PROGRAMMING AND PROBLEM-SOLVING USING PYTHON

MASTER OF COMPUTER APPLICATIONS (MCA)

SEMESTER-II, PAPER-V

LESSON WRITERS

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Vasantha Rudrarnalla
Faculty, Department of CS&E
Acharya Nagarjuna University

Mrs. Appikatla Pushpalatha
Faculty, Department of CS&E
Acharya Nagarjuna University

EDITOR

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

ACADEMIC ADVISOR

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

DIRECTOR, I/c.

Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W., M.Phil., Ph.D.

Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

MCA : Programming and Problem-Solving Using Python

First Edition : 2025

No. of Copies :

© Acharya Nagarjuna University

This book is exclusively prepared for the use of students of MASTER OF COMPUTER APPLICATIONS (MCA), Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Published by:

Prof. V. VENKATESWARLU
Director, I/c
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University.

MASTER OF COMPUTER APPLICATIONS (MCA)

Semester-II, Paper-V

205MC24: Programming and Problem-Solving Using Python

SYLLABUS

UNIT I

Introduction: The Process of Computational Problem Solving, Python Programming Language.

Python Data Types: Expressions, Variables and Assignments, Strings, List, Objects and Classes, Python Standard Library

Imperative Programming: Python programs, Execution Control Structures, User-Defined Functions, Python Variables and Assignments, Parameter Passing.

UNIT II

Text Files: Strings, Formatted Output, Files, Errors and Exception Handling

Execution and Control Structures: if Statement, for Loop, Two Dimensional Lists, while Loop, More Loop Patterns, Additional lteration Control Statements

Containers and Randomness: Dictionaries, Other Built-in Container Types, Character Encoding and Strings, Module random, Set Data Type.

UNIT III

Object Oriented Programming: Fundamental Concepts, Defining a New Python Class, User- Defined Classes, Designing New Container Classes, Overloaded Operators, lnheritance, User- Defined Exceptions

Namespaces: Encapsulation in Functions, Global versus Local Namespaces, Exception Control Flow, Modules and Namespaces.

Objects and Their Use: Software Objects, Turtle Graphics, Modular Design: Modules, Top-Down Design, Python Modules

Recursion: Introduction to Recursion, Examples of Recursion, Run Time Analysis, Searching, Iteration Vs Recursion, Recursive Problem Solving, Functional Language Approach.

UNIT IV

Graphical User Interfaces: Basics of tkinter GUI Development, Event-Based tkinter Widgets, Designing GUls, OOP for GUI,

The Web and Search: The World Wide Web, Python WWW API, String Pattern Matching, Database Programming in Python

Prescribed Book:

Ljubomir Perkovic, "Introduction to Computing Using Python: An Application Development Focus", Wiley, 2012.

Reference Book:

Charles Dierbach, "Introduction to Computer Science Using Python: A Computational **Problem- Solving Focus", Wiley, 2013.**

(205MC24)

M.C.A. DEGREE EXAMINATION, MODEL QUESTION PAPER
Second Semester
205MC24: Programming and Problem-Solving Using Python

Time: 3 Hours **Max. Marks: 70**

SECTION-A

Answer Question No.1 Compulsory **2 Marks × 7 = 14 Marks**

1. a) What is a Python variable?
b) Define tuple?
c) What is formatted output?
d) Define a set in Python?
e) What is inheritance?
f) What is a module in Python?
g) What is the purpose of the sqlite3 library?

SECTION-B

Answer ONE Question from Each Unit **4 × 14 = 56 Marks**

UNIT – I

2. a) Describe the features and advantages of Python as a programming language.
b) Explain expressions, variables, and assignments in Python with suitable examples.

OR

a) Explain objects and classes in Python with examples.
b) What is the Python Standard Library? Discuss any four useful modules.

UNIT – II

3. a) Explain string manipulation and file operations in Python with examples.
b) Describe error handling and exceptions in Python with appropriate code examples

OR

a) Explain control structures in Python (if, for, while, break, continue, and pass).
b) Describe the set data type and module random in Python with examples.

UNIT – III

4. a) Explain user-defined classes and method overriding with examples.
b) Discuss namespaces in Python — global, local, and built-in — with suitable examples.

OR

a) Explain recursion and write a Python function for binary search using recursion.
b) Describe modular programming and explain how modules improve software design.

UNIT – IV

5. a) Explain the design and development of GUIs using the tkinter library.
b) Describe event-driven programming using tkinter widgets with examples.

OR

a) Explain the use of Python for Web programming and string pattern matching using regular expressions.
b) Write a short note on database programming in Python with an example of CRUD operations.

CONTENTS

LESSON- 01
INTRODUCTION

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concepts of Computational Problem Solving & Python Programming. The chapter began with understanding of The Process of Computational Problem Solving and Python Programming Language. After completing this chapter, the student will understand the complete idea about of Computational Problem Solving & Python Programming.

STRUCTURE

1.1. INTRODUCTION

Python is an ideal language for computational problem solving due to its simplicity, readability, and powerful libraries. It allows programmers to focus on developing algorithms and logic rather than dealing with complex syntax. Python's extensive standard library and third-party modules, such as NumPy for numerical computations and SciPy for scientific computing, provide robust tools for tackling a wide range of problems. The language

supports various programming paradigms, making it suitable for everything from small scripts to large-scale applications. Python's dynamic typing and interactive environment further enhance its ability to prototype and test solutions quickly, making it a preferred choice for developers and researchers alike.
The chapter first covered the Process of Computational Problem Solving, Python Programming Language.

1.2 COMPUTATIONAL PROBLEM SOLVING

Computational problem-solving is a cornerstone of modern programming, where we translate real-world problems into executable computer solutions. This chapter delves into the core aspects of computational problem solving, focusing on Python, one of the most versatile and widely used programming languages today.

1.2.1 What is Computational Problem Solving?

- **Problem Definition**: Clearly defining the problem, you want to solve.
- **Algorithm Design**: Creating a step-by-step procedure to solve the problem. **Implementation**: Writing the code to implement the algorithm.
- **Testing and Debugging**: Ensuring the code work as intended and fixing any issues.
- **Optimization**: Improving the efficiency and performance of the solution.

1.2.2 Example

The basic structure and logic for an online shopping system is explained here. By following this plan, you can implement a functional program that allows users to browse items, add them to a cart, view and manage their cart, and checkout. This approach ensures that the system is user-friendly and efficient.

Problem Definition

We want to create a simple online shopping system that allows users to:

1. Browse items available for purchase.
2. Add items to a shopping cart.
3. View the shopping cart.
4. Remove items from the shopping cart.
5. Checkout and see the total price.

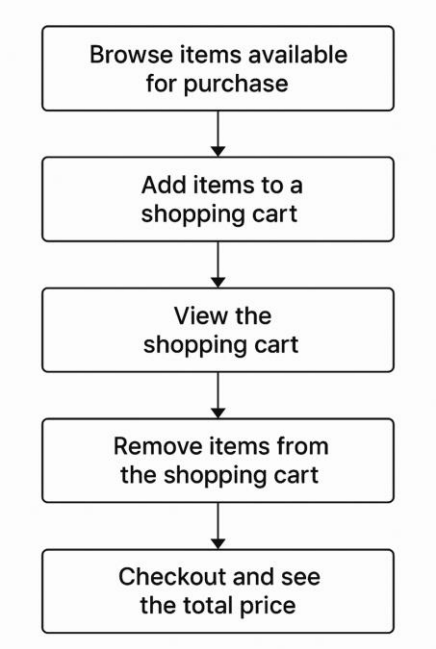


Fig 1.1 Online shopping **application**

Algorithm Design:

We'll design a basic algorithm to manage our online shopping system:

Step 1: Start

Step 2: Initialize Data

```
ITEMS = [
  (1, "Award-Winning Novel 1", 350),
  (2, "Award-Winning Novel 2", 280),
  (3, "Award-Winning Novel 3", 300),
  (4, "Award-Winning Novel 4", 275),
  (5, "Award-Winning Novel 5", 400),
  (6, "Award-Winning Novel 6", 325),
  (7, "Award-Winning Novel 7", 290),
  (8, "Award-Winning Novel 8", 310),
  (9, "Award-Winning Novel 9", 340),
  (10, "Award-Winning Novel 10", 360),
  (11, "Award-Winning Novel 11", 295),
  (12, "Award-Winning Novel 12", 315)
]

CART = empty list
```

Step 3: Display Menu Repeatedly

```
  REPEAT
  DISPLAY "------ ONLINE BOOK STORE ------"
  DISPLAY "1. View Items"
  DISPLAY "2. Add Item to Cart"
  DISPLAY "3. View Cart"
  DISPLAY "4. Remove Item from Cart"
```

```
DISPLAY "5. Checkout"
DISPLAY "6. Exit"
INPUT choice
```

Step 4: Perform Actions Based on User Choice

Case 1: View Items

```
FOR each item in ITEMS:
   DISPLAY item_number, item_name, item_price
```

Case 2: Add Item to Cart

```
DISPLAY "Enter Item Number to Add:"
INPUT item_no

IF item_no is valid THEN
   IF item already in CART THEN
      INCREMENT quantity by 1
   ELSE
      ADD (item_no, item_name, price, quantity=1) to CART
   ENDIF
   DISPLAY "Item added to cart successfully."
ELSE
   DISPLAY "Invalid item number."
ENDIF
```

Case 3: View Cart

```
IF CART is empty THEN
   DISPLAY "Your cart is empty."
ELSE
   DISPLAY "Items in your cart:"
   FOR each item in CART:
      DISPLAY item_name, quantity, (quantity * price)
   END FOR
ENDIF
```

Case 4: Remove Item from Cart

```
DISPLAY "Enter Item Number to Remove:"
INPUT item_no

IF item_no exists in CART THEN
   REMOVE item from CART
   DISPLAY "Item removed successfully."
ELSE
   DISPLAY "Item not found in cart."
ENDIF
```

Case 5: Checkout

```
IF CART is empty THEN
```

```
        DISPLAY "Your cart is empty."
      ELSE
        total = 0
        FOR each item in CART:
          total = total + (quantity * price)
        END FOR
        DISPLAY "Total Amount Payable: ₹", total
        DISPLAY "Thank you for purchasing twelve award-winning novels at a
      discounted price!"
        CLEAR CART
      ENDIF
Case 6: Exit
      DISPLAY "Exiting the system. Goodbye!"
      BREAK loop
```

Step 5: Stop

Here's a concise and clear table summarizing all the Python requirements for your Online Shopping System:

Table 1: Python Requirements for online shopping application

Category	**Python Concept / Statement**	**Purpose / Use in Program**
Data Types	int	Store item numbers, prices, and quantities
	float	Represent prices with decimals (if needed)
	str	Store item names and messages
	list	Maintain collection of items and shopping cart
	tuple	Represent each item as (id, name, price)
Control Statements	if, elif, else	Make decisions (menu options, valid input checks)
	while	Repeat the menu until user exits
	for	Traverse item lists and calculate totals
	break	Exit loop when user chooses to quit
Input / Output	input()	Take user input for menu choices and item numbers
	print()	Display menus, items, and messages
	f-string	Format output neatly (e.g., f"₹{price}")
Operators	+, *	Add totals and compute price × quantity
	==, !=, <, >	Compare menu options and check item validity
	and, or	Combine multiple conditions
Data Structures	List of Tuples	Store available novels (id, name, price)
	List (Cart)	Keep track of items added by the user
Functions (Optional)	def	Create reusable modules like view_items(), checkout()
Modules (Optional)	os	Clear screen using os.system('cls' or 'clear')
	time	Pause execution briefly using time.sleep()
Loop Control	Boolean Flag	Continue or stop the main loop (running = True/False)

1.3 PYTHON PROGRAMMING LANGUAGE

Python is a high-level, interpreted programming language known for its simplicity and readability. Created by Guido van Rossum and first released in 1991, Python emphasizes code readability with its notable use of significant whitespace. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming, making it versatile for a wide range of applications. Python's extensive standard library, dynamic typing, and ease of integration with other languages and tools have contributed to its widespread adoption in various fields, including web development, data science, artificial intelligence, scientific computing, and automation. The language's community-driven development ensures continuous improvements and the availability of numerous third-party libraries and frameworks, further enhancing its capabilities and appeal.

1.3.1. Evolution of the Python Programming Language:

Python was created by Guido van Rossum in 1980s. While in the Netherlands' National Research Institute for Mathematics and Computer Science, he created Python, an easy-to-read and use programming language. This programming language was called after the Pythons from Monty Python's Flying Circus, the founder's favorite comedians.

The first version, launched in 1991, contained few built-in data types and rudimentary capabilities. Python 1.0 was introduced in 1994 with map, lambda, and filter functions after scientists adopted it for numerical computations and data analysis. After that, adding features and releasing updated Python versions became popular. Python 1.0 introduced map, filter, and reduce methods in 1994 to process lists. Unicode support and a shorter list loop were added to Python 2.0 on October 16, 2000. Python 3.0 debuted December 3, 2008. It added print and number division support and error handling.

Python's new features benefit developers and boost performance. Python has grown in popularity and is a challenging programming language. It's in demand in machine learning, AI, data analysis, web development, and more, offering high-paying jobs. Python became the major programming language for many programmers and developers worldwide.

1.3.2 Advantages and Disadvantages of Python

Python language holds number of advantages and disadvantages which are shown in Table 3.1.

Table 1.1. Advantages and Disadvantages of Python

Advantages	Disadvantages
1. Easy to learn, read, and understand.	1. Restrictions in design
2. Versatile and open source	2. Memory inefficient
3. Improves productivity.	3. Weak mobile computing
4. Supports libraries.	4. Runtime errors
5. Huge library	5. Slow execution speed
6. Strong community	
7. Interpreted language.	

1.3.3 Companies used by Python.

This is a list of the best companies that use Python on a regular basis. Some of the names on the list provided below may surprise you which are shown in Figure 1.2.

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

Fig.1.2 Companies Use Python

1.3.4 Applications of Python

Python is emerging language and is used in wide range applications and are described detailed in below and is shown in Figure 1.3

- **Web Development**

Python's simplicity and features make it popular for web development. Python frameworks allow them to build user-friendly dynamic websites. The frameworks include Django for backend development and Flask for frontend. Because Python is easy to deploy, scalable, and efficient, most online companies utilize it as their primary technology. Top Python applications include web development, which is used across the business to build effective websites.

- **Data Science**

Python snippets help data scientists develop effective AI models. Its simplicity lets developers design complicated algorithms. Data science creates models and neural networks that learn like human brains but are faster. It helps organizations make decisions by extracting patterns from prior data. This field helps organizations invest in the future.

- **Artificial Intelligence and Machine Learning**

Data analysis and machine learning specialists can use Pandas and TensorFlow for statistical analysis, data manipulation, etc. One of the most popular programming languages is Python. The language of AI and ML is Python. Python has helped this field with its many libraries and community support. Python use will rise as artificial intelligence and machine learning evolve significantly.

- **Game Development**

Python developers can use Pygame to create 2D and 3D games. Pirates of the Caribbean, Battlefield 2, and others are popular Python games. Pygame is a Python library for making fun games. Since the gaming industry is growing, these types of development have become more popular. This package makes game development easy, so you can try building some simple games.

1.3.5 Features of Python

There are several characteristics that distinguish the Python programming language from others the main reason is its features and described below and shown in Figure 1.4.

❖ **Popularity**

Python is the fourth most popular and fastest-growing programming language, according to the Stack Overflow Developer Survey 2022. Businesses including Google, Instagram, Netflix, and Spotify use it.

❖ **Interpretation**

Python is an interpreted language; unlike compilers, which need the creation of machine code from the source code before it can be executed, Python passes directly to the interpreter, simplifying and speeding up the execution process.

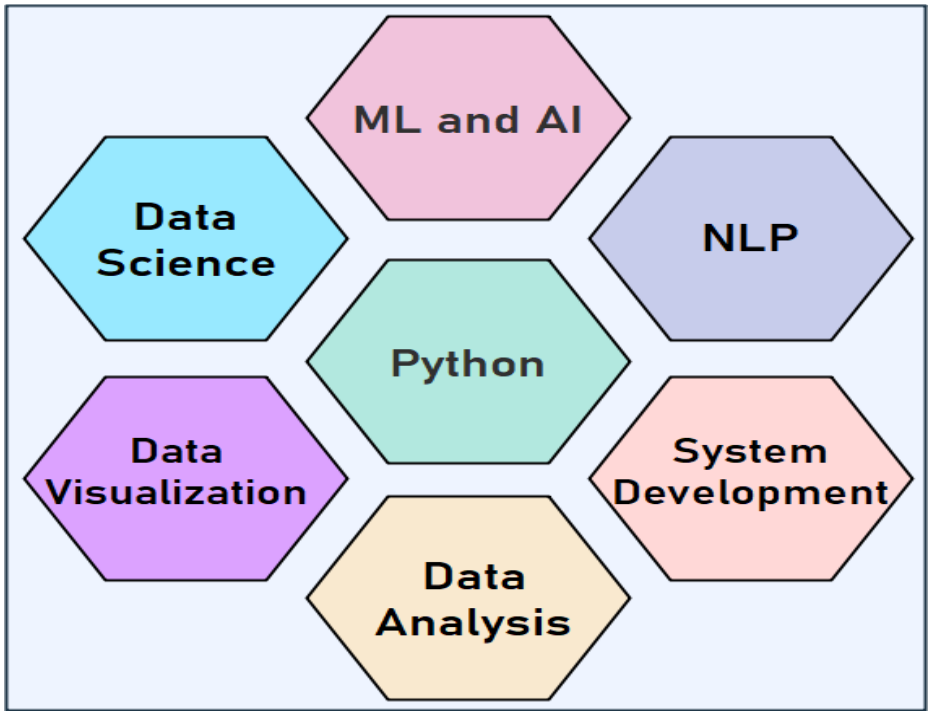


Fig 1.3 Applications of Python

- **Open Source**

The fact that Python is a free language created under an open-source license certified by OSI is among its strongest features.

- **Portability**

Major trouble comes in transferring a code from one platform to another without making blunders in the command. Python programming language, being a portable code can easily be transferred without making any errors.

- **Simplicity**

The only programming language that is similar to English is Python. It's so simple to read and comprehend. The Python programming language utilizes fewer keywords than C++ or Java. As a result, developers everywhere now favor the Python language above all others.

- **A high-level language**

Compared to several other programming languages, Python is more similar to human languages. As a result, its core features, such memory management and architecture, are unimportant to programmers.

- **An object-oriented language**

Python is a programming language that supports a variety of programming styles, including structured and functional programming, in addition to the standard object-oriented programming paradigm.

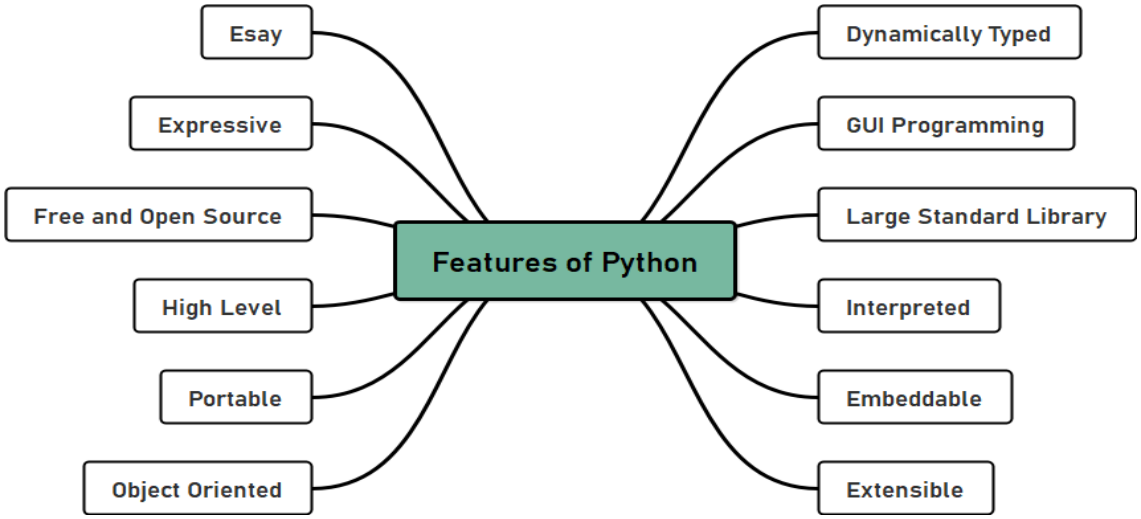


Fig 1.4 Features of Python

1.4 HOW TO WRITE AND RUN A PYTHON SCRIPT

Python programmers need to be familiar with all possible script and code execution scenarios. There is no other way to confirm that the code is functioning as intended. The Python programs are executed by the Python interpreter. A Python interpreter is a software that functions as a bridge between computer hardware and Python programs.

Here, we'll go over the various methods for executing Python programs. The simple program is created using notepad is shown in Figure 1.5.

- The operating system command-line or terminal.
- The Python interactive mode.
- The IDE

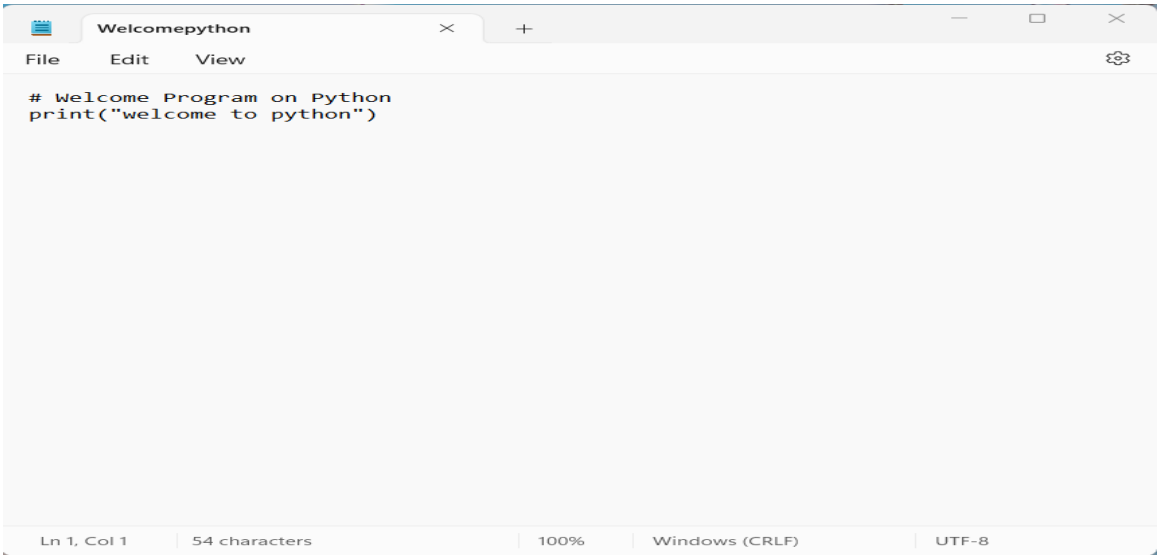


Fig 1.5. A sample python program created and saved on notepad.

1.4.1 The operating system command-line or Terminal

Since the Python shell loses all the code we write when the session is closed, we can run the Python code using a command line. Thus, using plain text files to write Python code is an excellent idea. The text file needs to be saved with the.py suffix.

The Python print statement is written and saved in the working directory as welcomepython.py. We are going to use the command line to execute this file now.

To run a Python script, open a command line. To run the file, we must input the file name and then Python. Once you press the enter key, the result will look like this if there are no errors in the file and is shown in Figure 1.6.

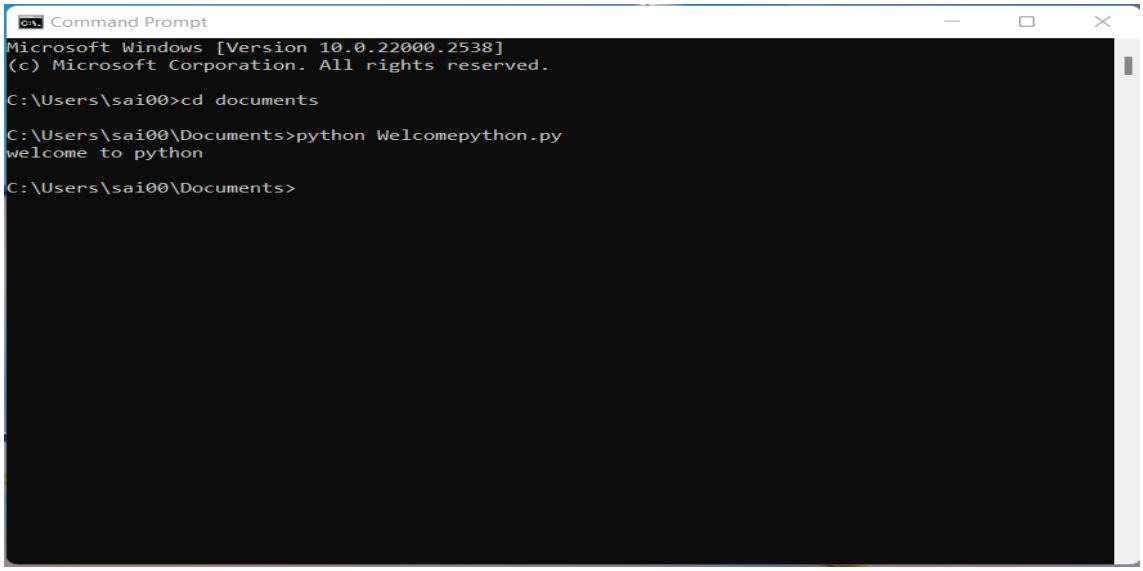


Fig 1.6 Command Line to Run python program.

1.4.2 The Python Program Create and Run on Interactive Shell

We can utilize the Python interactive session to write and execute the Python code. To launch an interactive Python session, simply select a command-line or terminal from the Start menu, type python, and hit the Enter key. It is a fantastic development tool because it enables us to review every line of code. However, all of our written code will be lost when the session ends. To exit the interactive shell, type quit(), exit(), or press the Ctrl+Z key.

This is an illustration of how to use an interactive shell to run Python code is shown in Figure 1.7.

1.4.3 How to Run Python Program on IDLE?

Windows and Mac Python installations contain Python IDLE. If you use Linux, you should be able to utilize your package manager to locate and download Python IDLE. After installation, Python IDLE can be used as a file editor or as an interactive interpreter.

Python is included with DLE, an Integrated Development and Learning Environment. A complete development environment for authoring, debugging, and testing code is offered by IDLE.

You must do the following actions to launch a Python application on IDLE:

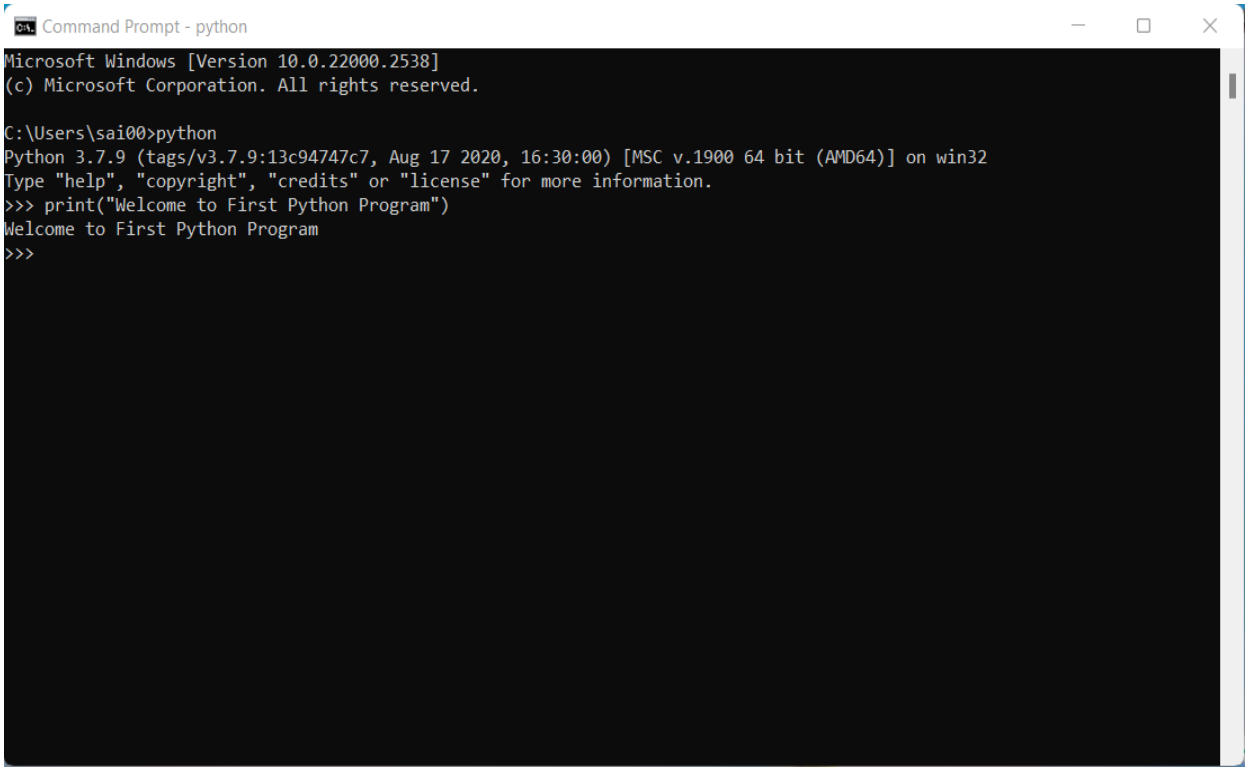


Fig 1.7. Create and Run python program in Interactive Shell.

