj.remove(n)

print(obj.display())
```

Output

```
enter number to append: 25
[2, 4, 66, 56, 34, 25]
enter element to be removed: 56

[2, 4, 66, 34, 25]
```

Program to overload the '*' operator:

number

number

is

first number:

ob1=A(a)

second number:

```
ob2=A(b)

ob3=ob1*ob2
```

Output

```
Enter first number: 12
Enter second number: 5
First number 12
Second number 5
Multiplication is 60
```

Program to find the area of a rectangle using classes:

```
length:

breadth:

obj=rect(a,b)
```

is

Output

```
Enter length: 5
Enter breadth: 6
Area is: 30
```

Program to check whether a class is a subclass of another class:

```
classBase(object):

Empty Class

classDerived(Base):

Empty Class

# Driver Code

print(issubclass(Derived, Base))

print(issubclass(Base, Derived))

d =Derived()

b =Base()
```

```
# b is not an instance of Derived

print(isinstance(b, Derived))

# But d is an instance of Base

print(isinstance(d, Base))
```

Output

```
True
False
False
True
```

Program to implement inheritance:

```
# Base or Super class



classPerson(object):

name):

=name


          returnself.name
```

```
        returnFalse

# Inherited or Subclass (Note Person in bracket)

classEmployee(Person):

name, eid):

''' In Python 3.0+, "super().__init__(name)"

             also

        super(Employee,

=eid



        returnTrue



        returnself.empID

# Driver code

emp =Employee("Pawan", "EMP100")
```

```
emp.isEmployee(), emp.getID())
```

Output

```
'('Pawan', True, ''EMP100') '
```

Conclusion

In this chapter, we learned about the OOP concept in Python from scratch. We also looked at creating classes and accessing attributes, understood different inheritances, and learned how to override methods. Examples are included in the chapter help learners understand the concepts effectively.

Key terms

Function overloading

Operator overloading

Inheritance

Multilevel inheritance

Multiple inheritance

The super() keyword

Base overloading methods

Index