Step 1: Launch the Python IDLE first. Since IDLE operates in the shell by default, this window will appear on your screen.

Step 2: Using the IDLE, we can create and run Python scripts and see the results directly on the screen, and is shown in Figure 1.8.

Step 3: Open a new file by selecting File → New File in order to run a whole Python program on IDLE.

Step 4: Write your Python program in the "New File" that appears when the previous step is completed shown in Figure 3.8.

Step5: Save your file in this step. It is saved here under the filename welcomepython.py.

Step 6: Click RUN → Run Module to start the process shown in Figure 3.9.

Step 7: The IDLE Shell will display the output.

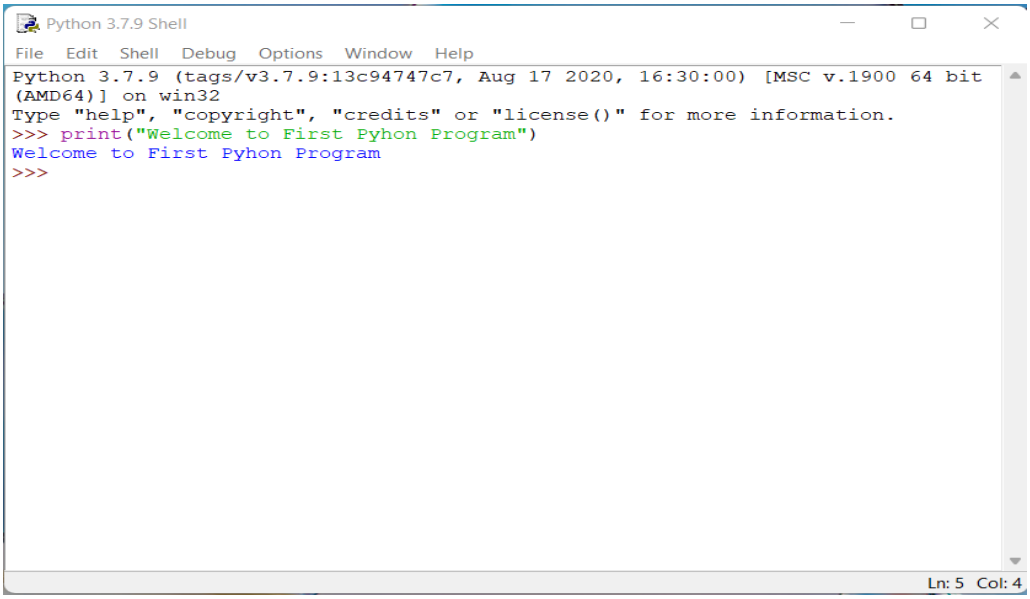


Fig 1.8. Create Python scripts in IDE.

You can run Python applications with ease by following the instructions in the description above, which include utilizing text editors, IDEs, or the command line. You can become more adept at executing Python code and utilizing its features to take on a variety of tasks and challenges with practice and experimentation.

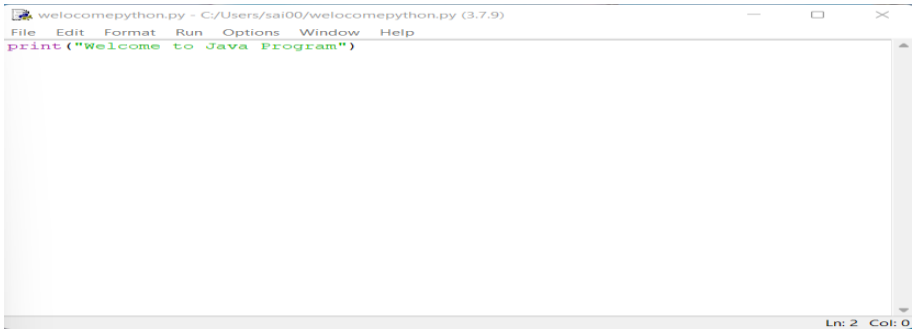


Fig 1.9. Create and Save Python Scripts in new file of IDE.

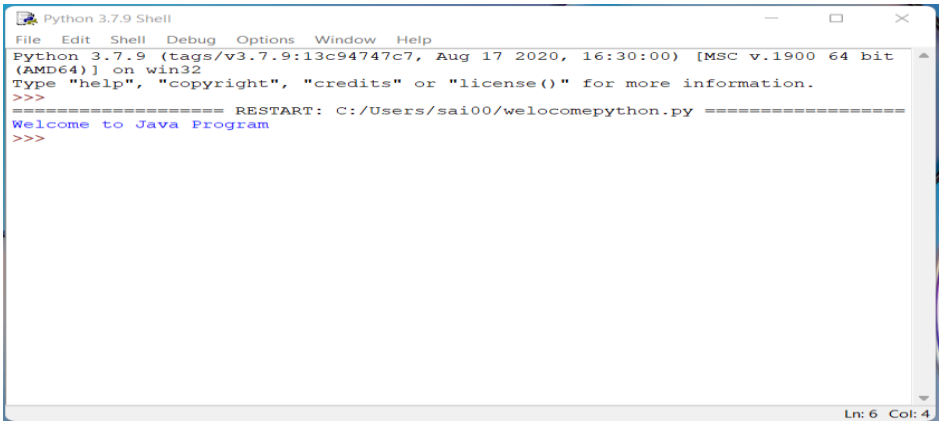


Fig 1.10. Run Python Scripts in IDE.

One of the most important skills for anyone studying or using Python is the ability to run programs. Knowing how to run Python code is essential, regardless of your level of experience—whether you're a novice learning the fundamentals of the language or an expert in creating complex apps.

1.5. SUMMARY

Computational problem solving is the process of using logical and systematic methods to design algorithms that can be executed by a computer to achieve a desired outcome. It involves understanding the problem, developing a step-by-step solution (algorithm), and implementing it using a programming language. Python, a versatile and beginner-friendly language, is widely used for this purpose because of its simple syntax, readability, and powerful libraries. It enables programmers to focus more on problem logic rather than complex syntax, making it ideal for tasks such as data analysis, automation, web development, artificial intelligence, and scientific computing.

1.6 TECHNICAL TERMS

- Computational Problem
- Python
- Command Line
- IDE
- Programming

1.7 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about computational Problem using python
2. Describe about applications of Python

Short Notes:

1. Write about Evolution of python
2. Discuss about how to run python program

1.8 SUGGESTED READINGS

1. "Python Crash Course" by Eric Matthes
2. "Learning Python" by Mark Lutz
3. "Python Programming: An Introduction to Computer Science" by John Zelle
4. "Think Python: How to Think Like a Computer Scientist" by Allen B. Downey
5. "Python for Data Analysis" by Wes McKinney

Dr. U Surya Kameswari

LESSON- 02

PYTHON DATA TYPES

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the Python data types. The chapter began with an understanding of Expressions, Variables and Assignments, Strings, List, Objects and Classes, Python Standard Library and so on. After completing this chapter, the student will understand the python data types in detail with suitable examples.

STRUCTURE

2.1. INTRODUCTION

Python is a high-level computer language that is interpreted and object-oriented. Its semantics change over time. Its high-level built-in data structures, along with dynamic typing and dynamic binding, make it a great choice for Rapid Application Development. Python's grammar is simple and easy to learn. It focuses on readability, which lowers the cost of maintaining programs.

Python lets you use modules and packages, which makes it easier to break up programs into smaller pieces and reuse code. For all major systems, you can get the Python interpreter and the large standard library for free in source or binary form, and you can share them with anyone else.

This chapter will cover the major basic concept of python programming including what is python, history of python, advantages and disadvantages of python, applications of python etc.

2.2 PYTHON DATA TYPES

In Python, data types define the kind of value a variable can hold and determine what operations can be performed on that value. They form the foundation of any Python program, allowing developers to store, manipulate, and process data efficiently. Python provides several built-in data types such as numbers, strings, lists, tuples, sets, and dictionaries, each designed for specific purposes. These data types make Python both flexible and powerful, enabling programmers to handle a wide range of computational tasks — from simple arithmetic to complex data processing — with ease and clarity.

Table 2.1. Python Data Types

Data Type	Category	Description
Numbers	int, float, complex	numeric values
String	Str	sequence of characters
Segquence	List, tuple	sequence of items
Mapping	Dict	data in key-value pair
Set	Set	collection of unique items
Bolean	True, False	Return Boolean result true or false

Numbers: Based on their names, these are the types of data that store numbers: integer, float, and complex. It can be either an int or a long int.

There are three numbers in Python:

Example:

```
A =  20   # Assing 20 to A

B = 4.67 # Assign 4.67 to B

print(A) # prints 20 on screen
print(B) # prints 4.65 on screen
```

Output:

```
20
4.65
```

Strings : These are in Python are groups of characters that are kept in memory together, like an array of characters. Either a single quote or two double quotes are used to show these characters.

Example:

```
S =  “  Happy “ #prints Happy to console
```

```
print S[1]  # prints first character to console
print S + " Morning "  # concatenates Morning to Happy and prints on console
```

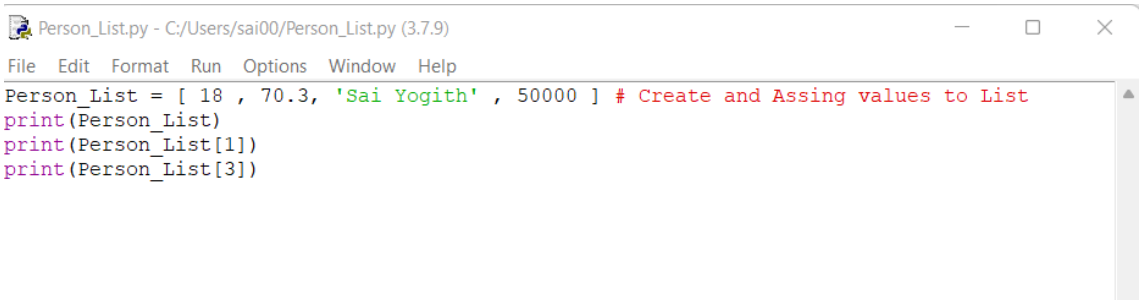
Output:

```
Happy
H
Happy Morning
```

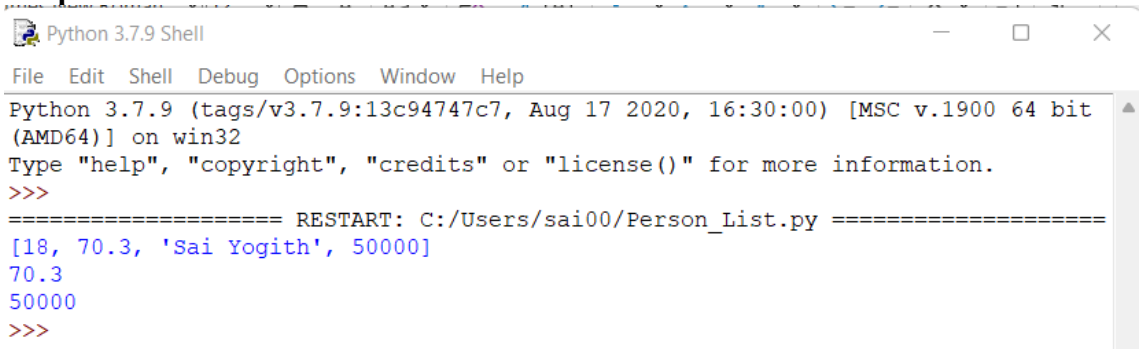
❖ Python List

In Python, a list is a sorted list of things separated by commas (,) and enclosed in square brackets ([]). If you access a Python list using the slicing operator [], you can change the value of any item in it. A list in Python is like a collection. The main difference is that an array is a collection of items that are all of the same type, while a list is a collection of items that can be of different kinds. The Python list can be changed.

Example:



Output:

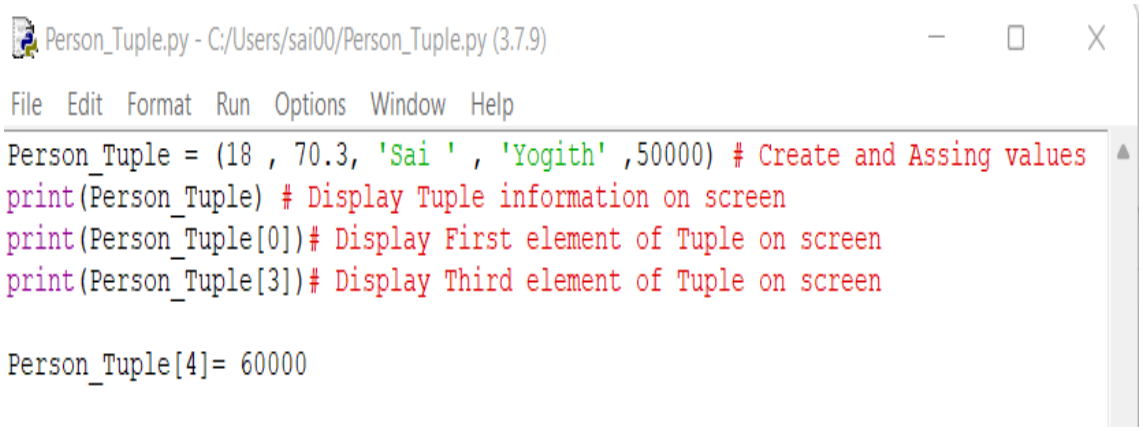


The code above shows that Person_List has items that are numbers, floats, strings, and long ints. The result shows that the whole Person_List was shown first.

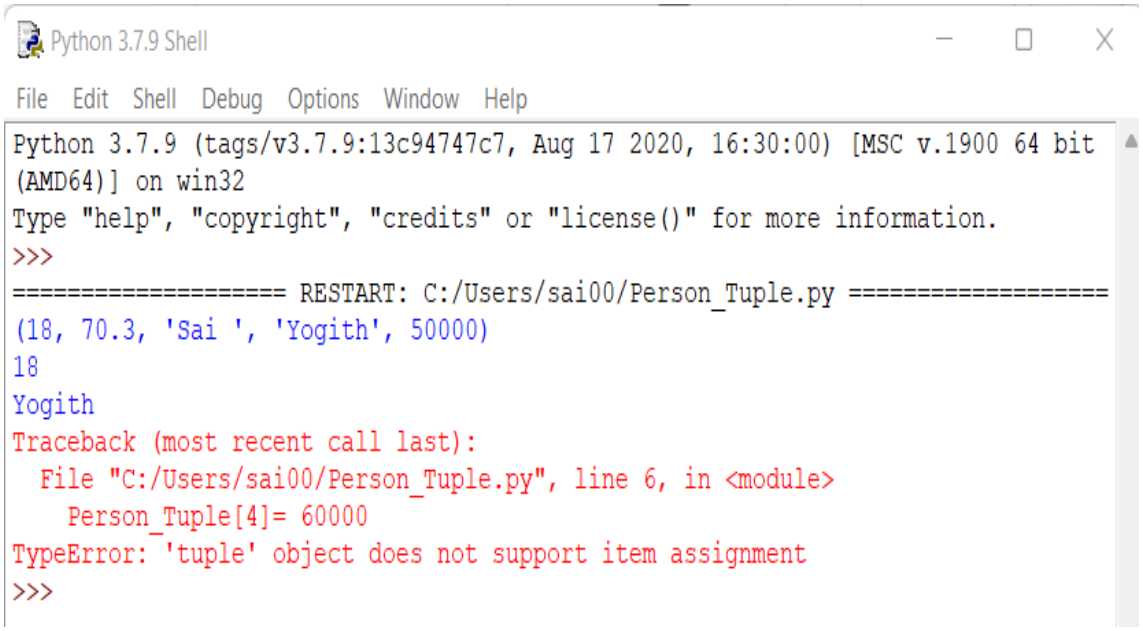
❖ Python Tuple

Python tuples are the same as Python lists. The only difference is that Python tuples are immutable, which means that you can access the things in them but not change their values. Besides being able to change, another big difference between tuples and lists is that lists are defined inside square braces [], while tuples are defined inside parentheses ().

Example:



Output:



The code above shows that the items in Person_Tuple are integers, floats, strings, long integers, and strings. The result shows the full Person_Tuple as the first item. After that The first and fourth items were printed.

But at the end of the last line, an error is made because the fourth member of the tuple is being changed. Based on the finding, we can say that tuple items can't be changed, but List data types can.

Python Dictionary

A sorted list of key-value pairs is called a dictionary in Python. The dictionary's entries are key-value pairs separated by commas. The value can always be retrieved if we know the key, but the opposite is not true. Python dictionaries are therefore designed for data retrieval. Python dictionaries are defined inside curly braces ({}), and the slicing operator ([ ]) is used to access and assign values.

Example:

week_dic.py - C:/Users/sai00/week_dic.py (3.7.9)

File Edit Format Run Options Window Help

```
# Creation of a Dictionary named Week

Week={ 'Monday'    : 'Mon',
       'Tuesday'   : 'Tue',
       'Wednesday' : 'Wed',
       'Thursday'  : 'Thu' }

print (Week)              # Prints Dictionary Week

print (Week['Thursday']) # Prints Dictionary Week third item by key

print (Week['Thu'])      # Prints Dictionary Week third item by value
```

Output:

Python 3.7.9 Shell

File Edit Shell Debug Options Window Help

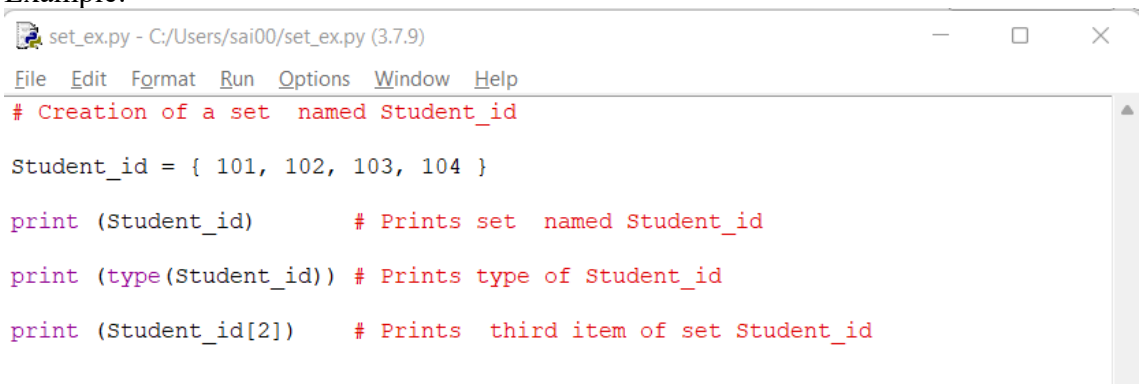
```
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/week_dic.py ==========
{'Monday': 'Mon', 'Tuesday': 'Tue', 'Wednesday': 'Wed', 'Thursday': 'Thu'}
Thu
Traceback (most recent call last):
  File "C:/Users/sai00/week_dic.py", line 12, in <module>
    print (Week['Thu'])      # Prints Dictionary Week third item by value
KeyError: 'Thu'
>>>
```

e have created a dictionary called week in the example above. In this case, the keys are Monday, Tuesday, Wednesday, and Thursday, and the values are Monday, Tuesday, Wednesday, and Thursday. To get the appropriate value, we employ keys. not the other way around, though. Here, we've used the week dictionary's keys to obtain the data. Capital_city['Thursday'] retrieves its corresponding value, Thu, since 'Thursday' is the key. But since 'Thu' is the value assigned to the 'Thursday' key, capital_city['Thu'] raises an error.

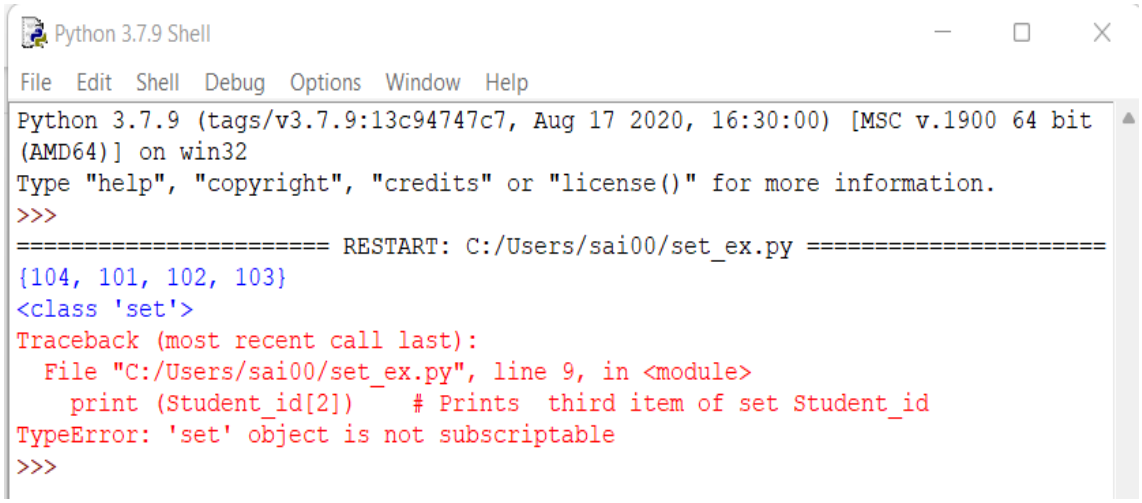
❖ **Python Set Data Type:**

A set is an arbitrary grouping of distinct objects. Values inside braces {} and separated by commas define a set.

Example:



Output:



Here, four integer values have been added to a set called student_id. As sets are collections that are not ordered, indexing is meaningless. The entire set is shown first. Afterwards, trying to access the element of the set using the slicing operator [] does not work. Similar to the output accessing the third item with the error message generated by the index.

Boolean: The datatype which returns only 2 values either TURE or FALSE.

Example:

A = 50 ;

Output:

>>> A = = 40

>>> FALSE

2.2.1. Expressions, Variables, and Assignments in Python

In Python, expressions are combinations of values and operators that evaluate to a value. They form the basic building blocks of any Python program, allowing for operations such as arithmetic calculations, string manipulations, and logical comparisons. Variables are used to store these values, acting as named references to data that can be easily accessed and manipulated throughout a program.

- **Expressions**

In Python, expressions are combinations of values, variables, operators, and function calls that are evaluated to produce a new value. They can perform a variety of operations, including arithmetic calculations, string manipulations, and logical comparisons. Expressions are fundamental components of a Python program.

```
# Examples of expressions
a = 5 + 3  # Arithmetic expression
b = "Hello" + " " + "World"  # String concatenation expression
c = 10 > 5  # Comparison expression
```

- **Variables**

Variables in Python are symbolic names that reference or point to objects or values stored in memory. They are used to store data that can be modified and accessed throughout a program. Python uses dynamic typing, which means that you do not need to declare the type of a variable explicitly; the type is inferred from the value assigned to it.

```
# Examples of variables
x = 10  # Integer variable
y = 3.14  # Float variable
name = "Alice"  # String variable
```

- **Assignments**

Assignment statements are used to assign values to variables. The assignment operator = is used to perform assignments. The variable on the left side of the = operator is assigned the value on the right side.

```
# Examples of assignments
x = 5  # Assigning an integer value to variable x
y = x + 2  # Assigning the result of an expression to variable y
message = "Hello, World!"  # Assigning a string value to variable message
```

- **Python Operators**

Operators are unique symbols or keywords in Python that perform operations on values and variables. They form the foundation of expressions, which are used to work with data and carry out calculations. Python has a number of operators, each having a distinct function. The Python programming language supports the following types of operators:

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators

- **Python Arithmetic Operators**

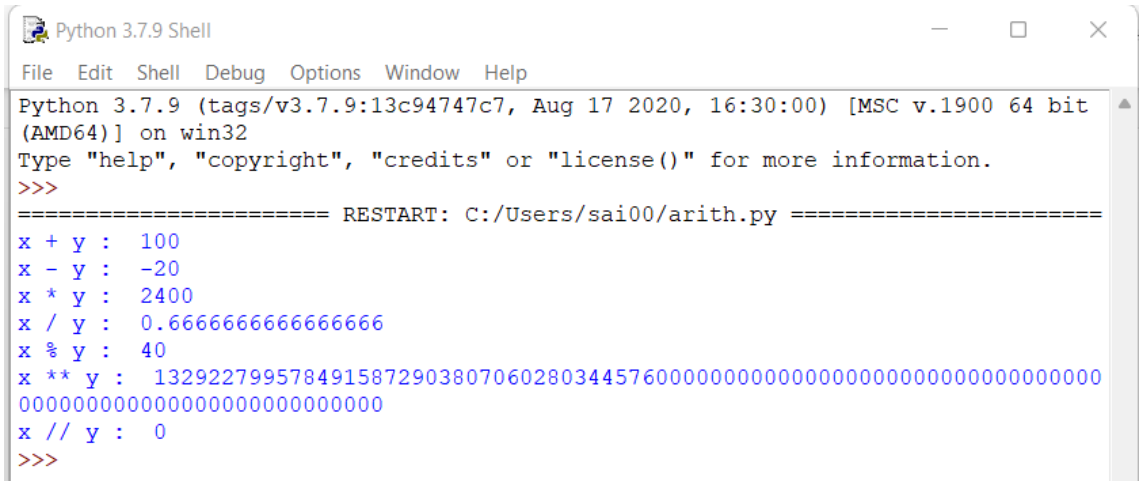
Common mathematical operations in addition modules are addition, subtraction, multiplication, and division; additional arithmetic operations include exponential and floor divisions are shown in Table 2.2. Expressions, variables, and integers are supported by all.

Table 2.2. Arithmetic Operations in Python

Operator	Description	Python Expression
+	Addition	**x + y**
-	Subtraction	**x - y**
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponent	x ** y
//	Floor Division	x // y

Example:

Output:



Once the two variables "x" and "y" are defined, this code does a number of mathematical operations, including floor division, modulus, addition, subtraction, multiplication, and division, and reports the results.

- **Python Comparison Operators**

Python comparison operators are required in order to compare two values.They produce a Boolean value (True or False) based on the comparison. The comparison operators in python is shown in Table 2.3.

Table 2.3. Comparison Operations in Python

Operator	Description	Python Expression
= =	Equal	**x == y**
! =	Not Equal	**x != y**
>	Greater Than	x > y
<	Less Than	x < y
> =	Greater Than or Equal	x >= y
< =	Less Than or Equal	x <= y

Example:

comparision.py - C:/Users/sai00/comparision.py (3.7.9)

File Edit Format Run Options Window Help

```
x = 20
y = 10
# Equal
print ("x == y : ", x == y)
# Not Equal
print ("x != y : ", x != y)
# Greater Than
print ("x > y : ", x > y)
# Less Than
print ("x < y : ", x < y)
# Greater Than or Equal
print ("x >= y : ", x >= y)
# Less Than or Equal
print ("x <= y : ", x <= y)
```

Ln: 15 Col: 0

Output:

Python 3.7.9 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/comparision.py ==========
x == y :  False
x != y :  True
x > y :  True
x < y :  False
x >= y :  True
x <= y :  False
>>>
```

Ln: 11 Col: 4

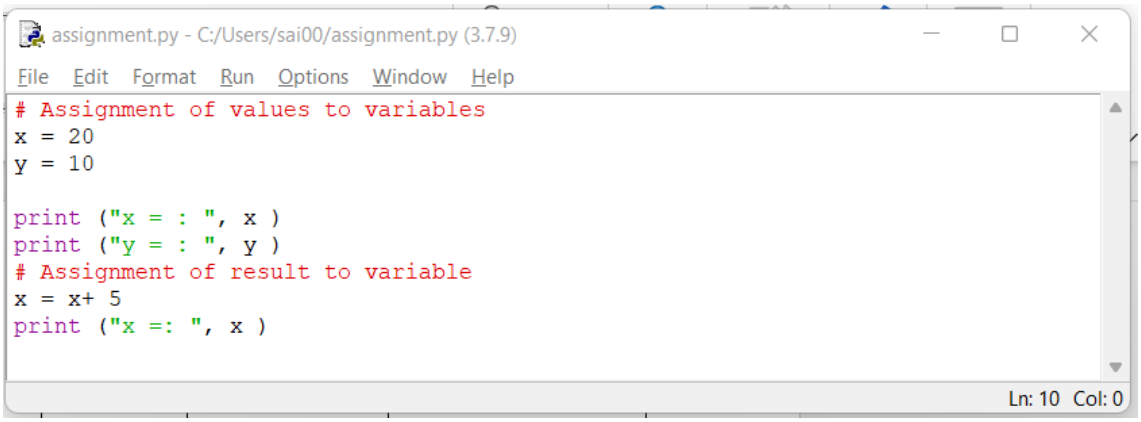
- **Python Assignment Operators**

To assign values to Python, utilize the assignment operators. The simplest assignment operator is the single equal symbol (=). The variable on the operator's left side is given the value on the operator's right side. The different approaches to use assignment operator in python is shown in Table 2.4.

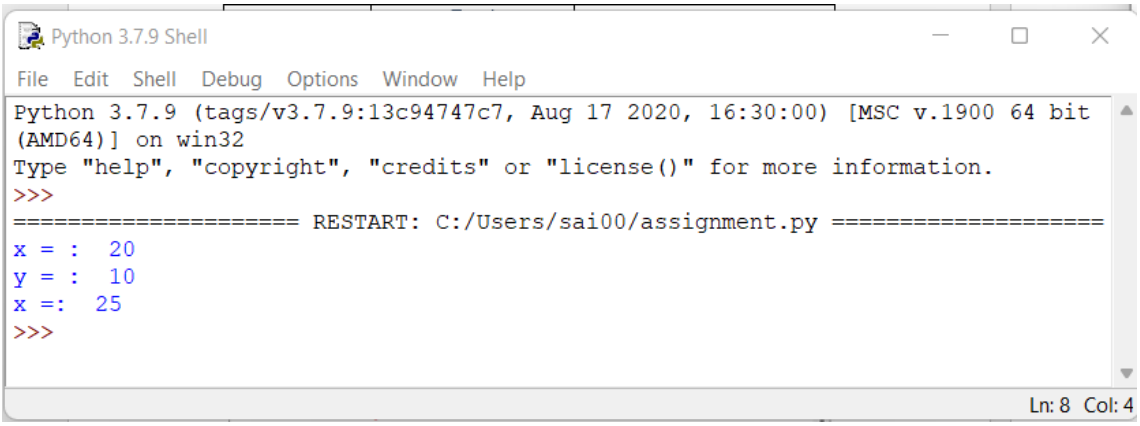
Table 2.4 Assignment Operations in Python

Operator	Description	Python Expression
=	Equal	**x = y** **x=x+5** **x= 2 * x + 4 * 5 + 8**

Example:



Output:



Above displays the assignment operators in Python. First, 'x' and 'y' have values of 20 and 10, respectively. Afterwards, x=25 is the output of applying expression x+5 to x.

- **Python Bitwise Operators :** Bitwise operators in Python carry out actions on discrete binary integer bits.They operate on each bit location logically while working with integer binary representations.Many bitwise operations, including AND (&), OR (|), NOT (), XOR (), left shift (), and right shift (>>), are included in Python.

- **Python Logical Operators**
  Boolean expressions are composed, and their truth values are evaluated using logical operators in Python. They are necessary for controlling the program's execution flow and for creating conditional statements. The three fundamental logical operators in Python are AND, OR, and NOT.

  - **Python Membership Operators**
  To determine whether a particular value appears in a series or not, one can utilize Python membership operators. They simplify the process of figuring out which elements belong in many types of data structures, including sets, tuples, lists, and strings. The is and is not operators are the two main membership operators in Python.

- **Combining Expressions, Variables, and Assignments**

  In Python, you often use expressions, variables, and assignments together to perform various operations and store results. This combination is essential for creating dynamic and interactive programs.

```
# Example combining expressions, variables, and assignments

a = 10

b = 20

sum_result = a + b  # Assigning the result of an expression to a variable

print(sum_result)  # Output: 30

greeting = "Hello"

name = "Bob"

personal_greeting = greeting + ", " + name + "!"  # Combining string expressions

print(personal_greeting)  # Output: Hello, Bob!
```

Expressions, variables, and assignments are the core concepts of programming in Python. Expressions allow you to perform operations and produce values. Variables provide a way to store and reference these values. Assignments enable you to set and update the values of variables. Understanding how to use these elements effectively is fundamental to writing Python programs.

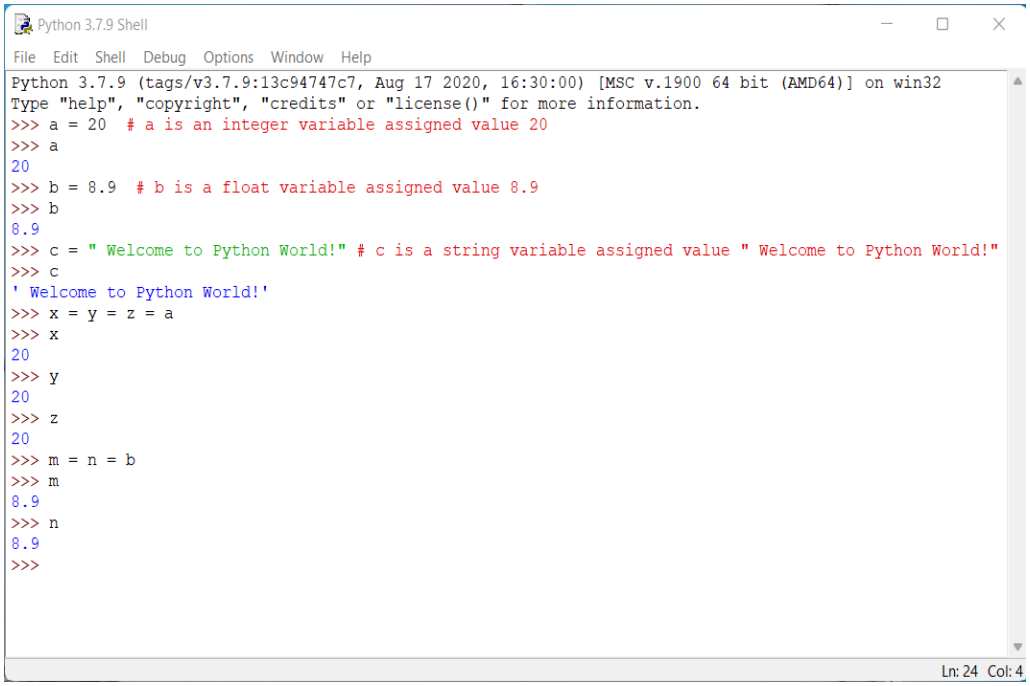


Fig 2.1 Example of Variable Declaration and Assignment

Python Keywords

Each language has words and rules that make sense when put together in a sentence. Also, the computer language Python has a set of predefined words that are called Keywords. You can't use these words anywhere else in Python because they have special meanings. Keywords set the rules for how the code is written. That word can't be used as a variable, function, or symbol name. The only words in Python that are written in capital letters are True and False. Python 3.11 has 35 keywords and are'shown in Figure 2.2.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Fig 2.2 Python Kewords

2.3. STRING IN PYTHON

In Python, strings are sequences of characters enclosed in either single quotes (' ') or double quotes (" "). They are immutable, meaning once a string is created, it cannot be changed. Python provides a variety of built-in methods for string manipulation, such as len() to get the length of a string, lower() and upper() to change the case, and split() to divide a string into a list of substrings. Strings support indexing and slicing, allowing access to individual

characters or segments of the string. Concatenation of strings can be done using the + operator, and the in keyword is used to check for the presence of a substring within a string.Python also supports triple quotes (''' ''' or """ """) for multi-line strings.

Key Points

1. **Creation**: Strings can be created using single quotes, double quotes, or triple quotes for multi-line strings.
2. **Concatenation**: Strings can be concatenated using the + operator or various string formatting methods such as f-strings, format(), and %-formatting.
3. **Accessing Characters**: Individual characters in a string can be accessed using indexing, and substrings can be extracted using slicing.
4. **Common Methods**: Python provides a rich set of methods for string manipulation, including:
    - upper(), lower(): Convert to uppercase or lowercase.
    - startswith(), endswith(): Check if a string starts or ends with a given substring.
    - find(): Locate the position of a substring.
    - replace(): Replace occurrences of a substring with another substring.
    - split(): Split a string into a list of substrings.
    - join(): Join a list of strings into a single string.
5. **String Formatting**: Formatting strings can be done using f-strings (for Python 3.6+), the format() method, and %-formatting.

Example :

Here's a quick example that incorporates several of these concepts:

```
# String creation

greeting = "Hello"

name = "Alice"

# Concatenation and formatting

full_greeting = f"{greeting}, {name}!"  # Using f-string

print(full_greeting)  # Output: Hello, Alice!

# String methods

upper_greeting = full_greeting.upper()

print(upper_greeting)  # Output: HELLO, ALICE!

# Slicing

first_word = full_greeting[:5]

print(first_word)  # Output: Hello
```

```
# Splitting and joining

words = full_greeting.split(", ")

joined_words = " - ".join(words)

print(joined_words)  # Output: Hello - Alice!
```

2.4. LIST IN PYTHON

lists in Python are dynamic, versatile, and powerful data structures that allow you to store and manipulate an ordered collection of items. Understanding how to create, access, modify, and manipulate lists is fundamental for effective programming in Python. Here's a summary of the key points:

Key Points

Creation: Lists can be created using square brackets [] and can store elements of different data types.

```
my_list = [1, 2, 3, 'apple', 'banana']
```

Accessing Elements: You can access elements using indexing and slicing.

```
first_element = my_list[0]  # 1

sub_list = my_list[1:3]     # [2, 3]
```

Modifying Lists: Lists are mutable, so you can change their content.

```
my_list[0] = 10

my_list.append('cherry')

my_list.insert(2, 'orange')
```

Removing Elements: You can remove elements using various methods.

```
my_list.remove('banana')

popped_element = my_list.pop()

del my_list[1]
```

List Operations: Lists support several operations such as concatenation, repetition, and membership testing.

```
new_list = my_list + [4, 5, 6]

repeated_list = my_list * 2
```

```
is_in_list = 'apple' in my_list
```

List Methods: Python provides a range of methods for list manipulation.

```
my_list.sort()

my_list.reverse()

index_of_apple = my_list.index('apple')

count_of_10 = my_list.count(10)
```

Iterating Over Lists: You can iterate over the elements of a list using loops.

```
for item in my_list:
    print(item)
```

List Comprehensions: List comprehensions provide a concise way to create lists.

```
squares = [x**2 for x in range(10)]
```

Here's a quick example that incorporates several of these concepts:

```
# List creation
fruits = ['apple', 'banana', 'cherry']

# Accessing elements
first_fruit = fruits[0]  # apple
last_two_fruits = fruits[-2:]  # ['banana', 'cherry']

# Modifying lists
fruits[1] = 'blueberry'
fruits.append('date')
fruits.insert(1, 'avocado')

# Removing elements
fruits.remove('cherry')
popped_fruit = fruits.pop()  # date
del fruits[0]
```

```
# List operations

more_fruits = ['elderberry', 'fig']

all_fruits = fruits + more_fruits

doubled_fruits = fruits * 2

is_elderberry_in_list = 'elderberry' in all_fruits

# List methods

all_fruits.sort()

all_fruits.reverse()

index_of_fig = all_fruits.index('fig')

count_of_avocado = all_fruits.count('avocado')

# Iterating over lists

for fruit in all_fruits:

    print(fruit)

# List comprehensions

lengths_of_fruits = [len(fruit) for fruit in all_fruits]
```

By mastering these list operations and methods, you can efficiently manage collections of data in Python, enhancing your programming capabilities. If you have any specific questions or need further assistance, feel free to ask!

2.5. OBJECT AND CLASS IN PYTHON

In Python, objects and classes are fundamental concepts of object-oriented programming (OOP). Here's a brief overview of these concepts:

Classes

A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.

Defining a Class

```
class Dog:

    # Class attribute
```

```
    species = "Canis familiaris"

    # Initializer / Instance attributes

    def __init__(self, name, age):

        self.name = name

        self.age = age

    # Instance method

    def description(self):

        return f"{self.name} is {self.age} years old"

    # Another instance method

    def speak(self, sound):

        return f"{self.name} says {sound}"
```

Objects

An object is an instance of a class. It has the attributes and methods defined in the class.

Creating an Object

```
my_dog = Dog("Buddy", 3)
```

Accessing Attributes and Methods

```
print(my_dog.name)  # Output: Buddy

print(my_dog.age)   # Output: 3

print(my_dog.species)  # Output: Canis familiaris

print(my_dog.description())  # Output: Buddy is 3 years old

print(my_dog.speak("Woof woof"))  # Output: Buddy says Woof woof
```

Example

Here's a summary example that includes class definition, object creation, and method invocation:

```
        class Dog:

            species = "Canis familiaris"
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def description(self):
        return f"{self.name} is {self.age} years old"
    def speak(self, sound):
        return f"{self.name} says {sound}"
# Creating an instance of the Dog class
my_dog = Dog("Buddy", 3)
# Accessing attributes and methods
print(my_dog.name)  # Output: Buddy
print(my_dog.description())  # Output: Buddy is 3 years old
print(my_dog.speak("Woof woof"))  # Output: Buddy says Woof woof]
```

Understanding these concepts will enable you to leverage the power of object-oriented programming in Python, making your code more modular, reusable, and easier to maintain. If you have any specific questions or need further details, feel free to ask!

2.6 THE PYTHON STANDARD LIBRARY

The Python Standard Library is a collection of modules and packages that come with Python, providing a wide range of functionality for various tasks such as file I/O, system operations, data manipulation, and networking. Here is a brief overview of some key modules and their uses:

Key Modules:

sys: Provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

```
import sys
print(sys.version)  # Output the Python version
sys.exit()  # Exit the program
```

os: Provides a way to use operating system-dependent functionality like reading or writing to the file system.

```
import os
```

```
print(os.name)  # Output the name of the operating system
os.mkdir('new_directory')  # Create a new directory
```

datetime: Supplies classes for manipulating dates and times.

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Format the current date and time
```

math: Provides access to mathematical functions.

```
import math
print(math.sqrt(16))  # Output: 4.0
print(math.pi)  # Output: 3.141592653589793
```

random: Implements pseudo-random number generators for various distributions.

```
import random
print(random.randint(1, 10))  # Output a random integer between 1 and 10
```

The Python Standard Library provides tools and utilities for many common programming tasks, enabling developers to write efficient and effective code without needing to install external libraries. If you have any specific questions or need further details, feel free to ask!

2.7 SUMMARY

Python supports a wide range of data types and programming constructs that make it both powerful and easy to use. Expressions in Python combine values, variables, and operators to produce new results, while variables are used to store and reference data dynamically without explicit type declarations. Strings represent sequences of characters and support numerous operations such as concatenation, slicing, and formatting. Lists allow storage of ordered collections of items that can be easily modified, whereas objects and classes provide the foundation for object-oriented programming, enabling modular and reusable code through encapsulation and inheritance. The Python Standard Library further enhances the language by offering a vast collection of built-in modules and functions for tasks like file handling, math operations, data manipulation, and system interaction — making Python a comprehensive and efficient language for solving computational problems.

2.8 TECHNICAL TERMS

- Expressions
- Operators
- Operands
- Evaluation
- Variables
- Objects
- Classes

2.9 SELF ASSESSMENT QUESTIONS

Essay Questions:

1. Explain the concept of expressions, variables, and assignments in Python with suitable examples.
2. Illustrate the various data types in Python and describe their key characteristics.
3. Describe the string data type in Python and discuss common string operations and methods.
4. Explain lists in Python, highlighting their features, indexing, and common list operations.
5. Discuss the concepts of objects and classes in Python and explain how object-oriented principles are implemented.
6. Write a detailed note on the Python Standard Library and its importance in program development.

Short Notes:

1. Write short notes on Python Expressions with examples.
2. Explain the process of variable creation and assignment in Python.
3. Write about string slicing and concatenation in Python with examples.
4. Discuss any four list methods with suitable examples.
5. Write a short note on object-oriented features in Python.
6. List some commonly used modules from the Python Standard Library and their uses.

2.10 SUGGESTED READINGS

1. Steven cooper – Data Science from Scratch, Kindle edition.
2. Reemathareja – Python Programming using problem solving approach, Oxford Publication
3. Think Python: How to Think like a Computer Scientist
4. Brown, A.- Mastering Python Modules. Publisher.
5. Ljubomir Perkovic, “Introduction to Computing Using Python: An Application Development Focus”, Wiley, 2012.
6. Charles Dierbach, “Introduction to Computer Science Using Python: A Computational Problem-Solving Focus”, Wiley, 2013.

Dr. U Surya Kameswari

LESSON- 03

IMPRATIVE PROGRAMMING

AIMS AND OBJECTIVES

The primary goal of this chapter is to introduce the concept of **imperative programming** and its implementation in Python. Students will learn about program structure, control flow, variable assignments, functions, and parameter passing mechanisms. After completing this chapter, students will be able to write, execute, and manage Python programs effectively using imperative programming concepts.

STRUCTURE

3.1 INTRODUCTION

Imperative programming is one of the most common programming paradigms that focuses on describing *how* a program operates.

It uses statements that change a program's state, executing instructions step by step. Python, being a multi-paradigm language, supports imperative programming efficiently.

This approach helps programmers control the flow of execution using variables, functions, and control structures such as conditionals and loops.

3.2 IMPERATIVE PROGRAMMING CONCEPTS

Imperative programming is based on commands that change the state of the program. The program is a sequence of statements that tell the computer what to do, one after another. Examples of imperative languages include C, Java, and Python.
Characteristics of Imperative Programming:

- Uses variables to store and modify program data.

- Includes control structures like loops and conditionals.
- Programs are executed sequentially.
- Focuses on how to perform tasks rather than what to perform.

3.3 PYTHON PROGRAMS

A Python program is a collection of statements written to perform a specific task.
It may include variables, expressions, functions, and control structures.
Python programs can be written using any text editor and executed using the Python interpreter.

Example:

```
print('Welcome to Python Programming')

x = 10

y = 20

sum = x + y

print('Sum:', sum)
```

Output:

```
Welcome to Python Programming

Sum: 30
```

3.4 EXECUTION CONTROL STRUCTURES

Control structures determine the flow of execution in a program.
They allow the programmer to decide which statements to execute and in what order.

Types of Control Structures:

- Sequential: Executes statements in order.
- Selection: Uses conditional statements like *if*, *elif*, and *else*.
- Iteration: Repeats execution using loops like *for* and *while*.

if STATEMENT

The **if statement** tests a condition; if True, its indented block executes.

Syntax

```
if condition:
    statement_block
```

Example

```
score = 90
```

```
if score >= 80:
    print("Excellent performance")
```

Flow Description

1. Evaluate condition.
2. If True → execute block.
3. If False → skip block.

Practice

Write a program to check whether a number entered by the user is positive.

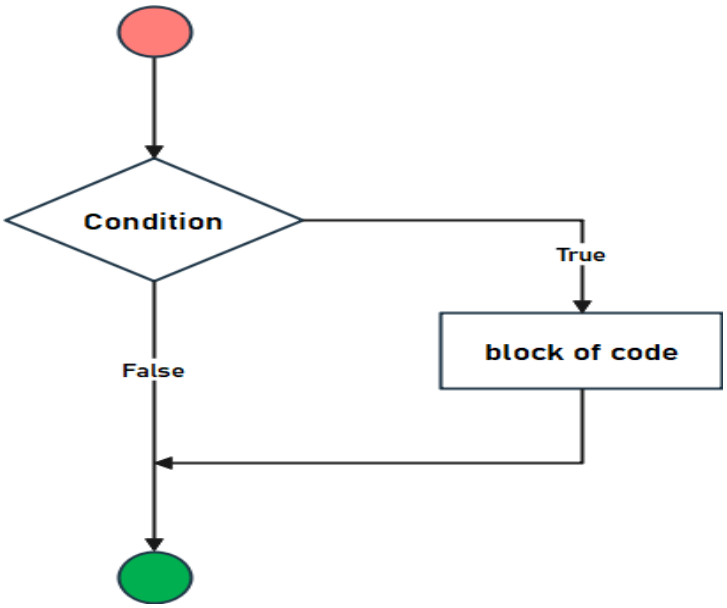


Fig 3.1 Flow Chart of if Statement

if–else STATEMENT

Used when two mutually exclusive paths exist.

Syntax

```
if condition:
    block_true
else:
    block_false
```

Example

```
num = int(input("Enter a number: "))
```

```
if num % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

Flow Description:

If condition fails, the *else* block executes.

Practice

Write a program that accepts a temperature in

if–elif–else STATEMENT

When multiple conditions must be checked in order, Python uses elif.

```
if condition1:
    ...
elif condition2:
    ...
else:
    ...
```

Example -Grade Evaluation

```
marks = int(input("Enter marks: "))
if marks >= 75:
    print("Distinction")
elif marks >= 60:
    print("First Class")
elif marks >= 40:
    print("Pass")
else:
    print("Fail")
```

Output

Enter marks: 82

First Class

Practice Task

Modify the program to print “Outstanding” for marks ≥ 90.

For Loop Statement

It is possible to iterate over a series of elements in Python by using the for loop, which is one of the looping instructions contained inside the language. There are a variety of objects that can be iterated, including a list, a tuple, a text, and any other object.

Syntax:

for **variable** in **sequence**:

Code block to be executed

The preceding syntax,

- variable is a temporary variable that stores the value of each element in the sequence during each iteration of the loop
- The code block that comes after the for statement is carried out many times for each individual element that is included in the sequence.

Example:

for **i** in **10**:

Code block to be executed

Total 10 time block will be repeated

The flowchart to represent for loop statement in python is shown in Figure 5.2

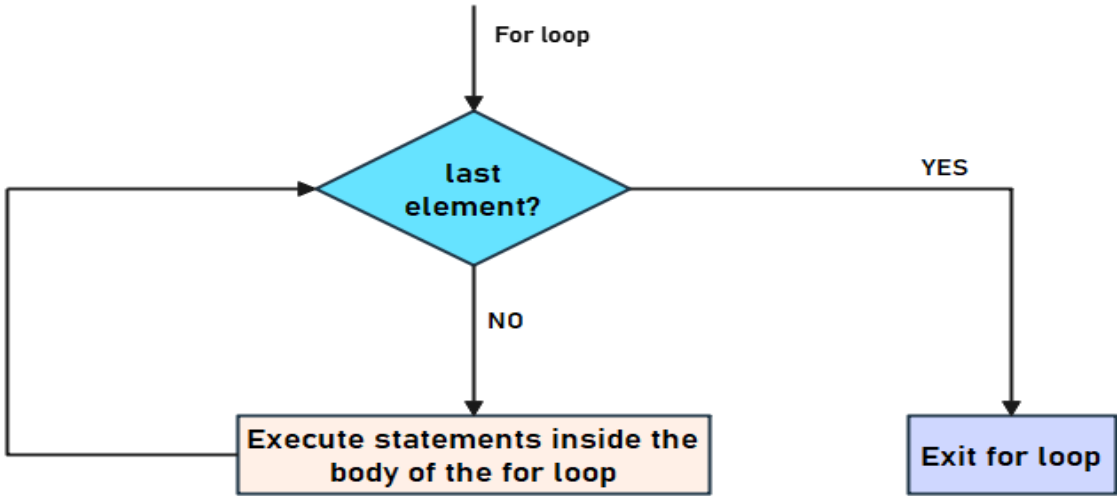
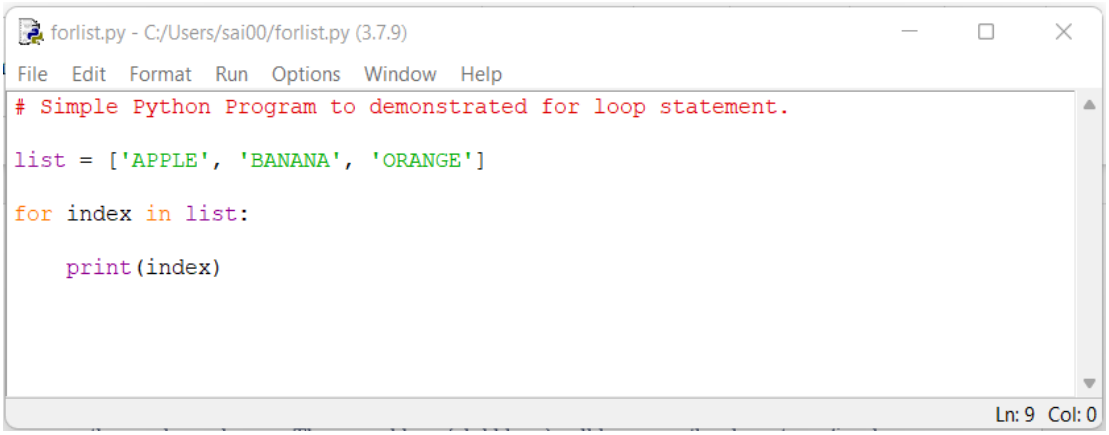


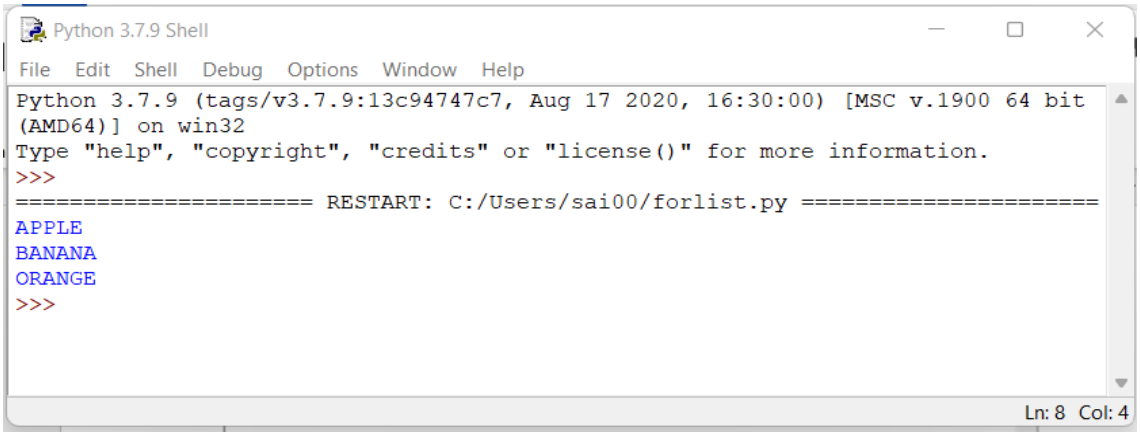
Fig 3.5. Flowchart of For-loop Statement

Example :

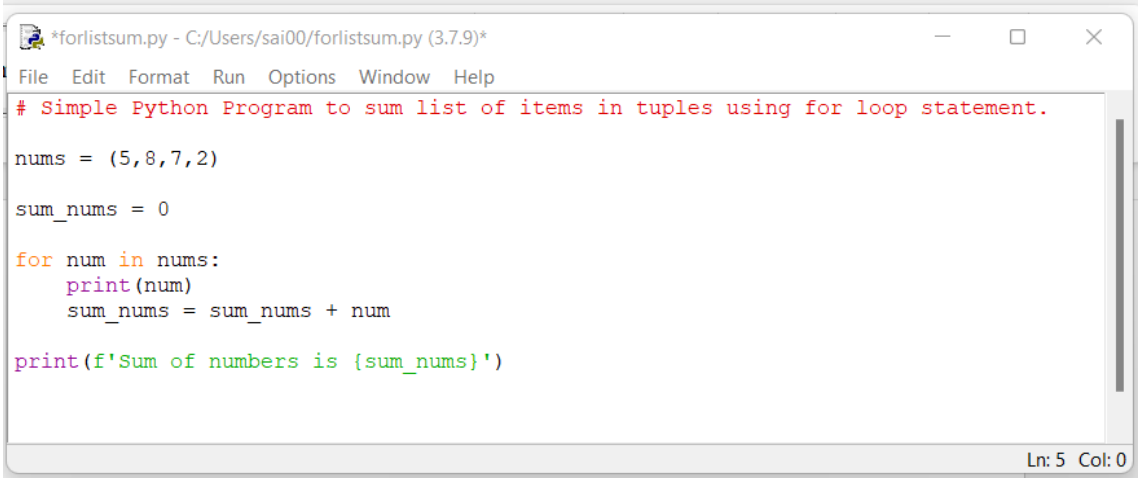


The code that you see above has a for loop that prints each element of the 'list' list on a new line after iterating over each entry in the list. The output is shown on the next page.

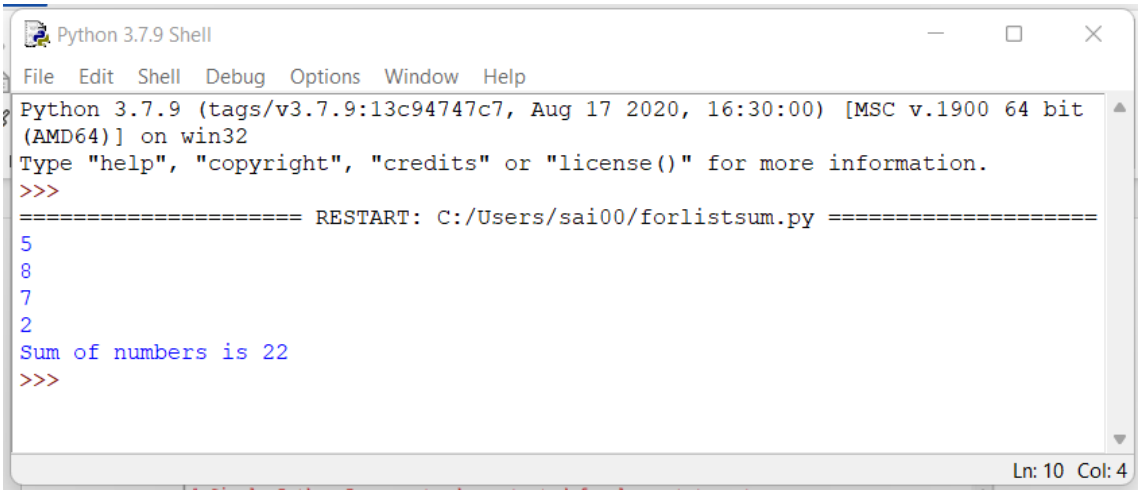
Output:



Example 2:



Output:



Using the code that was just presented, the for loop will iterate over each element in the tuple that is referred to as 'num' and then display it on a new line. In addition, the sum of each

number was computed, the result was saved in the "sum_nums" variable, and the sum value was eventually printed out. In the run tuple, a sequence of distinct integers (5,8,7,2) is used, and the result is "the sum of the numbers is 22"

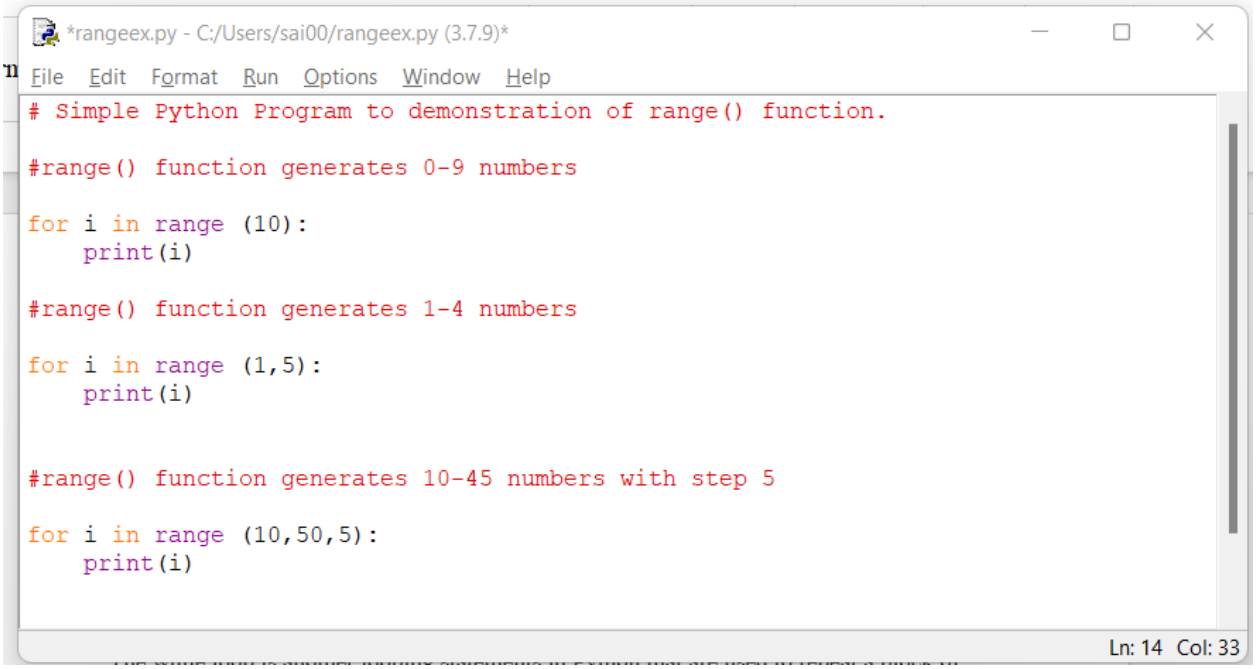
According to Python, a range object is a sequence of numbers that cannot be changed. When using a for loop, it is helpful to keep track of the number of times a block is repeated.

You can use the range() method in the following ways:

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

Every one of the three arguments must be an integer. The value of the [start] parameter is always set to zero, unless an alternative number is provided. The only parameter that is required for the function described above is stop. It is one less than the stop parameter that the last integer in the series is. In the intervals between, the [step] value, which is set to 1 by default, is used to increment the numbers.

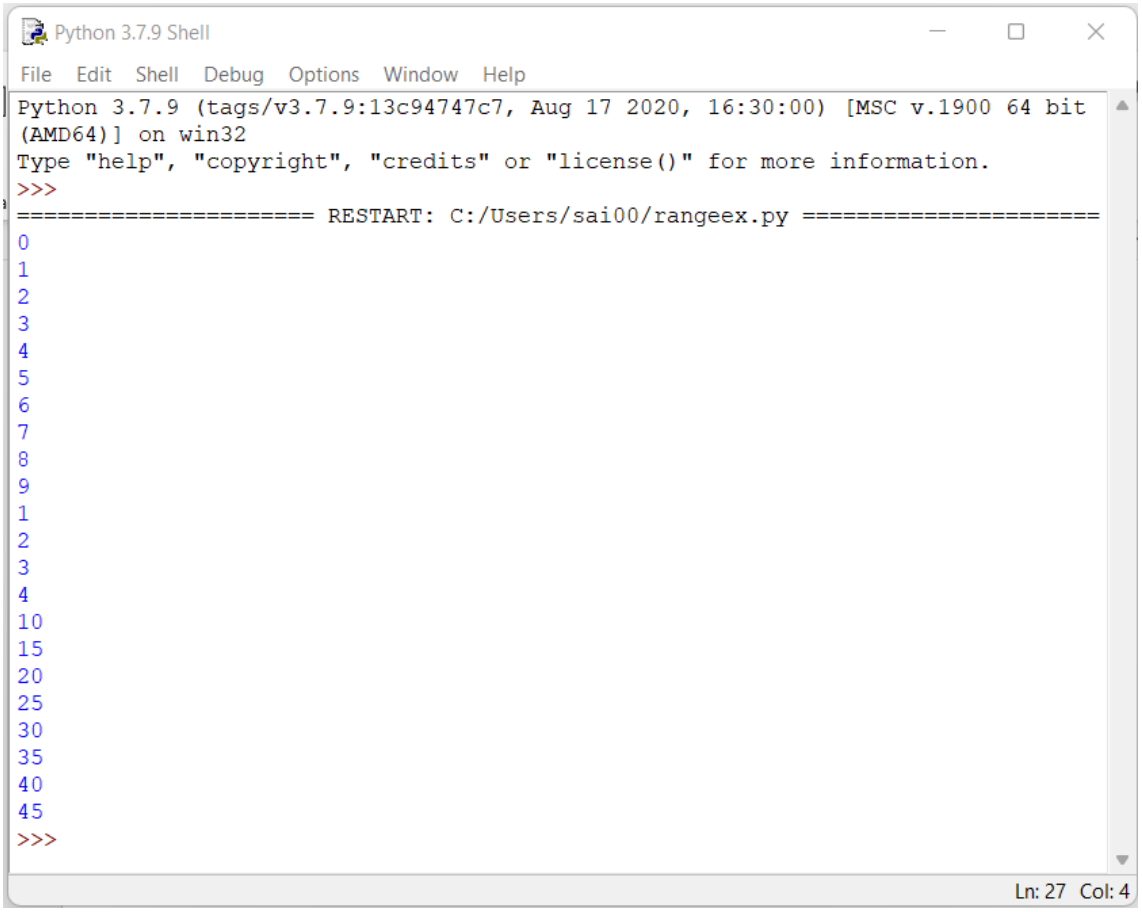
Example:



The range() method was used instead of a for loop statement in the Python code above. Three for loop statements in all, each printing a distinct range of numbers according on the inputs passed to the range () function.

When the first "10" value was entered into range (10) it produced numbers starting at 0 and ending with 10-1, or 9. A for-loop statement is then given range(1,5), and values are printed starting at 1 and ending at end 5-1, or 4. Lastly, range(10,50,5) is sent to the for-loop expression, which outputs values starting at 10 and ending at 50-4, or 45, because step=5.

Output:



While Loop Statement

Another Python looping expression used to repeat a block of code until a predetermined condition is met is the while loop.

Syntax:

The syntax of the while loop in Python is given below.

while condition:

    # Code block to be executed

A boolean expression called condition in this syntax is evaluated at the beginning of each loop iteration. The while statement is followed by a code block that is periodically run until the condition evaluates to False.

The flowchart to represent while loop statement in python is shown in Figure 5.3

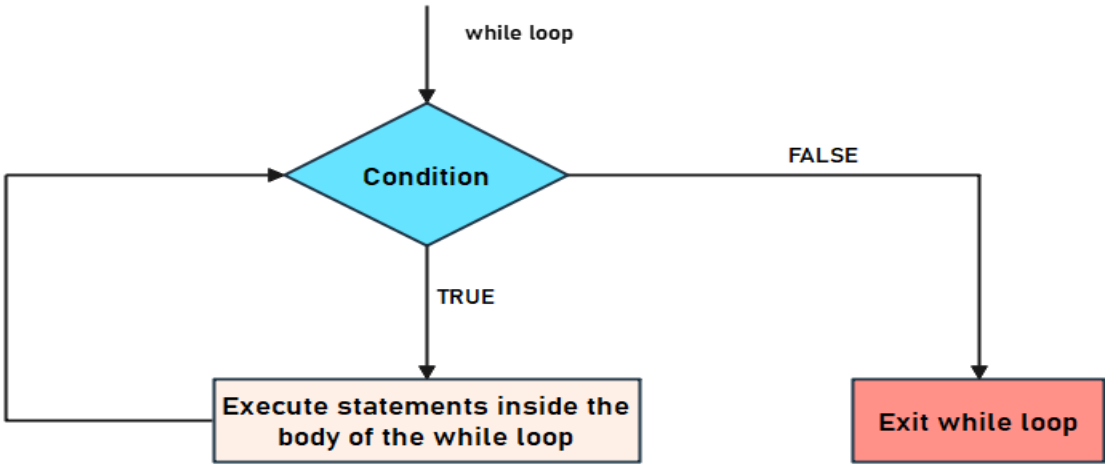


Fig 3.6 Flowchart of While-loop Statement

Example:

whilelistsum.py - C:/Users/sai00/whilelistsum.py (3.7.9)

File Edit Format Run Options Window Help

```
# Simple Python Program to demonstration of while loop statement.


sum = 0

n = int ( input( "Enter n value: ") );

while ( sum < n):
    print(sum)
    sum = sum+1

print(f'Sum of numbers is {sum}')
```

Ln: 8 Col: 17

Output:

Python 3.7.9 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/whilelistsum.py ==========
Enter n value: 5
0
1
2
3
4
Sum of numbers is 5
>>>
```

Ln: 12 Col: 4

The code block is repeated here by the while loop until the sum variable is less than 5. As we can see in the output, the sum variable is increased by 1 at each iteration, and the current value of the sum is printed on a new line.

Nested Loop Statement

A loop inside another loop is known as a nested loop in Python. When we wish to loop over a series of components with several degrees of nesting, we utilize it.

Syntax:

for variable in sequence:

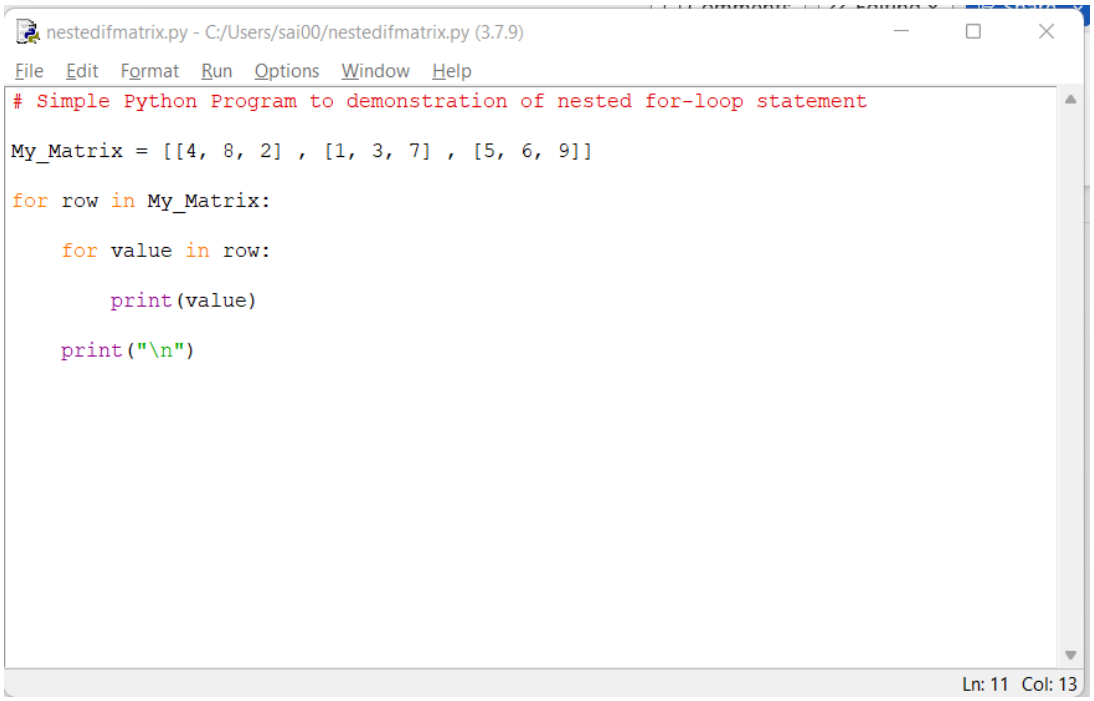
    for i_variable in i_sequence:

        # Code block to be executed

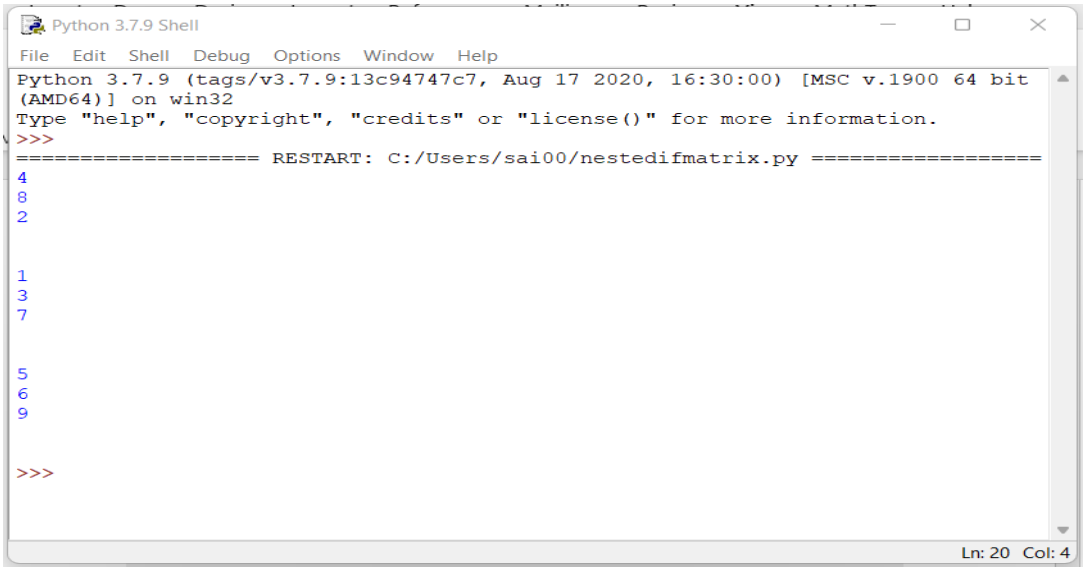
Variable, as used in this syntax, is a temporary variable that, for each iteration of the outer loop, stores the value of each element in the sequence. Every time the inner loop iterates, the value of every element in the i_sequence is stored in the i_variable, a temporary variable. Every element in the inner sequence and every element in the outer sequence is subjected to several executions of the code block that follows the inner for statement.

Example :

The code given below uses the Nested Loop.



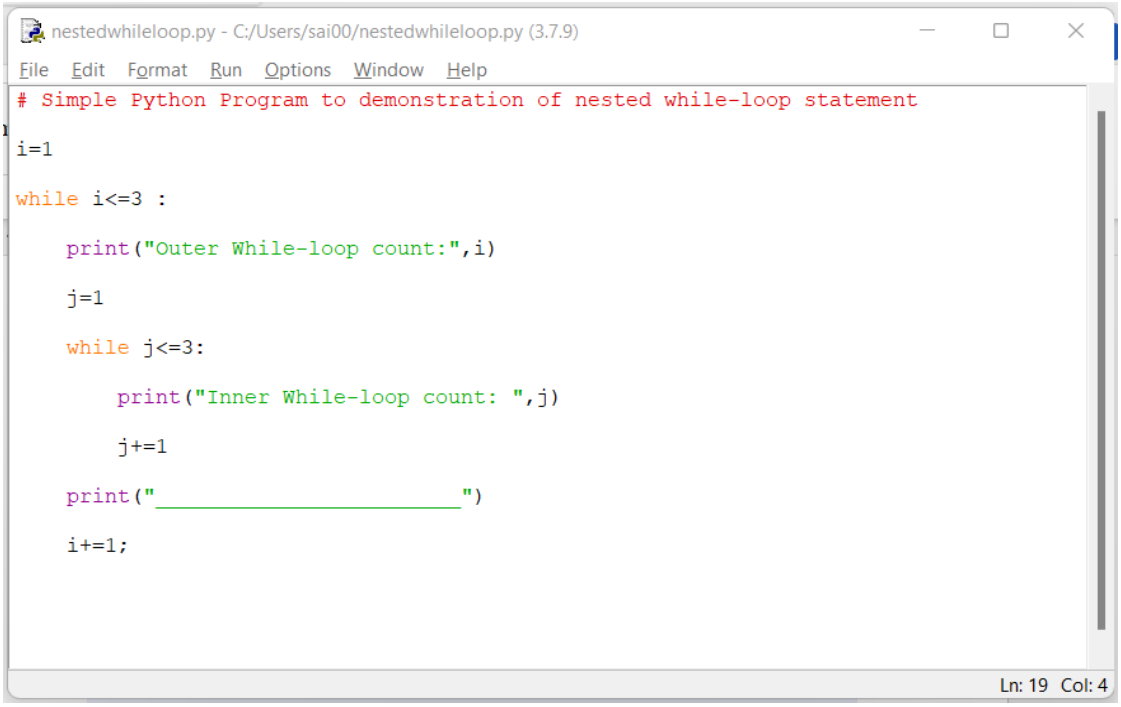
Output:



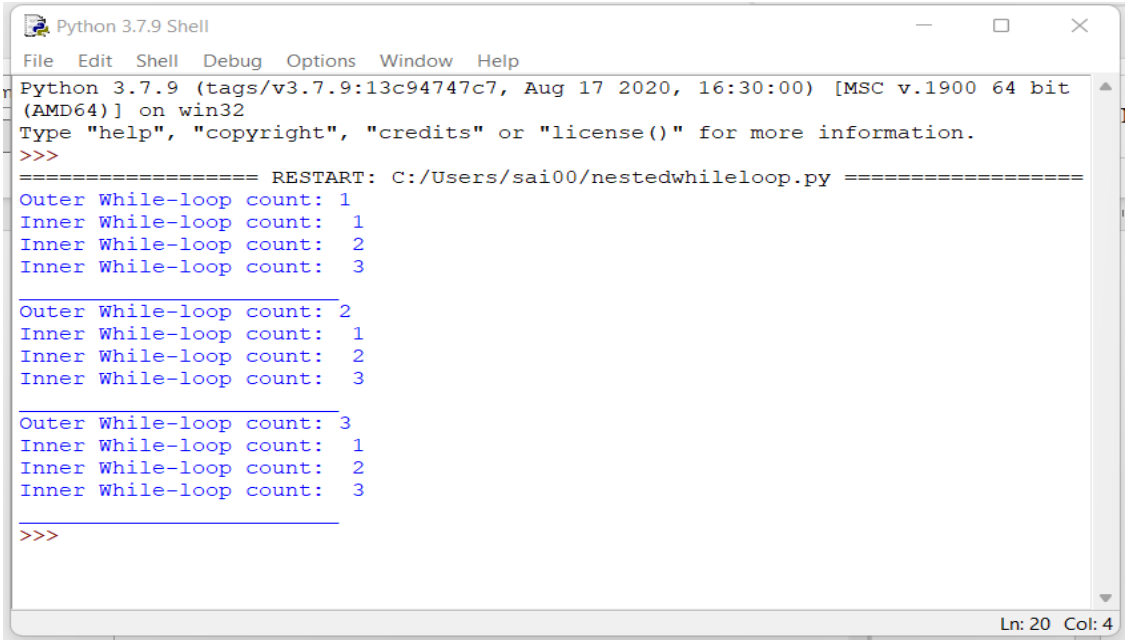
In this example, the nested loop performs an iteration over each item in the 'matrix' list and then prints the elements on a new line.

When one while loop is contained within another while loop, the resulting structure is referred to as a nested while loop. We require nested loops in most of our apps.

Example:



Output:



In this example, the nested loop performs an iteration over each item in the 'matrix' list and then prints the elements on a new line.

Break Statement

A premature termination of the loop in Python can be accomplished with the help of the break statement. It is utilized in situations in which we wish to exit the loop prior to it having finished all of its iterations.

Syntax:

The syntax of the break statement in Python is as follows:

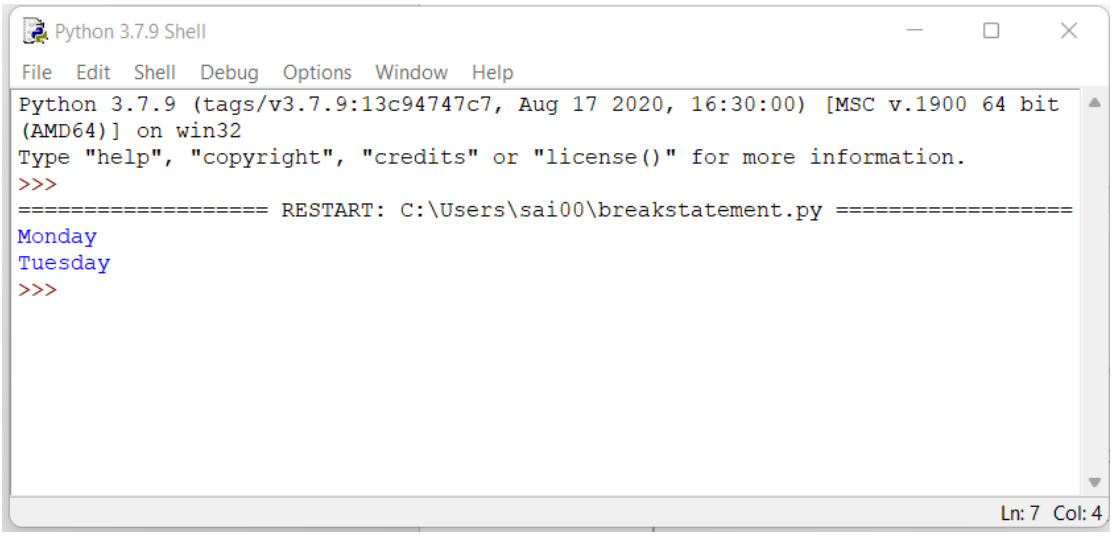
```
for variable in sequence:
    if condition:
        break
```

- The value of each element in the sequence is stored in the variable, which is a temporary variable, and it is used for each iteration of the loop to save the value.
- The condition is a statement that receives a boolean value and is evaluated at the beginning of each iteration of the loop. If the condition is found to be true, the break statement is carried out, therefore bringing an end to the loop.

Example:

The code that you see above has a for loop that outputs each item in the "fruits" list on a new line after iterating over each item in the list. On the other hand, the break statement is executed, and the loop is halted when the value of the "fruit" variable is equal to "banana."

Output:



Continue Statement

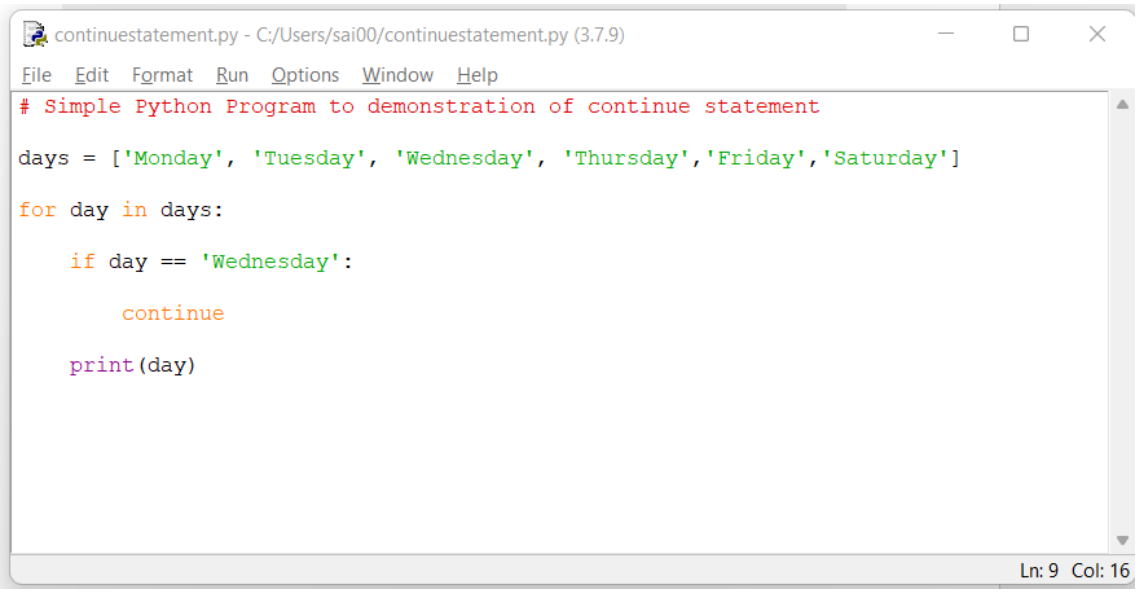
Using the continue statement in Python, one can skip the iteration of the loop that is currently being executed. It is utilized in situations in which we wish to skip a certain component of the sequence and proceed with the subsequent iteration of the loop onward.

Syntax:

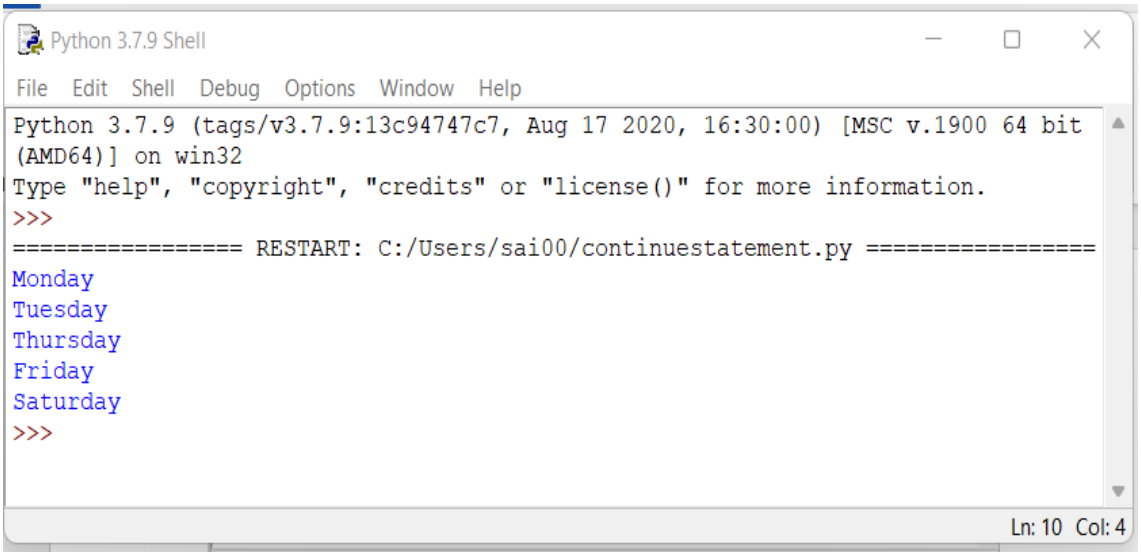
```
for variable in sequence:
    if condition:
        continue
    # Code block to be executed
```

- The value of each element in the sequence is stored in the variable, which is a temporary variable, and it is used for each iteration of the loop to save the value.
- The condition is a statement that receives a boolean value and is evaluated at the beginning of each iteration of the loop.

Example



Output:



The for loop iterates through each item in the "fruits" list in this example, printing each one on a new line. Nevertheless, the loop's current iteration is skipped and the continue statement is executed when the value of the "fruit" variable equals "banana."

Pass Statement

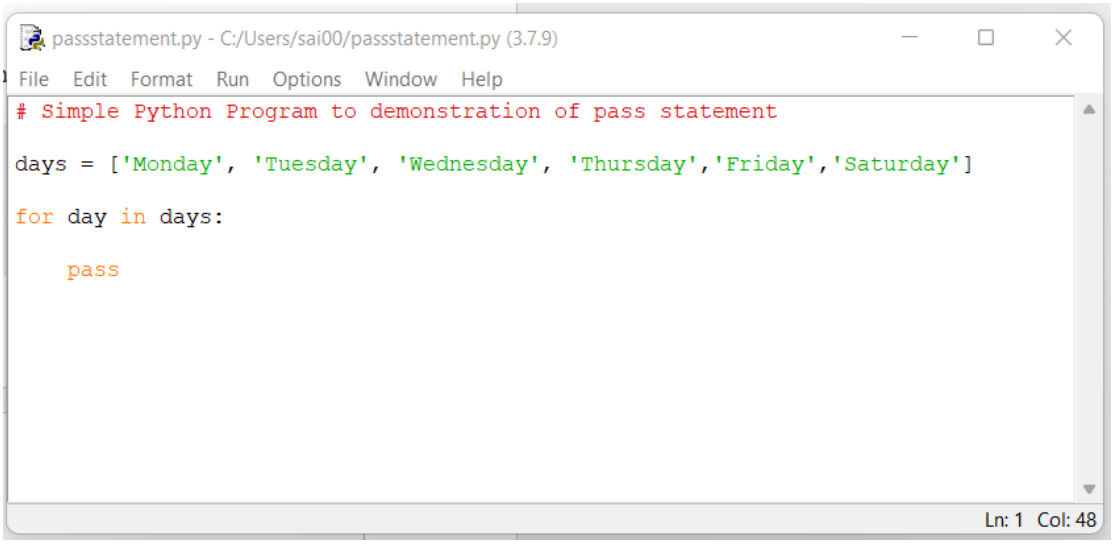
The pass statement is used as a placeholder in Python. It is used when we want to write empty code blocks and want to come back and fill them in later. The syntax of the pass statement in Python is given below.

Syntax:

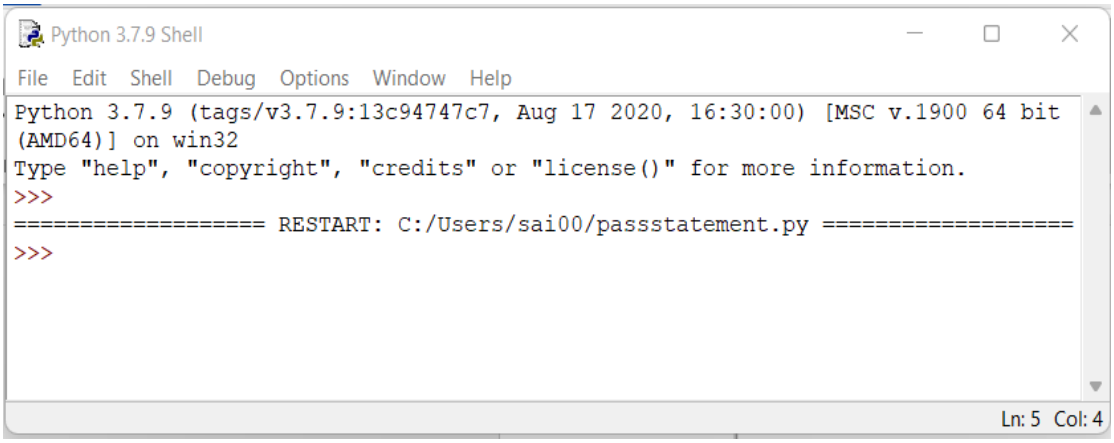
```
for variable in sequence:
    pass
```

- Every time the loop iterates, the variable—which is a temporary variable—holds the value of every element in the sequence.
- An empty code block is created using the pass statement and is subsequently filled in.

Example:



In this example, the pass statement is used to create an empty code block while the for loop iterates over each element in the "fruits" list.



3.5 USER-DEFINED FUNCTIONS

Functions are reusable blocks of code designed to perform a specific task. Python allows users to define their own functions using the def keyword.

Syntax:

```
def function_name(parameters):
    # function body
    return value
```

Example:

```
def greet(name):
    print('Hello', name)
greet('Lavanya')
```

Output:

Hello Lavanya

Built-in Functions

Python's built-in functions are already defined. A user must remember the name and parameters of a certain function. There is no need to redefine these functions because they have already been defined.Some of the widely used built-in functions are given below and shown in Table 10.1:

Table 3.1 Built-in Functions in Python

Function	Description
pow()	Returns the power of two numbers
abs()	Returns the absolute value of a number
max()	Returns the largest item in a python iterable
min()	Returns the largest item in a python iterable
sum()	Sum() in Python returns the sum of all the items in an iterator
type()	The type() in Python returns the type of a python object
Sqrt()	Executes the python built-in to find sqrt of the given number

The following two example python codes shown in below demonstrate the usage of built-in functions to fulfil the specific task. In the first example python code imported math module

and later performed the two functions pow () and sqrt() operations. The result of each function is produced on the output.

Example

predined.py - C:/Users/sai00/predined.py (3.7.9)

File Edit Format Run Options Window Help

```
# Python code demonstrate predefined funtions

# import package

import math

# find square root of a given number

print("The squre root of 25 is =",math.sqrt(25))


# find power of two numbers

print("The power of 5 & 9 =",pow(5,9))
```

Ln: 16 Col: 0

Output

Python 3.7.9 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/predined.py ==========
The squre root of 25 is = 5.0
The power of 5 & 9 = 1953125
>>>
```

Ln: 7 Col: 4

Similarly, the second example also imported math module and perform the abs (), max () and min () operations respectively. The absolute function took the -25 is a negative number and produced the output as 25. The maximum of 5 and 9 is determined by max () and minimum is returned by min () function.

Example

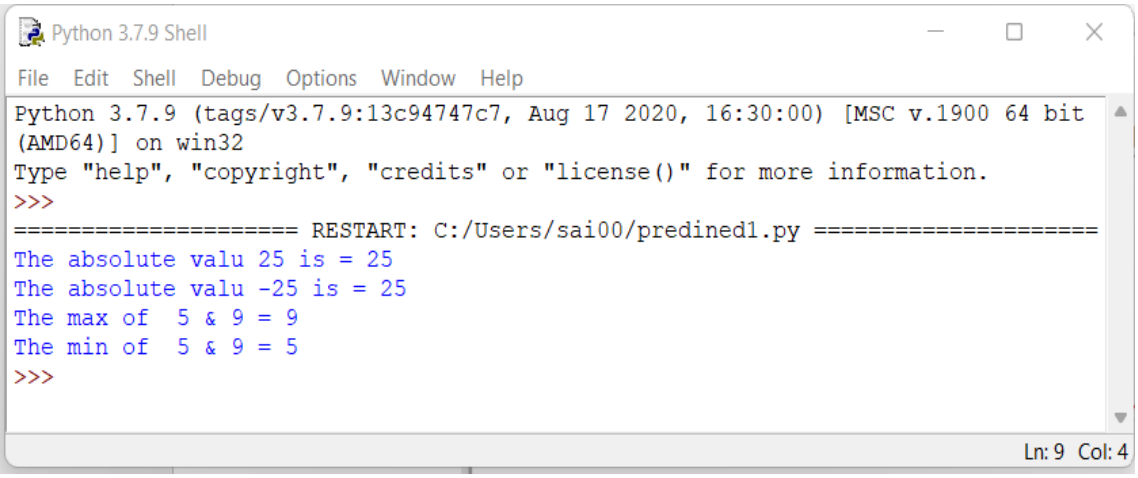
predined1.py - C:/Users/sai00/predined1.py (3.7.9)

File Edit Format Run Options Window Help

```
# Python code demonstrate predefined funtions

# import package

import math

# find absolute value of  a given number

print("The absolute valu 25 is =",abs(25))

print("The absolute valu -25 is =",abs(-25))

# find max and min of given number

print("The max of  5 & 9 =",max(5,9))

print("The min of  5 & 9 =",min(5,9))
```

Ln: 11 Col: 40

Output:



3.6 PYTHON VARIABLES AND ASSIGNMENTS

Variables are used to store data values. In Python, variables are created automatically when a value is assigned. Python is dynamically typed, so there is no need to declare variable types explicitly.

```
x = 5
y = 2.5
name = 'Python'
print(x, y, name)
```

3.7 PARAMETER PASSING

Python supports several ways to pass parameters to functions: positional, keyword, default, and variable-length arguments.

Example:

```
def add(a, b=10):
    return a + b
print(add(5))
print(add(5, 15))
```

Output:

15
20

3.8 SUMMARY

In this chapter, we explored imperative programming concepts, Python program structure, control flow statements, and user-defined functions.

We also discussed variable assignments and different parameter passing techniques. Understanding these concepts helps in writing efficient and structured Python programs.

3.9 TECHNICAL TERMS

- Imperative Programming
- Control Structure
- Function
- Variable
- Parameter Passing
- Sequential Execution

3.10 SELF-ASSESSMENT QUESTIONS

Essay Questions:

1. Explain the concept of Imperative Programming with suitable examples.
2. Describe different types of execution control structures in Python.
3. Discuss the types of parameter passing in Python with examples.

Short Notes:

1. Write about Python variables and assignments.
2. Discuss about user-defined functions with examples.

3.11 SUGGESTED READINGS

1. Steven cooper – Data Science from Scratch, Kindle edition.
2. Reemathareja – Python Programming using problem solving approach, Oxford Publication
3. "Think Python: How to Think Like a Computer Scientist" by Allen Downey
4. "Python Cookbook" by David Beazley and Brian K. Jones
5. "Programming Python" by Mark Lutz
6. Ljubomir Perkovic, “Introduction to Computing Using Python: An Application Development Focus”, Wiley, 2012.
7. Charles Dierbach, “Introduction to Computer Science Using Python: A Computational Problem-Solving Focus”, Wiley, 2013.

Dr. U Surya Kameswari

LESSON- 04

STRING

AIMS AND OBJECTIVES

The primary goal of this chapter is to grasp the concept of string in Python programming. The chapter began with an understanding of basic definition of string, creating a string, and so on. After completing this chapter, the student will understand how to work with string in python in terms various methods, operations, and functions.

STRUCTURE

4.1. INTRODUCTION

Python strings, like those in many other well-known programming languages, are arrays of bytes that represent unicode characters. Nevertheless, a single character in Python is just a string with a length of 1. Python does not have a character data type. You can access the string's constituents by using square brackets.

Since it is an immutable data type, you are unable to alter a string after you have created it. Strings are extensively utilized in a wide range of applications, including the storing and manipulation of text data as well as the representation of names, addresses, and other text-representable data types. This chapter will cover Python strings, one of the core data types in Python programming, and will cover Python string methods, operators and functions, working with them, and more.

4.2. PYTHON STRING

A string is a sequence of alphabets, words, or other characters. It is one of the most basic data structures, serving as the foundation for data manipulation. Python includes a built-in string class called str. Python strings are "immutable," which implies they cannot be modified once formed.

4.2.1. Creating a Python String

To create a String in python there are three different types of approaches:

- With a Single quotes

  'Welcome to the world of "Python" keep Loving.'

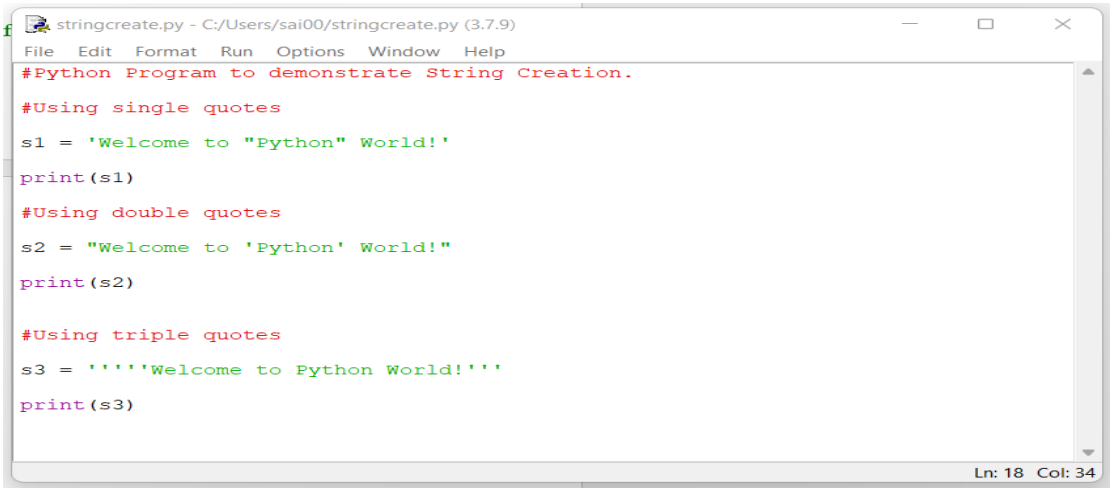
- With a Double quotes.

  "Welcome to the world of ' Python ' keep Loving."

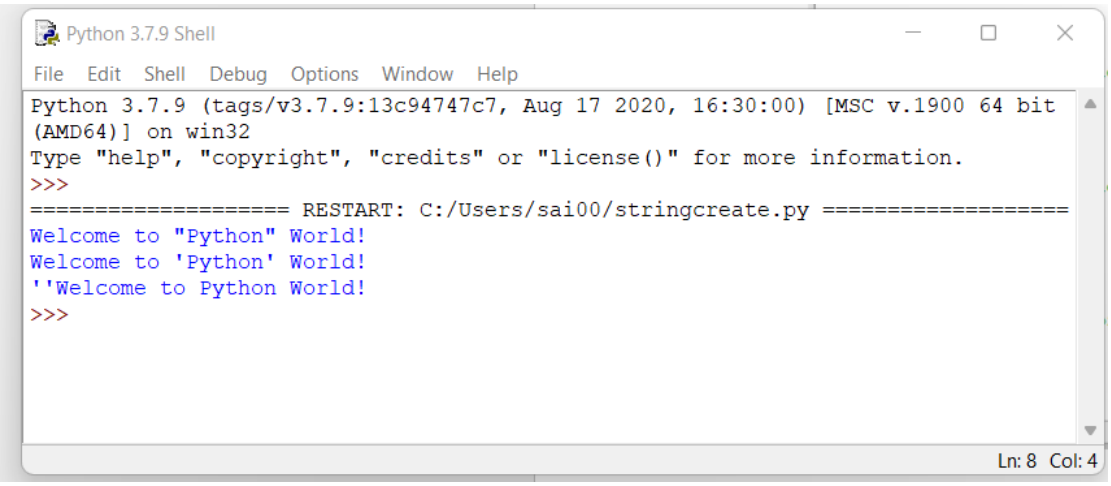
- With a Triple quotes,

  """ Welcome to the world of Python """, '''Keep Loving.'''

Example:



Output:



The above example, where three strings are created namely S1,S2 and S3 in different styles with same content. Finally displayed the three strings output is shown above.

4.2.2. Applications of Python Sting

- Use of string matching algorithms to quickly detect instances of plagiarism in both code and text.
- Strings can be utilized for encoding and decoding purposes, ensuring the secure movement of data from source to destination.
- We are able to offer better filters for the approximate suffix-prefix overlap problem by utilizing strings and the techniques associated with them.
- HTTP requests and responses, among other data exchanged over networks, are encoded and decoded using strings.
- When working with files, you'll need to know that strings are the go-to for reading and writing file names and locations.
- Applications like sentiment analysis and natural language processing make use of strings to glean useful insights from massive text datasets.

4.3 ACCESSING THE STRING

There are three various methods that we can get the characters from the individual String that was already constructed in the previous section. The information is given below:

- Indexing
- Negative Indexing
- Slicing

4.3.1 Indexing

Using index values and treating strings like a list is one method. In Python, the Indexing function can be used to retrieve specific characters from a String. The idea of indexing technique is shown in Figure 4.1

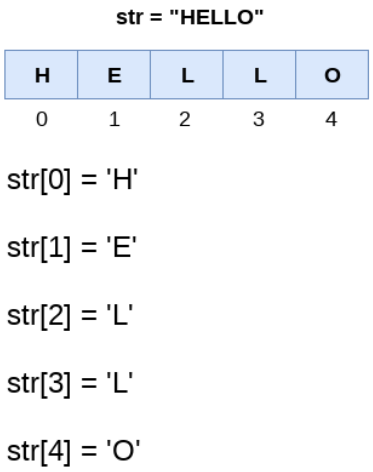
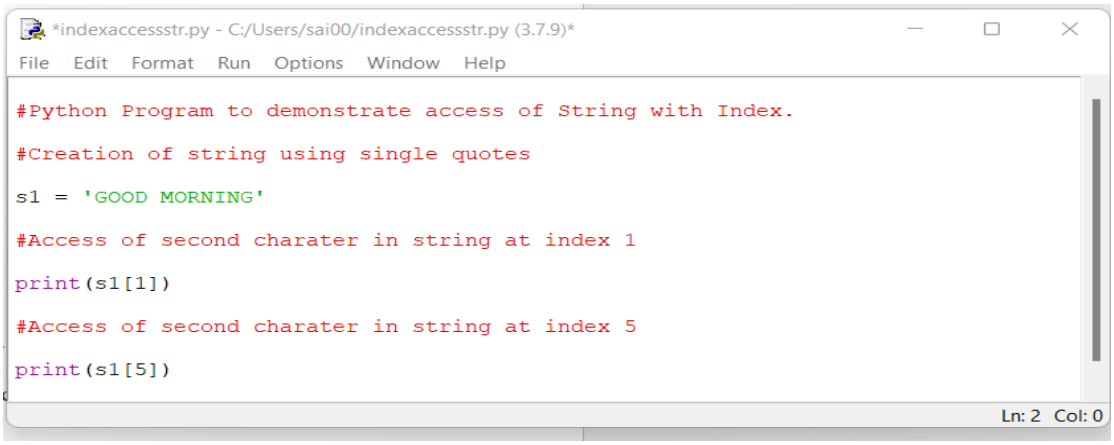
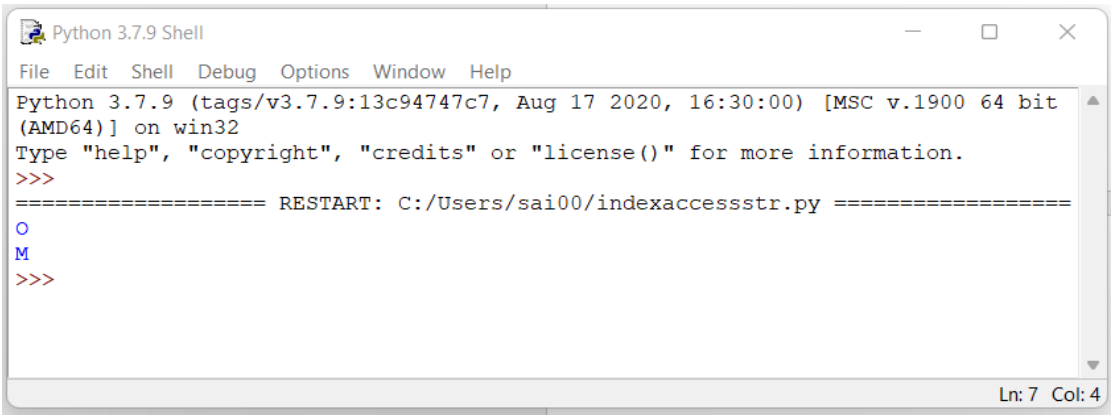


Fig 4.1. Indexing technique to access String

Example:



Output:



The above example illustrate the concept of indexing method, where one strings S1 is created with the content of "GOOD MORNING" and then accessed character at index 1 and 5. Finally displayed the extracted characters output is shown above.

4.3.2 Negative Indexing

Python's string language permits negative indexing, just as that of a list. Negative address references, such as -1 for the final character, -2 for the second last character, and so forth, can access characters from the back of the String thanks to indexing. The idea of negative indexing is shown in Figure 6.1.

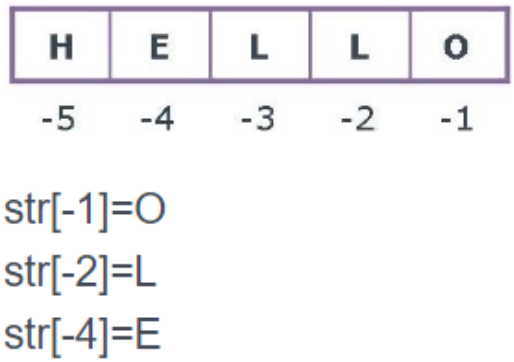
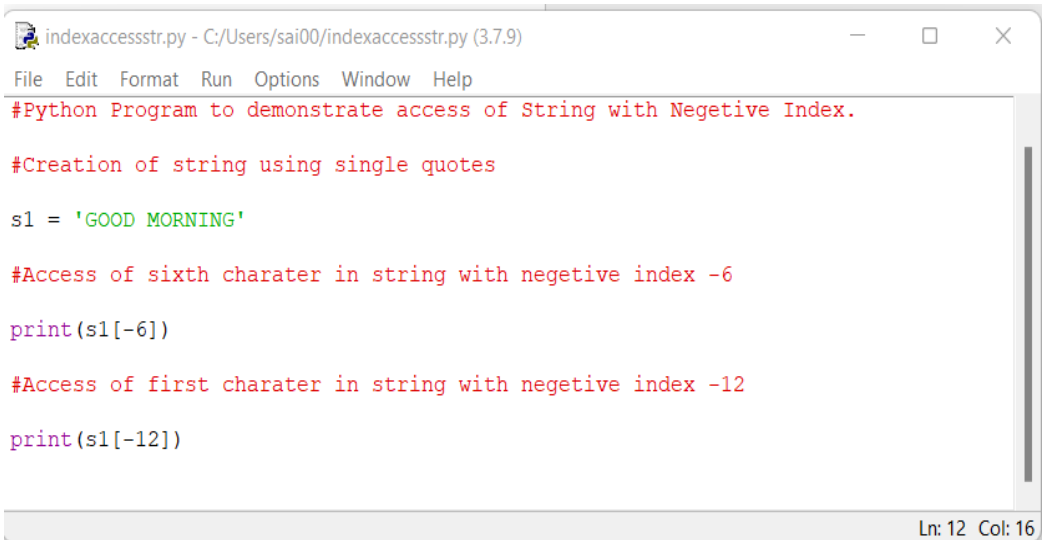
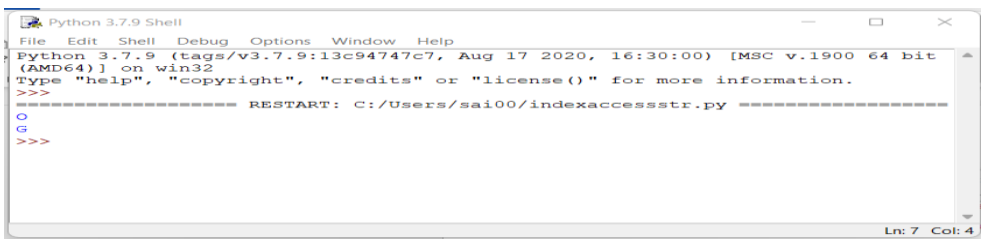


Fig 4.2. Negetive Indexing technique to access String

Example:



Output:

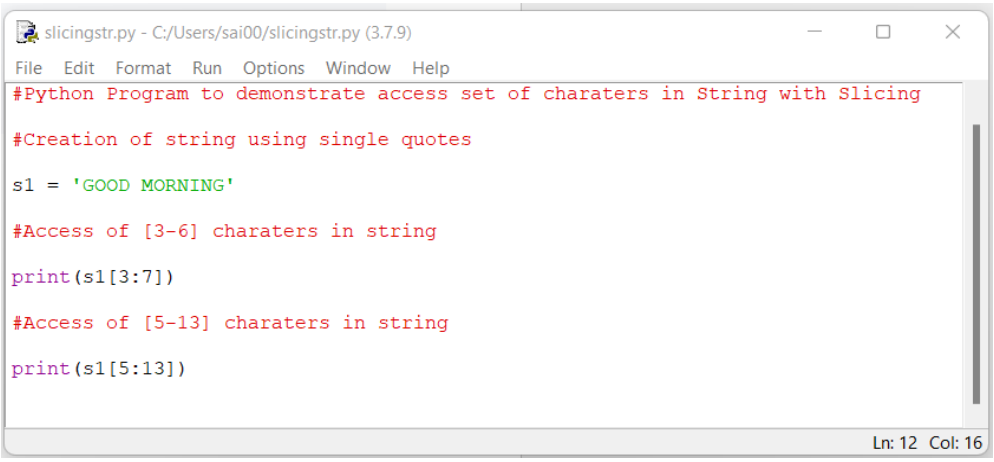


The above example illustrate the concept of negative indexing method, where one strings S1 is created with the content of "**GOOD MORNING**" and then accessed character at index-6 and -12. Finally displayed the extracted characters output is shown above.

4.3.3. Slicing

The String Slicing function in Python can be used to retrieve a range of characters from the String. To slice something in a string, use a slicing operator, such as a colon (:). When utilizing this method, bear in mind that the character at the start index is included in the string that is returned, but the character at the last index is not.

Example:



Output:

The above example illustrate the concept of slicing method, where one strings S1 is created with the content of "**GOOD MORNING**" and then accessed character with range of [3-7] and [5-13] . Finally displayed the extracted sub string output is shown above.

4.4 PYTHON STRING OPERATIONS

Python's basic string operations include doing simple arithmetic operations, verifying the character of an existing substring, repeating a string, and much more are shown in Table 6.1.

Table 3.1. Python String Operations

Operation	Python Expression	Description
Concatenation	s1 + s2	"Concatenation operator" is the name given to this operator, which is used to unite two or more stings
Repetition	s * n	The repetition operator is the name given to this. There will be several copies of the same string created by it.
Membership	in	The membership operator is the name given to this. Whether or whether a certain character or sub string is included in the string that was supplied is returned by it.
	not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular string or character is not present in the specified. It gives a return value of true if the character or sub string is not included in the string that was supplied. Otherwise return false.
Comparison	s1 == s2	Returns True if string, s1 is the same as string, s2. Otherwise False.
	s1 != s2	Returns True if string, s1 is not the same as string, s2. Otherwise False.

4.4.1 Concatenation Operator

Concatenating or joining two or more strings is a common task while programming. To connect or concatenate two strings in this sense, use the plus operator (+) and the idea is shown in Figure 6.3. Python's concatenation operator only joins items of the same type, in contrast to other languages like JavaScript where type coercion allows us to concatenate a string and an integer.

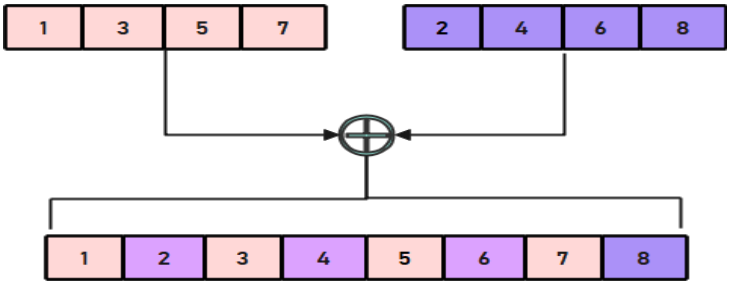
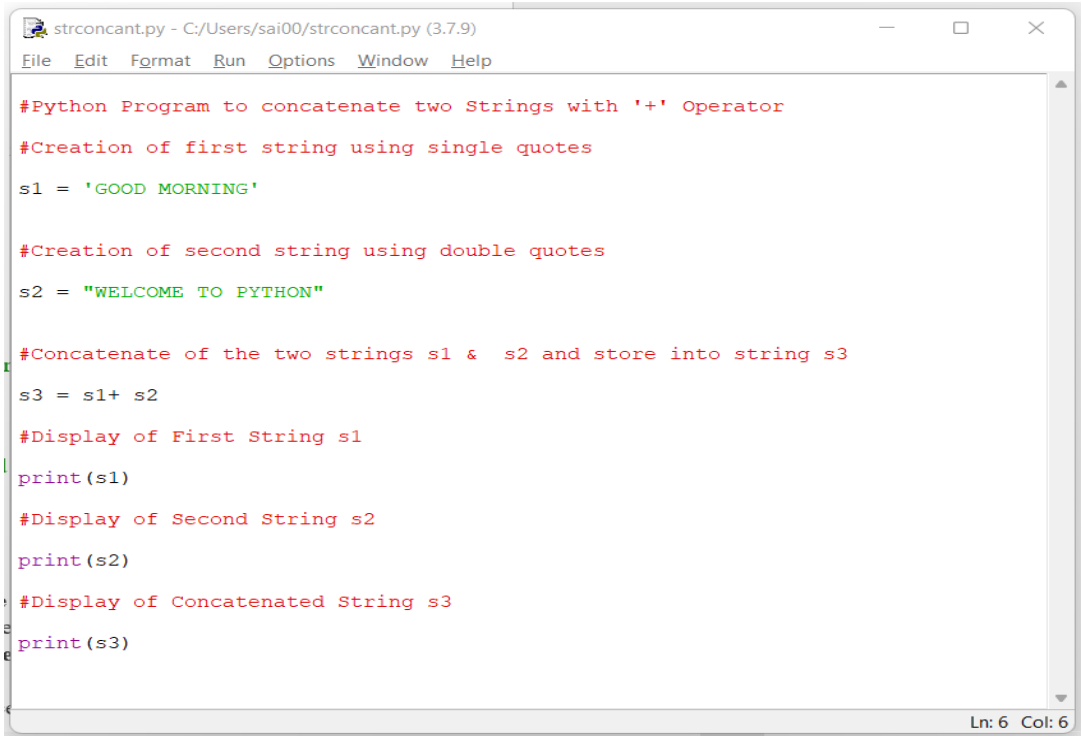
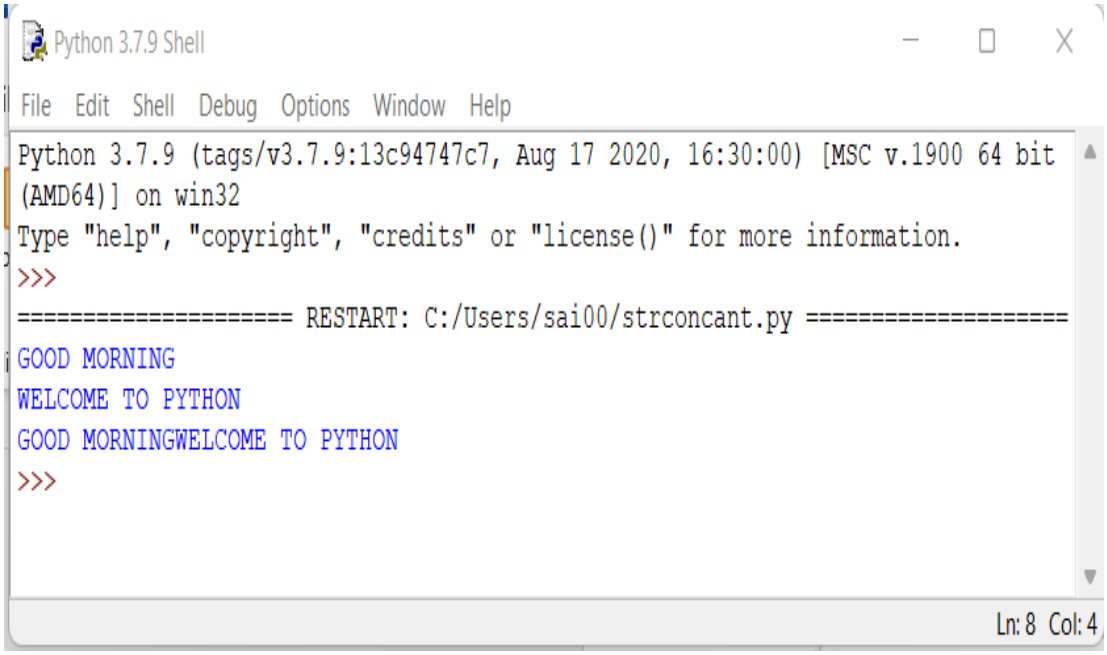


Fig 4.3. Concatenate of Two Lists with '+' Operator

Example:

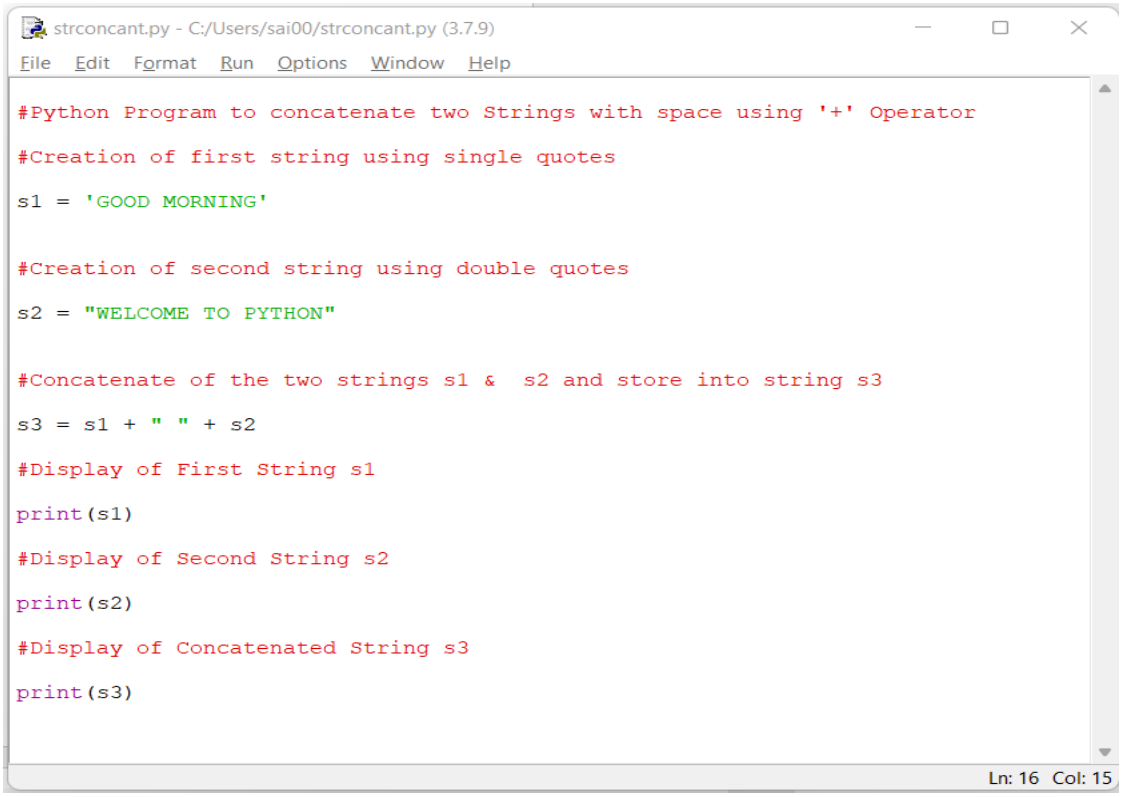


Output:

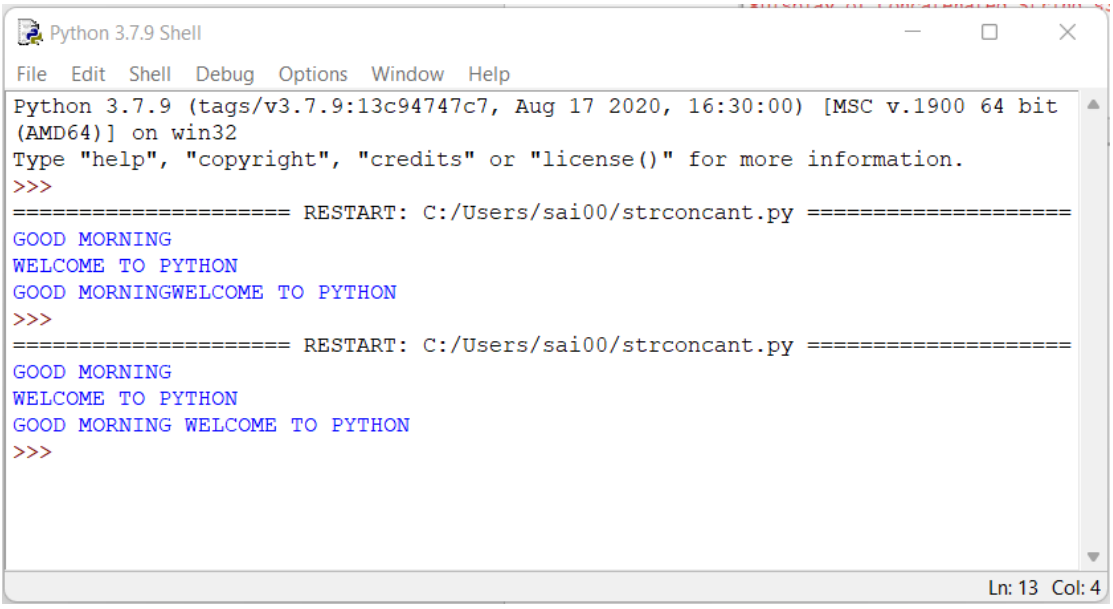


Strings are sequences that cannot be changed, as we previously stated. Concatenating the two strings in the previous example doesn't change either string. Rather, the process generates a new string called "S3" from the two strings "S1" and "S2." This operator is frequently used by beginners to add spaces between strings. This space is a string as well, but it's empty this time.

Example:



Output:



4.4.2 Repetition Operator

The purpose of this operator is to return a string that has been repeated a predetermined number of times. This string is included in the new string, and it is repeated the number of times that was requested. This is accomplished by the utilization of the multiplication

operator (*). Take for example that we have a string S and an integer N. Doing S times N or N times S will result in S being repeated N times. The idea is shown in Figure 6.4.

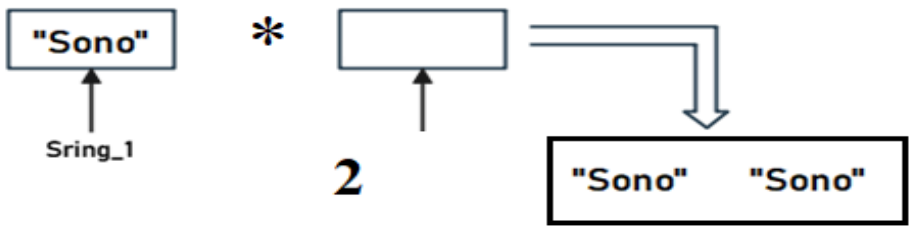


Fig 4.4. Repetition of Strings with '*' Operator

Example:

```
strrepet.py - C:/Users/sai00/strrepet.py (3.7.9)
File Edit Format Run Options Window Help
#Python Program to create 'n' copies of given Strings with '*' Operator
#Creation of string using single quotes
s1 = 'GOOD MORNING!'

#Create the string s1 into 3 copies
s2 = s1 * 3

#Create the string s1 into 5 copies
s3 = s1 * 5
#Display of First String s1
print(s1)
#Display of Second String s2 as 3 copies of s1
print(s2)
#Display of Third String s3 as 5 copies of s1
print(s3)
Ln: 2 Col: 59
```

Output:

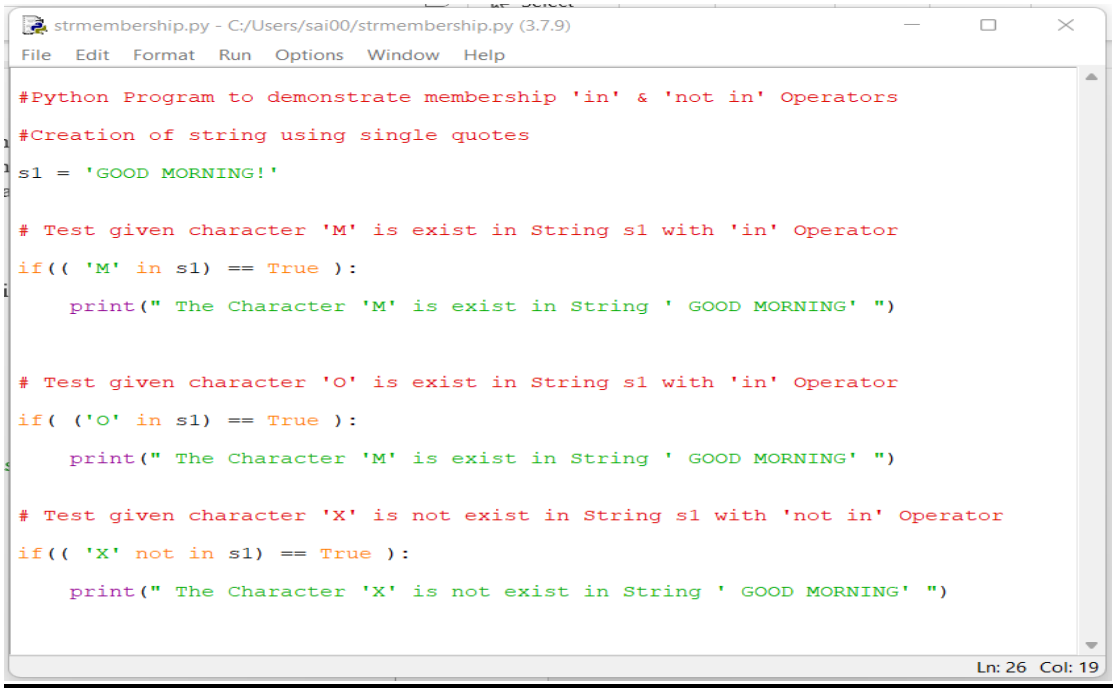
```
Python 3.7.9 Shell
File Edit Shell Debug Options Window Help
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/strrepet.py ==========
GOOD MORNING!
GOOD MORNING!GOOD MORNING!GOOD MORNING!
GOOD MORNING!GOOD MORNING!GOOD MORNING!GOOD MORNING!GOOD MORNING!
>>>
Ln: 8 Col: 4
```

Notice the last two print functions in the preceding example. Both actually output empty strings. The last but one step seems sense because it creates zero copies of the string, but the last operation appears strange. However, multiplying a string by a negative number yields an empty string.

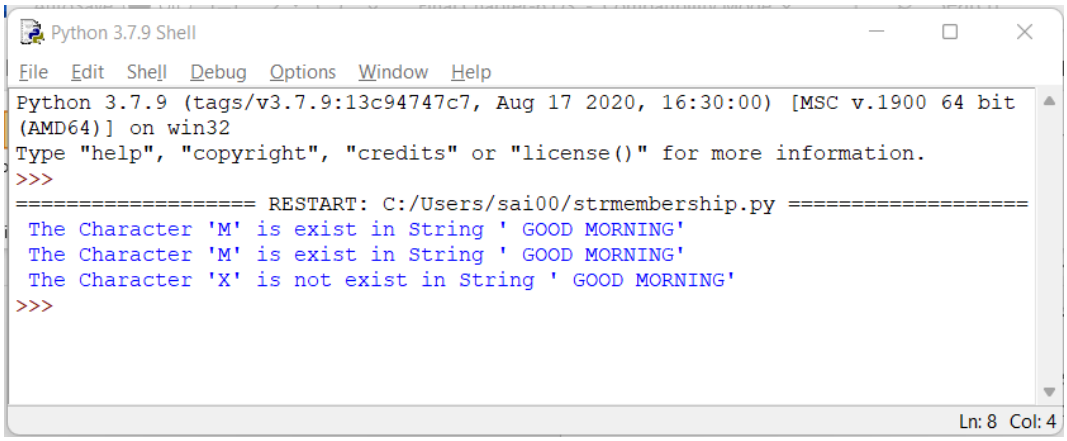
4.4.3 Membership Operator

These operators are commonly used to determine whether or not an element or character occurs in a specific string. The in function returns True if a character x exists in a given string, and False otherwise. The not in function returns True if a character x does not appear in a provided string, and False otherwise.

Example:

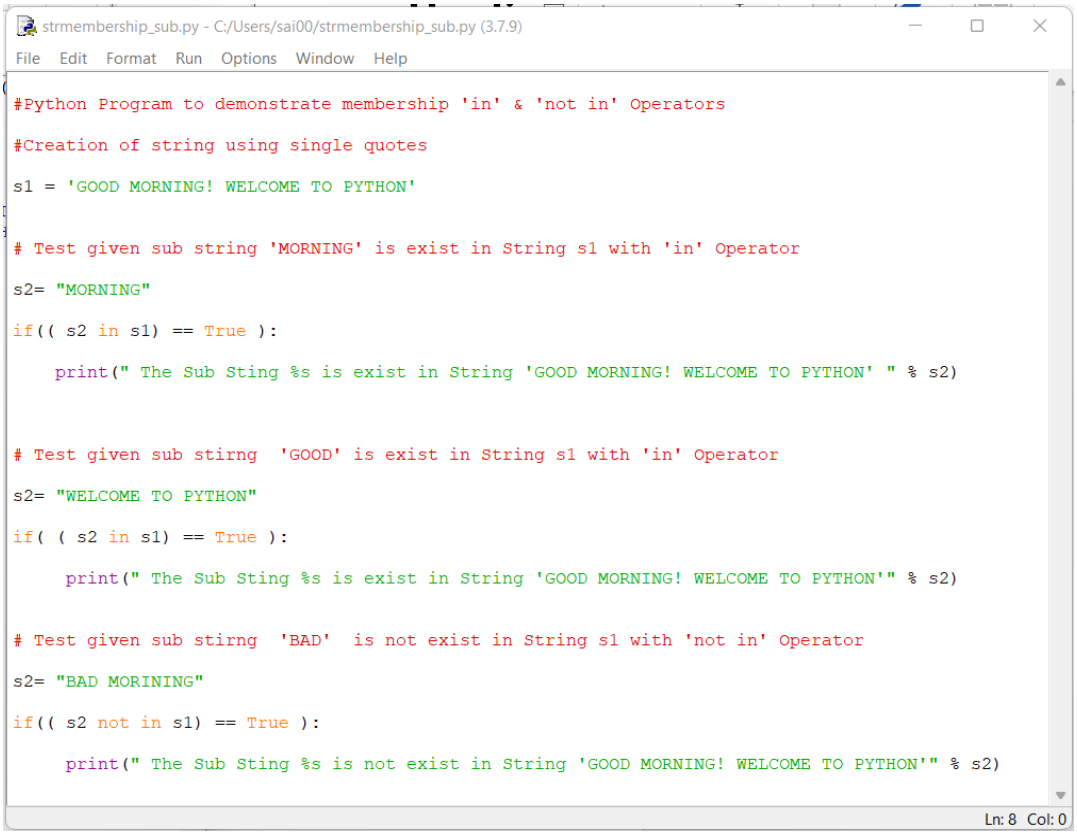


Output:

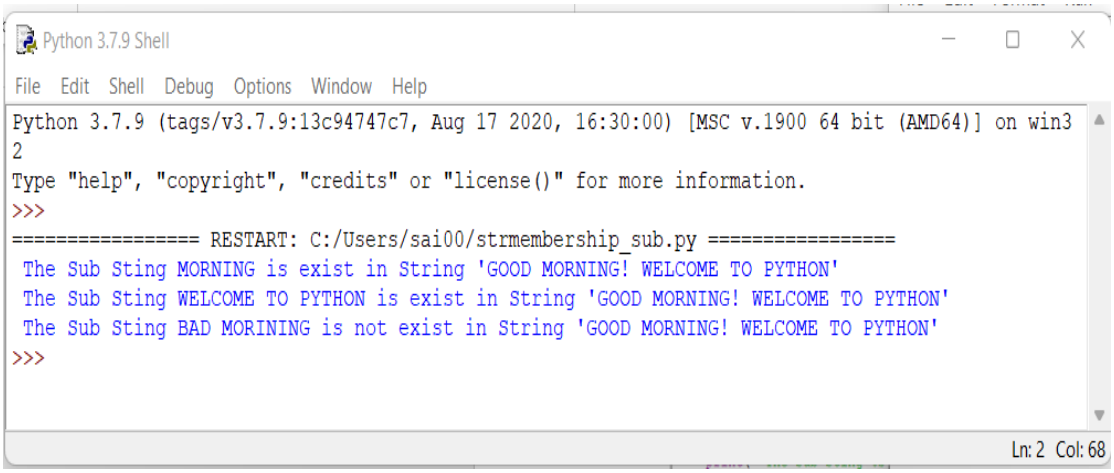


It is important to keep in mind that the membership operators are also capable of working with substrings; that is, they can determine whether or not a substring is present in a string.

Example:



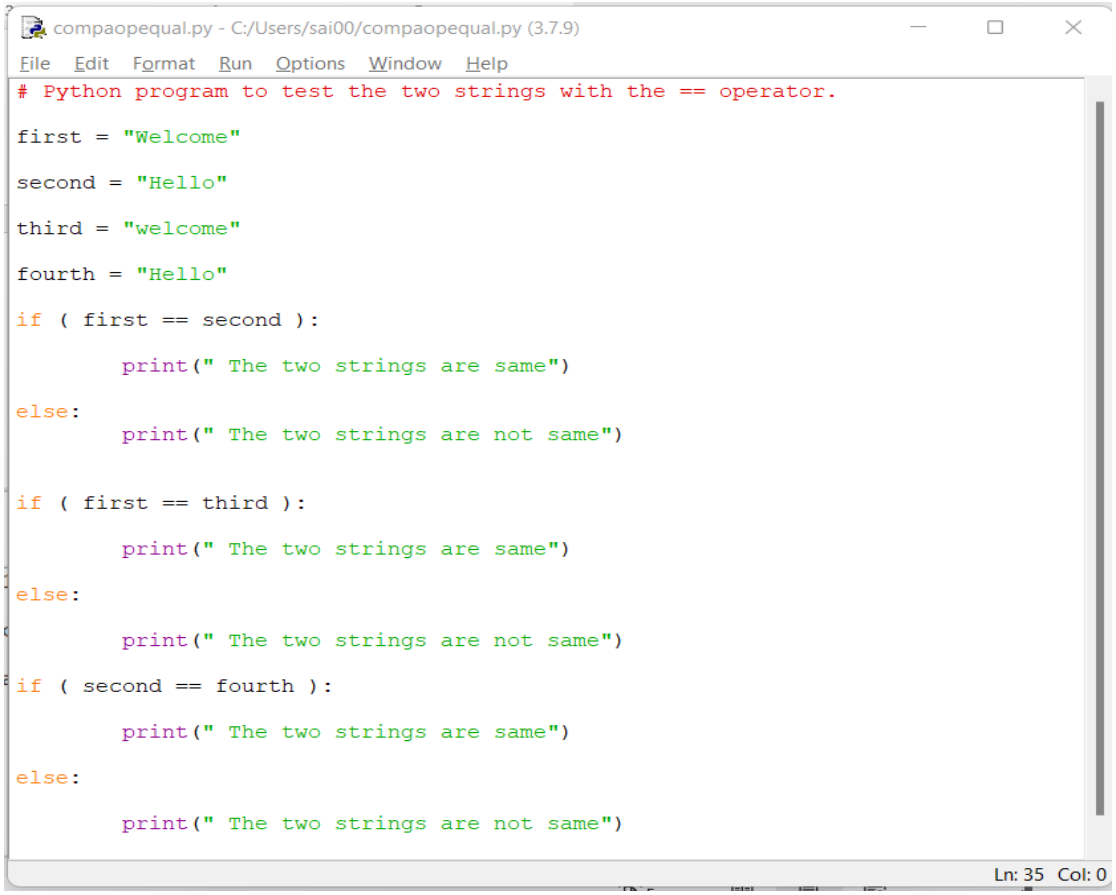
Output:



4.4.4 Comparison Operator

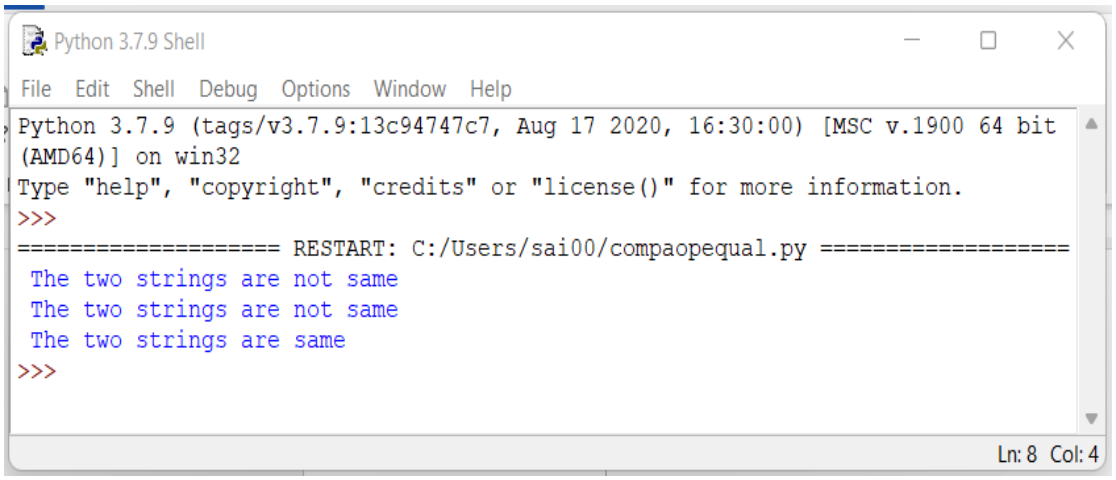
The purpose of these operators in Python is to verify the equivalence of two operands, which in this case are two strings.

Example:

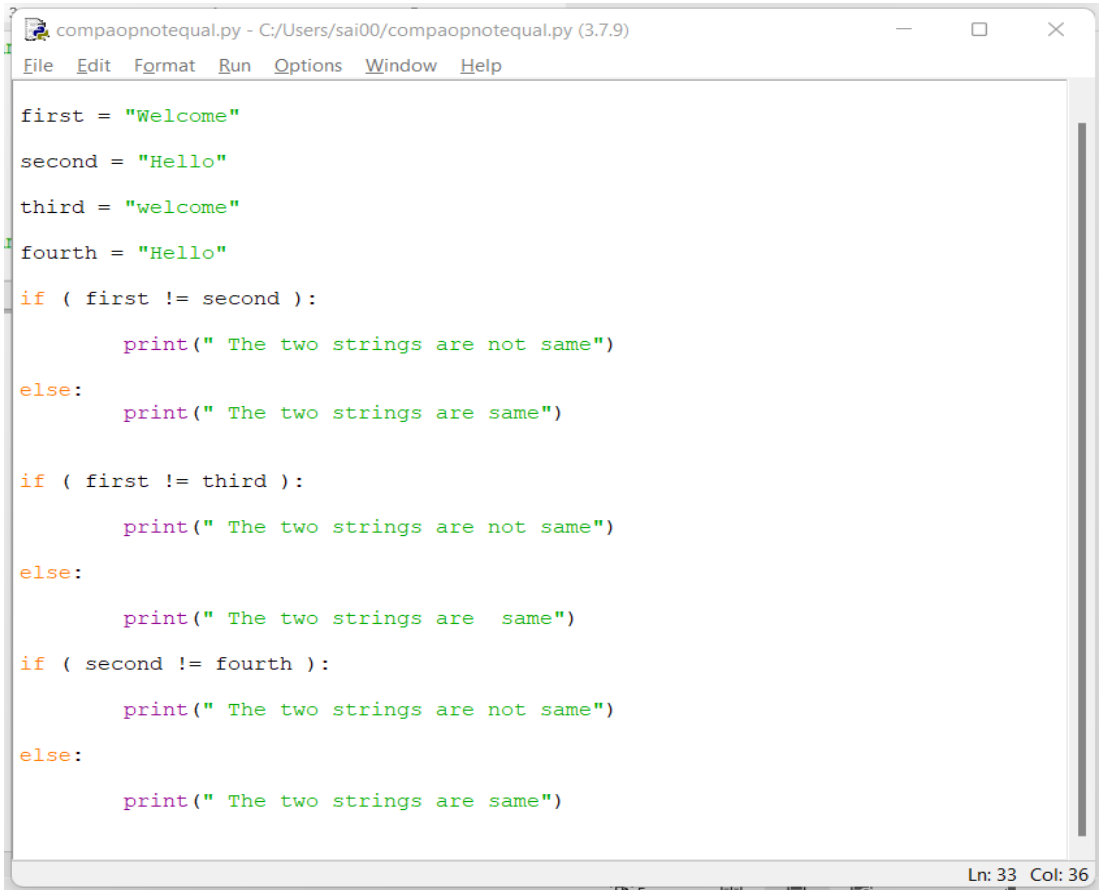


Their names also indicate that they are used for this purpose. However, because they return a boolean, they are most utilized in conditional expressions to determine whether or not two strings are identical. A True value is returned by the == operator when the two strings in question are identical, whereas a False value is returned when the strings in question are not identical.

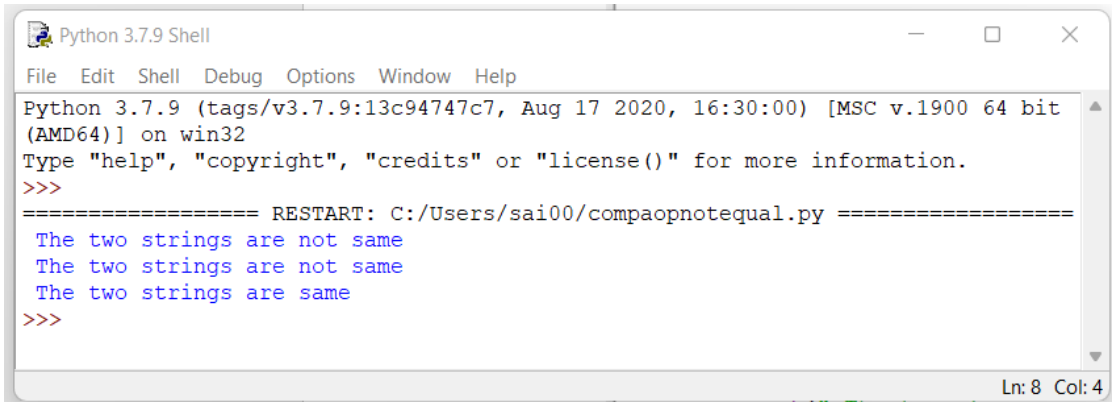
Output



Example:



Output:



As shown in the preceding examples, make a comparison between the first string, the second string, and the third string. Along the same lines, second with fourth. != is the operator that returns. True when the two strings are equal otherwise return false.

4.5. PYTHON STRING METHODS

Python comes with several built-in methods that can be used to execute operations and manipulations while working with strings. Listed below are several string methods that are frequently used:

4.5.1 len()

It is possible to utilize the `len()` function in order to determine the length of a string. A count of the characters contained in the string is returned by it.

Example:

```
String_new = "Welcome to Python!"

length = len(String_new)

print(String_new)
```

Output

18

The total number of characters includes space returned by the len() function i.e. 13

4.5.2 upper()

The string that is returned by the upper() method is one in which all of the characters are capitalized.

Example:

```
String_new = "Welcome to Python!"

String_new = String_new.upper()

print(String_new)
```

Output :

WELCOME TO PYTHON!

The upper() is called along with string object String_new.upper and it returns a string where all characters are in upper case.

4.5.3 replace()

Using the replace() method, a phrase that is supplied is replaced with another term that is also specified.

Example:

```
String_new = "Welcome to Python!"

String_new = String_new.replace("Pyhton", "PYTHON" )

print(String_new)
```

Output:

Welcome to PYTHON!

The reverse r() is called along with old and new string and it replaces an old string "Python" with new string " PYHON"

4.5.4. find()

Using the find() method, one can locate the initial instance of the value that has been supplied. However, if the value cannot be located, this procedure will return -1. This method is essentially identical to the index() method; the only difference is that the index() method throws an exception if the value is not found. In addition, this method is almost identical to the index() method.

Example:

```
String_new = "Welcome to Python!"

String_new = String_new.find("Python")

print(String_new)
```

Output:

11

4.6 FORMATTED OUTPUT

The output of a program often needs to be organized so that users can easily read and interpret the results.

Formatted Output Using print() Function

To achieve this, Python offers several formatting techniques:

- Using the print() function.
- Using the format() method for string interpolation.

Python's print() function provides several ways to produce formatted output. It can display variables with separators, new lines, or in customized layouts.

Example 1 – Printing with different separators

```
n = 5
r = 5 / 3
name = 'Ida'
print(n, r, name)
print(n, r, name, sep=';')
```

```
print(n, r, name, sep='\n')
```

Output

```
5 1.66666666667 Ida
5;1.66666666667;Ida
5
1.66666666667
Ida
```

Formatted Output with the format() Method

Python's str.format() function allows inserting variables into a string at specific placeholders {}.

Example1:

```
weekday = 'Wednesday'
month = 'March'
day = 10
year = 2010
hour = 11
minute = 45
second = 33
print('{}, {} {}, {} at {}:{}:{}'.format(weekday, month, day, year, hour, minute, second))
```

Output

Wednesday, March 10, 2010 at 11:45:33

Explanation

In the statement above:

'{} , {} {}, {} at {}:{}:{}'.format(weekday, month, day, year, hour, minute, second)

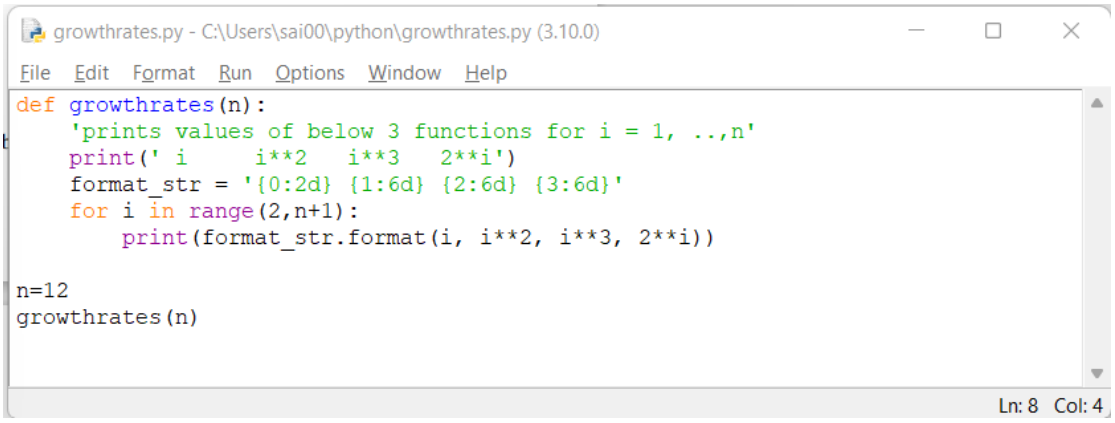
Each pair of curly braces {} is a placeholder for a variable supplied to the format() function.Python replaces them in the same order they appear inside the parentheses.

Example2:

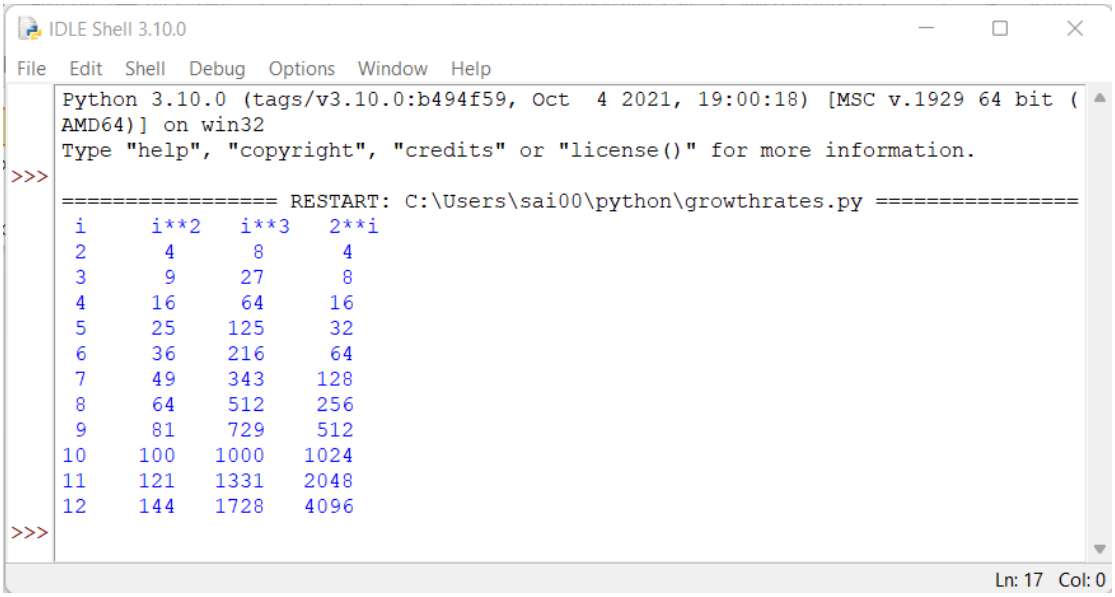
To illustrate the issues, let's consider the problem of properly lining up values of functions i^2, i^3 and 2^i for $i = 1, 2, 3, \ldots$ Lining up the values properly is useful because it illustrates the very different growth rates of these functions:

```
 i    i**2    i**3   2**i
 1       1       1      2
 2       4       8      4
 3       9      27      8
 4      16      64     16
 5      25     125     32
 6      36     216     64
 7      49     343    128
 8      64     512    256
 9      81     729    512
10     100    1000   1024
11     121    1331   2048
12     144    1728   4096
```

Python code:



Output:



4.7 SUMMARY

Strings are an essential data type in Python, and they are utilized widely for activities that involve working with textual data. In this chapter, we covered the fundamentals of creating and manipulating strings, as well as accessing characters, string slicing, concatenation, string length, and the different string methods that are available in Python. Your ability to work effectively with strings in your Python programs and to handle text-based data in an efficient manner will be directly correlated to your level of comprehension of these ideas.

4.8 TECHNICAL TERMS

String, Indexing, Negative Indexing, Concatenation, Membership, comparison and Slicing, format,print

4.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. How is a String created and called? Explain.
2. What are the various List Operations? Explain.
3. Explain about List Methods with example.
4. Illustrate formate output with formate() function.

Short Notes:

1. Write about indexing method for sting access.
2. Discuss about applications of python string.
3. Explain about Slicing method with example.
4. Describe about format output using print().

4.10 SUGGESTED READINGS

1. Steven cooper – Data Science from Scratch, Kindle edition.
2. Reemathareja – Python Programming using problem solving approach, Oxford Publication
3. "Python Crash Course" by Eric Matthes
4. "Automate the Boring Stuff with Python" by Al Sweigart
5. "Learning Python" by Mark Lutz
6. Ljubomir Perkovic, “Introduction to Computing Using Python: An Application Development Focus”, Wiley, 2012.
7. Charles Dierbach, “Introduction to Computer Science Using Python: A Computational Problem-Solving Focus”, Wiley, 2013.

Dr. U Surya Kameswari

LESSON- 05

FILES

AIMS AND OBJECTIVES

The goal of this chapter is to explain how Python programs interact with files stored in a computer's file system.
Students will learn how to open, read, write, and close files, as well as explore the most common patterns used to process textual data.

After completing this chapter, students will be able to:

- Understand how file systems organize data.
- Use Python's built-in file handling functions.
- Read and write text files efficiently.
- Manage file resources properly using the with statement.
- Recognize common file-handling errors and handle them safely

STRUCTURE

5.1 INTRODUCTION

Programs often need to store information permanently, beyond the life of a single execution. Variables and data structures reside in main memory (RAM), which is temporary; once the program ends, that data disappears. To preserve data, we store it in files on a secondary storage device (like a hard disk). A file is a named collection of data saved on a storage medium. Python can read data from existing files and write new information to files. This process is called file input and output (file I/O).

5.2 UNDERSTANDING THE FILE SYSTEM

Every computer's operating system manages a file system, which organizes data into files and directories (folders).Each file has a name, location (path), and a type that determines how the contents are interpreted.

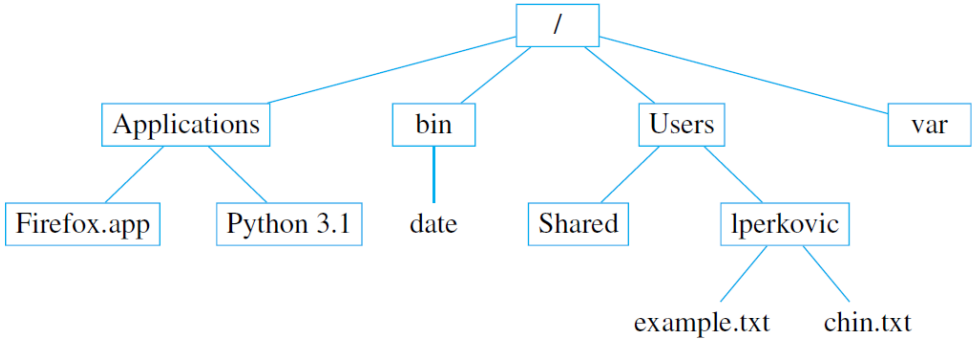


Figure 5.1 Mac OS X file System organization

For example:

- data.txt – Text file
- sales.csv – Comma-separated values file
- image.jpg – Binary image file

File paths can be absolute (full path from the root directory) or relative (based on the current working directory).

For example, the absolute pathname of folder Python 3.1 is

/Applications/Python 3.1

while the absolute pathname of file example.txt is

/Users/lperkovic/example.txt

the relative pathname of file example.txt in Figure 5.1 is

lperkovic/example.txt

Python interacts with this file system through built-in functions such as open(), read(), write(), and close().

5.3 OPENING AND CLOSING A FILE

Processing a file consists of these three steps:

1. Opening a file for reading or writing
2. Reading from the file and/or writing to the file
3. Closing the file

The open() function creates a connection between the program and the file:

file_object = open(filename, mode)

Parameters:

- filename – The name or path of the file.
- mode – A string indicating the file access mode.

Table 5.1 Different Modes of opening a file

Mode	Meaning	Description
'r'	Read	Opens a file for reading (default).
'w'	Write	Opens a file for writing; existing contents are erased.
'a'	Append	Opens a file for writing at the end; retains existing data.
'r+'	Read/Write	Opens for both reading and writing.
'b'	Binary	Used with above modes for binary files.

After finishing, close the file using:

file_object.close()

Example:

```
f = open('example.txt', 'r')
data = f.read()
print(data)
f.close()
```

5.4 PATTERNS FOR READING A TEXT FILE

Reading data from a file can be done in several ways depending on the amount and structure of data.

5.4.1 Reading the Entire File

```
f = open('poem.txt', 'r')
contents = f.read()
print(contents)
f.close()
```

This method reads the whole file into one string.

Use this for small text files only, because it loads everything into memory at once.

5.4.2 Reading Line by Line

A common and memory-efficient way is to iterate directly through the file object:

```
f = open('poem.txt', 'r')
for line in f:
    print(line.strip())
f.close()
```

Here each iteration reads one line until the end of the file.

5.4.3 Using readline() and readlines()

readline() reads one line at a time;

readlines() returns a list containing all lines.

```
f = open('data.txt', 'r')
line1 = f.readline()
print(line1)
all_lines = f.readlines()
print(all_lines)
f.close()
```

Method Usage	Explanation
`infile.read(n)`	Read *n* characters from the file `infile` or until the end of the file is reached, and return characters read as a string
`infile.read()`	Read characters from file `infile` until the end of the file and return characters read as a string
`infile.readline()`	Read file `infile` until (and including) the new line character or until end of file, whichever is first, and return characters read as a string
`infile.readlines()`	Read file `infile` until the end of the file and return the characters read as a list lines
`outfile.write(s)`	Write string `s` to file `outfile`
`file.close()`	Close the file

Fig 5.2 Common File Methods in Python

5.4.4 Using the with Statement

The recommended method is to use with — it automatically closes the file:

```
with open('story.txt', 'r') as f:
    for line in f:
        print(line.strip())
```

When the block inside with completes, Python automatically calls f.close() even if an error occurs.

5.4. PATTERNS FOR READING A TEXT FILE

Reading data from a file can be done in several ways depending on the amount and structure of data.

Example: Counting Lines, Words, and Characters in a Text File

The following program counts the number of **lines**, **words**, and **characters** in a given text file.

```
def fileStats(filename):
infile = open(filename, 'r')
lines = infile.readlines()
infile.close()

num_lines = len(lines)
num_words = sum(len(line.split()) for line in lines)
num_chars = sum(len(line) for line in lines)

return num_lines, num_words, num_chars
```

Input File – example.txt

The 3 lines in this file end with the new line character.

There is a blank line above this line.

Output

Lines: 3
Words: 15
Characters: 96

Explanation:

- readlines() reads all lines from the file.
- len(lines) counts total lines (including blank lines).
- line.split() splits each line into words, and the total is summed for word count.
- len(line) counts the number of characters (including spaces and newline characters).

This is a common **file-reading pattern** used in text processing and data analysis.

Example: Searching for a Target String in a File

Python allows searching for specific substrings in text files.
The following function myGrep() imitates the Unix grep command — it prints every line that contains a given target string.

```
def myGrep(filename, target):
    infile = open(filename, 'r')
    for line in infile:
        if target in line:
            print(line.strip())
    infile.close()
```

Input File – example.txt

The 3 lines in this file end with the new line character.

There is a blank line above this line.

Program Call

myGrep('example.txt', 'line')

Output

The 3 lines in this file end with the new line character.

There is a blank line above this line.

Explanation:

- The file is opened in read mode ('r').
- Each line is checked using the in operator to see if the target substring occurs in it.
- Lines containing the target word are printed after removing trailing newlines with .strip().
- The file is then closed using .close().

This simple function demonstrates pattern searching in text files — a foundational concept for text processing, data filtering, and log analysis in Python.

Example: Replacing a Target String in a File

Sometimes, we need to modify text inside a file — for example, to replace one word with another.

The function myReplace() reads a text file line by line, replaces all occurrences of a target string with a replacement string, and writes the modified lines to a new output file.

```
def myReplace(fileName, target, replacement):
   infile = open(fileName, 'r')
   outfile = open('new_' + fileName, 'w')

   for line in infile:
      new_line = line.replace(target, replacement)
      outfile.write(new_line)

   infile.close()
   outfile.close()
```

Input File – example.txt

The 3 lines in this file end with the new line character.

There is a blank line above this line.

Program Call

```
myReplace('example.txt', 'line', 'sentence')
```

Output File – new_example.txt

The 3 sentences in this file end with the new sentence character.

There is a blank sentence above this sentence.

Explanation:

- The program opens the input file (example.txt) in read mode and creates a new output file (new_example.txt) in write mode.
- Each line is processed using the replace() method, which substitutes all occurrences of the target string with the replacement.
- The modified lines are written to the new file.
- Finally, both files are closed to ensure proper resource handling.

This example illustrates a common file-processing pattern:

1. Read from a source file.
2. Modify the content according to some rule.
3. Write the updated data into a new file.

5.5 WRITING TO A TEXT FILE

Writing data to a file is similar but uses write-enabled modes ('w', 'a', 'r+').

5.5.1 Writing Strings and Data

```
f = open('output.txt', 'w')
f.write('Welcome to Python file handling.\n')
f.write('This is line 2.')
f.close()
```

This creates (or overwrites) the file and writes the specified text.

5.5.2 Appending to a File

Appending adds new data without removing existing content:

```
with open('output.txt', 'a') as f:
   f.write('\nThis line was appended later.')
```

5.5.3 Writing Lists and Formatted Data

Use a loop or the writelines() method:

```
lines = ['apple\n', 'banana\n', 'cherry\n']
with open('fruits.txt', 'w') as f:
   f.writelines(lines)
```

You can also format text before writing:

```
with open('marks.txt', 'w') as f:
   for name, mark in [('Ravi', 85), ('Meena', 90)]:
      f.write('{}: {}\n'.format(name, mark))
```

Example: Writing Data to a Text File

When a program generates output that should be stored permanently, it can write data to a text file.

Python provides the write() method to store string data into a file.

If numeric or non-string values are to be written, they must first be converted into strings.

```
# Example: Writing to a Text File
outfile = open('test.txt', 'w')

outfile.write('1 This is the first line. Still the first line...\n')
outfile.write('2 Now we are in the second line.\n')

value = 5
outfile.write('3 Non string value like ' + str(value) + ' must be converted first.\n')
outfile.write('4 Non string value like ' + str(value) + ' must be converted first. WRITING TO A TEXT FILE\n')

outfile.close()
```

Output File – test.txt

```
1 This is the first line. Still the first line...
2 Now we are in the second line.
```

3 Non string value like 5 must be converted first.
4 Non string value like 5 must be converted first. WRITING TO A TEXT FILE

Explanation:

- The file test.txt is opened in write mode ('w'), creating a new file in the current working directory.
- Each write() call adds one line to the file.
- The \n character ensures each new line starts properly.
- The numeric value 5 is first converted to a string using str(value) before being written.
- Finally, close() is called to save and release the file.

Example: Appending Data to a File

```
# Open file in append mode
    outfile = open('test.txt', 'a')

    outfile.write('\n5 This line is added later using append mode.')
    outfile.write('\n6 File content is preserved and new data is added at the end.')

    outfile.close()
```

Updated File – test.txt

```
    1 This is the first line. Still the first line...
    2 Now we are in the second line.
    3 Non string value like 5 must be converted first.
    4 Non string value like 5 must be converted first. WRITING TO A TEXT FILE
    5 This line is added later using append mode.
    6 File content is preserved and new data is added at the end.
```

Explanation:

- Opening a file with 'a' does not erase its existing contents.
- Every call to write() adds text to the end of the file.
- This mode is ideal for logging, adding new records, or progressively storing results.

Example: Reading and Writing Together with 'r+'

```
# Open file for both reading and writing
    file = open('test.txt', 'r+')
# Read and display existing content
    print('Existing File Content:\n')
    print(file.read())
# Write additional data
    file.write('\n7 This line is added using r+ mode.')
    file.close()
    Output (on screen):
```

Existing File Content:

1 This is the first line. Still the first line...
2 Now we are in the second line.
3 Non string value like 5 must be converted first.
4 Non string value like 5 must be converted first. WRITING TO A TEXT FILE
5 This line is added later using append mode.
6 File content is preserved and new data is added at the end.
Updated File – test.txt
...
7 This line is added using r+ mode.

Explanation:

- The 'r+' mode permits both reading and writing operations.
- Reading begins at the start of the file, and writing begins after the file pointer position (which can be moved using seek() if required).
- It is useful when you want to read existing content, make changes, and rewrite data within the same file.

5.6 PRACTICAL EXAMPLES OF FILE HANDLING

Example 1: Counting Lines and Words

```
with open('essay.txt', 'r') as f:
   lines = f.readlines()
print('Number of lines:', len(lines))
word_count = sum(len(line.split()) for line in lines)
print('Number of words:', word_count)
```

Input File – essay.txt

Python is an easy to learn programming language.
It supports multiple programming paradigms.
File handling in Python is simple and powerful.

Output

Number of lines: 3
Number of words: 18

Explanation:

- readlines() reads all the lines from the file into a list.
- len(lines) gives the total number of lines.
- Each line is split into words using split(), and the total number of words is computed with sum().

Example 2: Copying Contents from One File to Another

```
with open('source.txt', 'r') as src, open('copy.txt', 'w') as dst:
   for line in src:
      dst.write(line)
```

Input File – source.txt

Learning Python is fun.
This file will be copied to another file.

Output File – copy.txt

Learning Python is fun.
This file will be copied to another file.

Explanation:

- The first file (source.txt) is opened for reading.
- The second file (copy.txt) is opened for writing.
- Each line from the source is written to the destination.
- The with statement ensures both files are properly closed after use.

Example 3: Filtering Data from a File

```
with open('numbers.txt', 'r') as f:
    for num in f:
        if int(num) % 2 == 0:
            print(num.strip())
```

Input File – numbers.txt

11
12
15
20
23
30

Output

12
20
30

Explanation:

- Each line is read as a string and converted to an integer using int(num).
- The expression int(num) % 2 == 0 checks whether the number is even.
- Only even numbers are printed after removing newline characters using .strip().

5.7 COMMON FILE ERRORS

Typical issues when dealing with files:

- File not found (FileNotFoundError)
- Permission denied (PermissionError)
- Wrong file mode or path

File Not Found (FileNotFoundError)

This error occurs when the program tries to open or read a file that does not exist in the specified directory.

Example:

```
f = open('missing.txt', 'r')
```

Output:

```
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
```

Reason:

- The file missing.txt does not exist in the current working directory.

Solution:

Use a try–except block to catch the error gracefully:

```
try:
   f = open('missing.txt', 'r')
except FileNotFoundError:
   print('Error: The specified file was not found.')
```

Permission Denied (PermissionError)

This error occurs when the program does not have the necessary permissions to access, read, or write a file.

Example:

```
f = open('/system/config.txt', 'w')
```

Output:

```
PermissionError: [Errno 13] Permission denied: '/system/config.txt'
```

Reason:

- The user may not have permission to modify or create files in that directory**.**

Solution:

Ensure you have the required access rights or change the file's location:

```
try:
   f = open('config.txt', 'w')
except PermissionError:
   print('Error: You do not have permission to modify this file.')
```

3. Wrong File Mode or Path

This error occurs when the file is opened using an incorrect mode (e.g., trying to read from a file opened in write mode) or when an invalid path is provided.

Example 1: Incorrect Mode

```
f = open('data.txt', 'w')
print(f.read())
```

Output:

```
io.UnsupportedOperation: not readable
```

Reason:

- The file is opened in 'w' mode, which allows writing only, not reading.

Solution:

```
f = open('data.txt', 'r')   # Open in read mode
print(f.read())
Example 2: Invalid Path
f = open('C:/wrongfolder/data.txt', 'r')
```

Output:

FileNotFoundError: [Errno 2] No such file or directory: 'C:/wrongfolder/data.txt'

Reason:

- The directory wrongfolder does not exist.

Solution:

- Verify the file path or use an absolute path:

```
f = open('C:/Users/YourName/Documents/data.txt', 'r')
```

5.8 SUMMARY

In this chapter, you learned how to:

- Work with the file system and understand file paths.
- Open, read, and write files using Python's built-in functions.
- Use different reading patterns such as read(), readline(), and readlines().
- Employ the with statement for automatic file management.
- Write and append text efficiently.
- Handle file-related errors using exceptions.

5.9 TECHNICAL TERMS

File System, File Modes, open(), close(), read(), write(), with Statement, Exception Handling

5.10 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the purpose of the file system and how Python interacts with it.
2. Describe the different file access modes in Python with examples.
3. Discuss various methods for reading data from a text file.
4. Explain the use of the with statement in file operations.
5. How can you handle exceptions during file input/output operations?

Short Notes

1. Write about the readline() and readlines() methods.
2. Differentiate between write ('w') and append ('a') modes.
3. Describe how to write lists to a file using writelines().

5.11 SUGGESTED READINGS

1. Ljubomir Perković – *Introduction to Computing Using Python*, John Wiley & Sons, 2012.
2. Reema Thareja – *Python Programming Using Problem-Solving Approach*, Oxford University Press.
3. Eric Matthes – *Python Crash Course*, No Starch Press.
4. Al Sweigart – *Automate the Boring Stuff with Python*.
5. Mark Lutz – *Learning Python*, O'Reilly Media.

Dr. Neelima Guntupalli

LESSON- 06

EXCEPTION HANDLING

AIMS AND OBJECTIVES

The objective of this chapter is to explain the nature of program errors, differentiate between syntax and runtime errors, and introduce Python's exception-handling mechanism for building reliable programs.

After completing this chapter, students will be able to:

- Recognize various types of errors in Python.
- Use try, except, else, and finally blocks effectively.
- Raise and define exceptions.
- Employ debugging and logging tools.
- Develop robust programs that recover from unexpected events

STRUCTURE

6.1 INTRODUCTION

Errors are an inevitable part of programming. Some errors prevent the program from running at all, while others occur only when it executes.

Python distinguishes between:

1. Syntax Errors – mistakes in program structure detected before execution.
2. Runtime Errors (Exceptions) – errors that occur during execution, such as dividing by zero or opening a missing file.
3. Logical Errors – the program runs but produces incorrect output due to faulty logic.

Exception handling provides a structured way to detect and respond to runtime problems without halting the entire program.

6.2 UNDERSTANDING PROGRAM ERRORS

- **Compile-time (syntax) errors:** violate Python's grammar rules.
- **Runtime errors (exceptions):** occur only when a particular statement executes.
- **Logic errors:** are semantic; Python cannot detect them automatically.

Good programming practice includes anticipating possible exceptions and writing code that handles them gracefully.

6.3 SYNTAX ERRORS

A SyntaxError occurs when the Python interpreter cannot parse the code.

```
# Missing colon
if x > 0
   print("Positive")
```

Output

```
SyntaxError: expected ':'
```

The interpreter stops immediately and points to the offending line.
These errors must be corrected before execution; they cannot be caught by try/except.

6.4 RUNTIME ERRORS AND EXCEPTIONS

A runtime error is detected while the program runs.
Python signals such problems by raising exceptions.

Example:

```
x = 5 / 0
```

Output

```
ZeroDivisionError: division by zero
```

When an exception occurs:

1. Normal flow stops.
2. Python searches for a matching except block.
3. If none is found, it prints a traceback and terminates the program.

6.5 EXCEPTION HIERARCHY IN PYTHON

All exceptions derive from the base class **BaseException.**

The commonly used root is Exception, from which most runtime errors inherit.

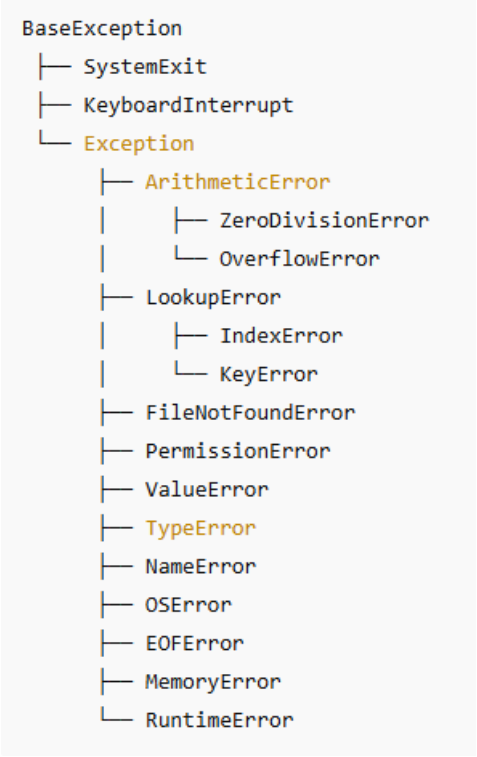


Fig 6.1. Exception Hierarchy

6.6 HANDLING EXCEPTIONS – try AND except

Use try / except to catch and handle predictable exceptions. The try block is used to test a block of code for errors.Python executes the statements inside the try block first.

Exception	Explanation
KeyboardInterrupt	Raised when user hits Ctrl-C, the interrupt key
OverflowError	Raised when a floating-point expression evaluates to a value that is too large
ZeroDivisionError	Raised when attempting to divide by 0
IOError	Raised when an I/O operation fails for an I/O-related reason
IndexError	Raised when a sequence index is outside the range of valid indexes
NameError	Raised when attempting to evaluate an unassigned identifier (name)
TypeError	Raised when an operation of function is applied to an object of the wrong type
ValueError	Raised when operation or function has an argument of the right type but incorrect value

Fig 6.2 Common exception types

If no error occurs, the program skips all except blocks and continues normally. If an error (exception) occurs, Python immediately stops the try block and looks for a matching except

The except block is used to handle the exception that occurs in the try block. It specifies what should happen when a particular error is detected.

```
try:
    num = int(input("Enter an integer: "))
    print(10 / num)
except ValueError:
    print("Input must be an integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Output Examples

Input: ten → Input must be an integer.

Input: 0 → Cannot divide by zero.

Multiple Except Blocks

You can use several except blocks to handle different error types separately.

```
try:
    f = open("data.txt", "r")
    n = int(f.readline())
    print(10 / n)
except FileNotFoundError:
    print("File not found.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid data in file.")
```

Key Points

- The try block contains code that may cause an exception.
- The except block contains code that handles that exception.
- You can have multiple except blocks for different exception types.
- If no exception occurs, the except blocks are skipped.
- You can also use a generic exception handler to catch any unexpected error:

```
except Exception as e:
        print("An error occurred:", e)
```

6.7 ELSE AND FINALLY BLOCKS

Python provides two additional optional clauses — **else** and **finally** — that can be used with the try–except structure to make exception handling more precise and complete.

They define what happens **when no exception occurs** (else) and **what must always be executed** (finally).

- **The else Block**

The **else** block executes **only if no exception occurs** in the try block.

It is useful for placing code that should run **only when the try block succeeds** without errors.

Syntax:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Handle the exception
else:
    # Code to execute if no exception occurred
```

Example:

```
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("Error: File not found.")
else:
    print("File opened successfully.")
    print(content)
    f.close()
```

Output (if file exists):

```
File opened successfully.
<file contents>
```

Output (if file does not exist):

```
Error: File not found.
```

Explanation:

- If open() raises a FileNotFoundError, the except block executes.
- If no error occurs, Python skips except and executes the else block.

2. The finally Block

The **finally** block executes **no matter what happens** — whether an exception occurs or not. It is used for **cleanup operations** like closing files, releasing resources, or disconnecting from databases.

Syntax:

```
try:
    # Risky operation
except ExceptionType:
    # Handle error
finally:
    # Code that always runs
```

Example:

```
try:
    f = open("data.txt", "r")
    data = f.read()
    print("Reading file completed.")
except FileNotFoundError:
    print("File not found.")
finally:
    print("Closing file (if opened).")
    try:
        f.close()
    except:
        pass
```

Possible Outputs:

```
Reading file completed.
Closing file (if opened).
or
File not found.
Closing file (if opened).
```

Explanation:

- The finally block executes regardless of whether an exception occurs.
- Even if a return, break, or continue statement is used inside try or except, the finally block will still run before the program exits that scope.
- It is commonly used to ensure resources (files, network connections, etc.) are released properly.

3. Combining try, except, else, and finally

All four clauses can be combined in one structure for full control:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Division successful. Result =", result)
finally:
    print("Program execution completed.")
```

Output 1

```
Enter a number: 0
Cannot divide by zero.
Program execution completed.
```

Output 2

```
Enter a number: 5
Division successful. Result = 2.0
Program execution completed.
```

Key Points

- **else** runs only if the try block has **no exceptions**.
- **finally** always executes — whether an exception occurs or not.
- finally is ideal for cleanup tasks like closing files or releasing resources.
- These clauses make programs more reliable and maintainable.

6.8 RAISING EXCEPTIONS WITH RAISE AND ASSERT

You can **raise** an exception manually when a condition is invalid.

```
def withdraw(balance, amount):
```

```
    if amount > balance:
        raise ValueError("Insufficient funds")
    return balance - amount
```

The **assert** statement checks logical conditions during testing.

```
assert 2 + 2 == 4
assert 5 < 3, "Assertion failed: invalid condition"
```

6.9 CREATING CUSTOM EXCEPTIONS

Python allows programmers to define their own **custom exceptions** to handle specific error situations that are not covered by built-in exceptions. Custom exceptions make programs **more readable, modular, and meaningful** because they describe the exact problem in the program's domain.

A **custom exception** is a user-defined class that **inherits from Python's built-in Exception class** (or one of its subclasses). By creating subclasses of Exception, programmers can define error types that are specific to their application.

Basic Syntax

```
class MyException(Exception):
    """Custom exception class."""
    pass
```

The pass statement is used here because we do not need to add new behavior; the class simply acts as a new type of exception.

Example 1: Creating and Raising a Custom Exception

```
class NegativeNumberError(Exception):
    """Raised when a negative number is encountered."""
    pass


def square_root(x):
    if x < 0:
        raise NegativeNumberError("Cannot compute square root of a negative number.")
    else:
        return x ** 0.5


try:
    print(square_root(-9))
except NegativeNumberError as e:
```

```
    print("Error:", e)
```

Output

Error: Cannot compute square root of a negative number.

Explanation:

- The class NegativeNumberError extends Exception.
- When a negative value is passed, the raise statement triggers this exception.
- The except block catches and handles it gracefully.

Example 2: Adding Custom Attributes

Custom exception classes can store additional information such as error codes or variable values.

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        super().__init__(f"Insufficient funds: Balance={balance},
 Withdrawal={amount}")
        self.balance = balance
        self.amount = amount


def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
    else:
        return balance - amount


try:
    new_balance = withdraw(500, 800)
except InsufficientFundsError as e:
    print("Transaction Failed!")
    print(e)
    print("Remaining balance:", e.balance)
```

Output

Transaction Failed!

Insufficient funds: Balance=500, Withdrawal=800

Remaining balance: 500

Explanation:

- The __init__ method initializes the exception with custom attributes.
- The super() call passes a formatted error message to the base Exception class.
- The caught exception can access e.balance or e.amount for more details.

Example 3: Multiple Custom Exceptions

You can define multiple custom exceptions to represent various application-specific errors.

```
class InvalidAgeError(Exception):
    pass

class AgeTooHighError(Exception):
    pass

def check_age(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative.")
    elif age > 120:
        raise AgeTooHighError("Age seems unrealistically high!")
    else:
        print("Valid age:", age)

try:
    check_age(150)
except (InvalidAgeError, AgeTooHighError) as e:
    print("Error:", e)
```

Output

Error: Age seems unrealistically high!

Advantages of Custom Exceptions

- **Clarity:** Communicate the exact type of problem in user-defined domains.
- **Modularity:** Separate error-handling logic from main program logic.
- **Hierarchy:** You can organize related exceptions under a parent custom exception.
- **Maintainability:** Easier debugging and understanding of specific failure conditions.

Example 4: Creating a Hierarchy of Custom Exceptions

```
class StudentError(Exception):
```

```
    """Base class for all student-related exceptions."""
    pass

class InvalidIDError(StudentError):
    pass

class MissingGradeError(StudentError):
    pass

try:
    raise MissingGradeError("Grade record not found for student ID 2025.")
except StudentError as e:
    print("Student Database Error:", e)
```

Output

Student Database Error: Grade record not found for student ID 2025.

6.10 MULTIPLE AND NESTED HANDLERS

You can nest or chain multiple handlers for complex logic.

```
try:
    with open('numbers.txt') as f:
        total = sum(int(x) for x in f)
except FileNotFoundError:
    print("Missing input file.")
except ValueError:
    print("File contained non-numeric data.")
else:
    print("Total =", total)
```

By creating a hierarchy, all student-related exceptions can be caught together using the base class StudentError.

- Custom exceptions are **classes derived from Exception**.
- Use the raise statement to trigger them intentionally.
- Include descriptive messages and attributes for context.
- Organize related custom exceptions using **inheritance**.
- Handling them separately enhances **readability** and **error diagnostics**.

Table 6.1 Example Summary

Custom Exception	Purpose
Negative Number Error	Handles negative input values.
Insufficient Funds Error	Raised when withdrawal exceeds balance.
Invalid Age Error, Age Too High Error	Manage age-related input errors.
Student Error Hierarchy	Group of related user-defined exceptions.

6.11 USING with FOR RESOURCE MANAGEMENT

In Python, many operations involve **external resources** such as files, network connections, or databases.These resources must be **explicitly released** after use to prevent memory leaks, file corruption, or system slowdowns.The **with statement** provides a safe and elegant way to manage such resources automatically.

The **with** statement is used to **wrap the execution of a block of code within methods defined by a context manager**. It ensures that **resources are acquired and released properly**, even if an exception occurs inside the block.

Syntax:

with expression as variable:
  # Code block using the resource
When the block under with finishes execution:

- The resource is **automatically cleaned up** (e.g., the file is closed).
- Any exceptions that occur inside the block are **handled safely**.

How It Works

When a file (or any object) is opened using with, Python calls two special methods of that object:

Method	Purpose
__enter__()	Called when entering the with block; initializes the resource.
__exit__()	Called automatically when leaving the with block, even if an exception occurs; used for cleanup.

This automatic handling eliminates the need for a manual close() call.

Example 1: Using with to Handle Files

```
with open('example.txt', 'r') as f:
  contents = f.read()
  print(contents)
```

Explanation:

- The open() function returns a file object.
- The with statement calls f.__enter__() to open the file.
- The file is used inside the block.
- When the block ends, f.__exit__() is called automatically, closing the file — even if an error occurs.

Equivalent Code Without with:

```
f = open('example.txt', 'r')
try:
    contents = f.read()
    print(contents)
finally:
    f.close()
```

Both versions do the same thing, but the with version is cleaner and less error-prone.

Example 2: Writing to a File Safely

```
with open('output.txt', 'w') as outfile:
    outfile.write("This is written safely using 'with' in Python.\n")
    outfile.write("File will close automatically after this block.")
```

Explanation:

The file output.txt is created (or overwritten) in write mode.

When the with block ends, the file is automatically closed — no explicit close() call is required.

Advantages of Using with

- **Automatic Cleanup:** Resources are released even when exceptions occur.
- **Simpler Syntax:** No need for explicit close() calls or finally blocks.
- **Readable and Safe:** Encourages clean, readable, and error-free code.
- **Extensible:** Works with any object implementing __enter__ and __exit__.

Table 6.2 Example Summary with ( ) for resource management

Use Case	Code Snippet	Purpose
Reading File	with open('data.txt', 'r') as f:	Automatically closes file after reading
Writing File	with open('output.txt', 'w') as f:	Ensures file closure even on error
Multiple Files	with open('a.txt') as a, open('b.txt') as b:	Manage multiple files safely
Custom Context	__enter__() / __exit__() methods	Build your own managed resources

6.12 DEBUGGING IN PYTHON

Debugging is the process of finding and fixing defects.

Common approaches:

- **Print Statements** – insert temporary print() calls to track variables.
- **Using the Debugger (pdb)**
- import pdb
- pdb.set_trace()

Allows step-by-step execution in the terminal.

- **IDE Debuggers** – VS Code, PyCharm, and IDLE provide breakpoints, watches, and variable inspectors.
- **Code Review and Unit Testing** – systematic testing reveals logical errors early.

6.13 LOGGING RUNTIME INFORMATION

Instead of printing messages, use the built-in **logging** module.

```
import logging
logging.basicConfig(filename='app.log', level=logging.INFO)

logging.info('Program started')
try:
   result = 10 / 0
except ZeroDivisionError:
   logging.exception('Division by zero encountered')
logging.info('Program ended')
```

app.log

```
INFO:root:Program started
ERROR:root:Division by zero encountered
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
INFO:root:Program ended
```

Logging is essential for diagnosing production errors without interrupting program execution.

6.14 CASE STUDY – LOGGING FILE ACCESS (Brief)

Logging is a technique used to **record events** that occur while a program runs, such as file openings, errors, or successful operations.

Python's **logging** module makes it easy to keep a record of such activities in a separate log file.

Example: Logging File Access

```
import logging


logging.basicConfig(
    filename='file_access.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)


def open_file(filename):
    try:
        with open(filename, 'r') as f:
            data = f.read()
            logging.info(f"File opened successfully: {filename}")
    except FileNotFoundError:
        logging.error(f"File not found: {filename}")
    except PermissionError:
        logging.error(f"Permission denied: {filename}")
```

Sample Output on Screen

Error: File not found: missing.txt

Contents of file_access.log

2025-10-28 17:10:03,502 - INFO - File opened successfully: notes.txt

2025-10-28 17:10:04,210 - ERROR - File not found: missing.txt

Explanation

- The **logging** module writes messages to a log file instead of printing them.
- Each log entry includes a **timestamp**, a **severity level**, and a **message**.
- Levels such as INFO, WARNING, and ERROR describe the importance of each event.
- Logging helps trace errors, monitor program activity, and maintain system reliability.

Key Points

- logging.basicConfig() initializes log settings.

- logging.info() records successful operations.
- logging.error() records failures.
- Logs are saved in **file_access.log** for later review.

Logging file access helps developers **track file operations and detect errors** automatically, improving debugging, transparency, and program maintenance.

6.15 SUMMARY

In this chapter, you learned how Python handles **errors** and **exceptions**, and how to write programs that can detect and recover from unexpected situations gracefully.

Key takeaways include:

- **Errors** are problems in the program that can prevent execution. They are of three types: **syntax**, **runtime**, and **logical errors**.
- **Syntax errors** are detected by Python before execution and must be corrected.
- **Runtime errors (exceptions)** occur while a program is running and can be handled using the try–except structure.
- The try block identifies risky code, while except specifies how to handle each exception type.
- The else block runs only when no exception occurs, and the finally block always runs for cleanup.
- The raise statement is used to generate exceptions deliberately; assert is used for testing conditions.
- Programmers can create **custom exceptions** by defining new classes that inherit from Exception.
- The with statement automates **resource management**, ensuring files and connections are closed properly.
- The **logging** module records program events and errors for analysis and debugging.
- Combining **exception handling**, **resource management**, and **logging** leads to reliable, maintainable, and professional-grade Python programs.

6.16 TECHNICAL TERMS

Error, Exception, Traceback, SyntaxError, RuntimeError, Logical Error,
try, except, else, finally, raise, assert, Custom Exception, Exception Hierarchy,
with Statement, Resource Management, Context Manager, Logging,
FileNotFoundError, ValueError, ZeroDivisionError, PermissionError, Debugging,
pdb (Python Debugger), Logging Levels (INFO, WARNING, ERROR, CRITICAL)

6.17 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the difference between syntax errors, runtime errors, and logical errors with examples.
2. Discuss the working of the try–except–else–finally structure with suitable code.
3. Describe how custom exceptions are created and used in Python.
4. What is the purpose of the with statement? Illustrate its use in file operations.

5. Explain the use of the logging module for debugging and tracking file access.
6. Write a short note on exception hierarchy and the importance of handling specific exceptions.
7. Discuss how debugging tools like pdb can be used to find logical errors.

Short-Answer Questions

1. Define try and except.
2. What is the difference between raise and assert statements?
3. List any four built-in exceptions in Python.
4. What happens if a file opened in read mode does not exist?
5. What is the purpose of the finally block?
6. How does with open() differ from using open() and close() manually?
7. Mention two advantages of using logging over print statements.

6.18 SUGGESTED READINGS

1. **Ljubomir Perković** – *Introduction to Computing Using Python: An Application Development Focus*, John Wiley & Sons, 2012.
2. **Reema Thareja** – *Python Programming: Using Problem-Solving Approach*, Oxford University Press.
3. **Mark Lutz** – *Learning Python*, O'Reilly Media.
4. **Eric Matthes** – *Python Crash Course*, No Starch Press.
5. **Al Sweigart** – *Automate the Boring Stuff with Python*, No Starch Press.

Dr. Neelima Guntupalli

LESSON- 07

CONDITIONAL STRUCTURES

AIMS AND OBJECTIVES

After completing this chapter, students will be able to:

- Understand how programs make decisions using conditional logic.
- Use relational and logical operators in expressions.
- Apply if, if–else, if–elif–else, and nested if statements.
- Write programs that choose between alternative actions.
- Distinguish between simple and compound conditions.
- Use indentation correctly to represent decision structures.

STRUCTURE

7.1 INTRODUCTION

Every non-trivial program must be able to *make decisions*. A decision statement allows a program to choose a course of action depending on whether a condition evaluates to *True* or *False*. Python's main decision-making construct is the if statement. Together with relational and logical operators, it forms the basis for *control flow*.

Example motivation:

```
temperature = 37.8
if temperature > 37:
    print("Fever detected")
```

The program executes the print() statement only when the condition temperature > 37 is *True*.

7.2 NEED FOR DECISION MAKING IN PROGRAMS

Sequential execution alone cannot handle situations requiring alternative outcomes. Real-life decisions—granting a loan, grading a student, or controlling a machine—depend on specific conditions.

Without Decision Control

```
balance = 500
print("Withdrawal allowed")
```

The message appears even when balance < 0.

With Decision Control

```
balance = 500
if balance > 0:
   print("Withdrawal allowed")
else:
   print("Insufficient funds")
```

Decision control thus makes programs *intelligent* and *context-dependent*.

7.3 BOOLEAN EXPRESSIONS AND RELATIONAL & LOGICAL OPERATORS

A Boolean expression evaluates to one of the two truth values: True or False. Python treats True as 1 and False as 0 in numeric contexts.

Table 7.1 Relational Operators

Operator	Meaning	Example	Result
==	equal to	5 == 5	True
!=	not equal to	7 != 2	True
<	less than	3 < 9	True
>	greater than	8 > 10	False
<=	less than or equal to	5 <= 5	True
>=	greater than or equal to	9 >= 5	True

Table 7.2 Logical Operators

Operator	**Description**	**Example**	**Result**
and	True if both operands are True	(x > 0) and (x < 10)	True if x between 0 and 10
or	True if at least one operand is True	(x == 0) or (y == 0)	True if any zero
not	Reverses truth value	not(x > 5)	True if x ≤ 5

Relational operators are often combined with logical operators to form compound conditions.

Example

```
age = 20
if age >= 18:
   print("Adult")
```

Example

```
x = 5
y = 10
print((x < 10) and not(y < 5 or x == 7))
```

Evaluation order ensures predictable results.

7.4 SIMPLE IF STATEMENT

The if statement in Python is one of the conditional statements that is used the most frequently in programming languages. In this way, it determines whether or not particular statements are required to be executed. It performs a check to determine whether a particular condition is satisfied; if the condition is satisfied, the set of code included within the "if" block will be run; otherwise, it will not be executed.

Syntax:

if (EXPRESSION = = TRUE) :

if- Block of code

Next statement after Block of code is executed.

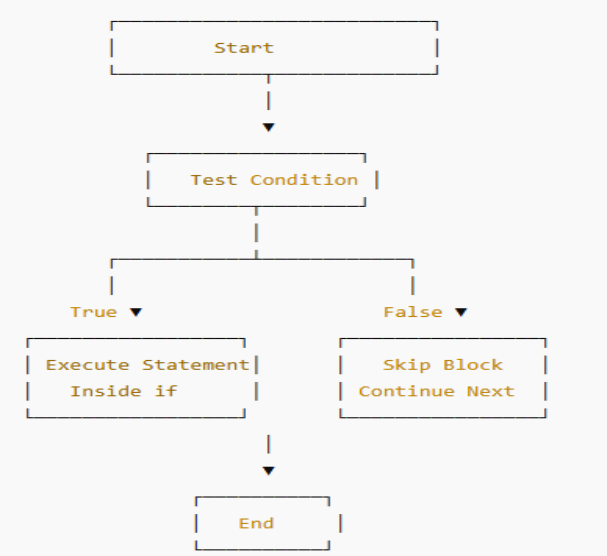


Fig 7.1. Simple if statement

In the syntax presented above,

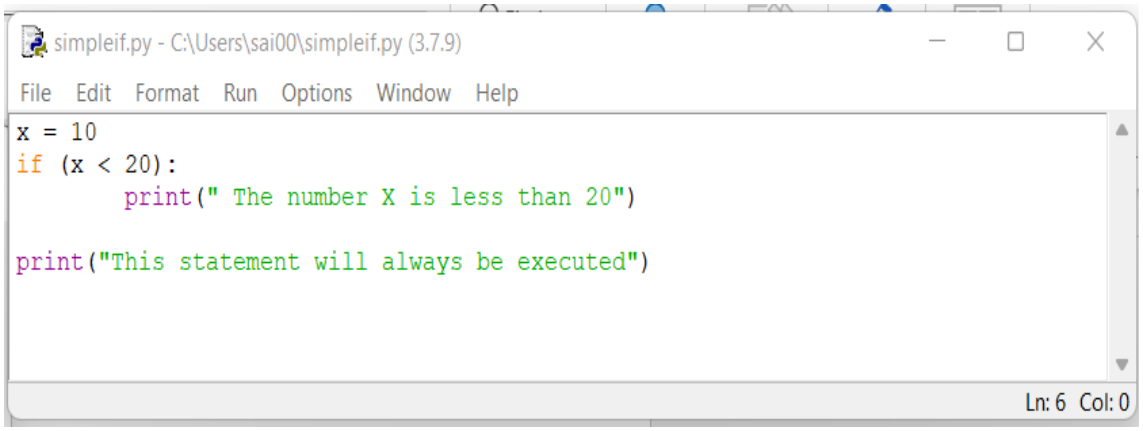
- if the expression "EXPRESSION = = TRUE" is successfully executed, then the conditional block of code will be run

- Otherwise, the statement that comes after the conditional block of code will be executed.

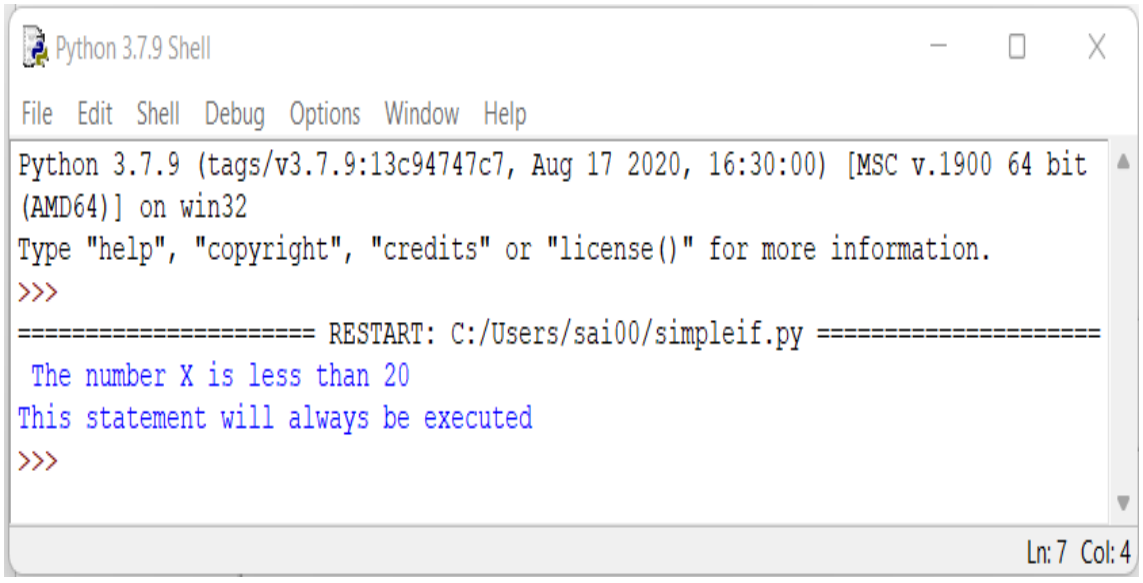
The flow chart of if statement is shown in Figure 7.1.

- If you look at the flowchart that was just presented, you will notice that the controller will first arrive at an if condition and then evaluate the condition.
- If the condition is true, then the statements will be executed.
- if it is not true, then the code that is present outside the block will be executed.

Example: 1

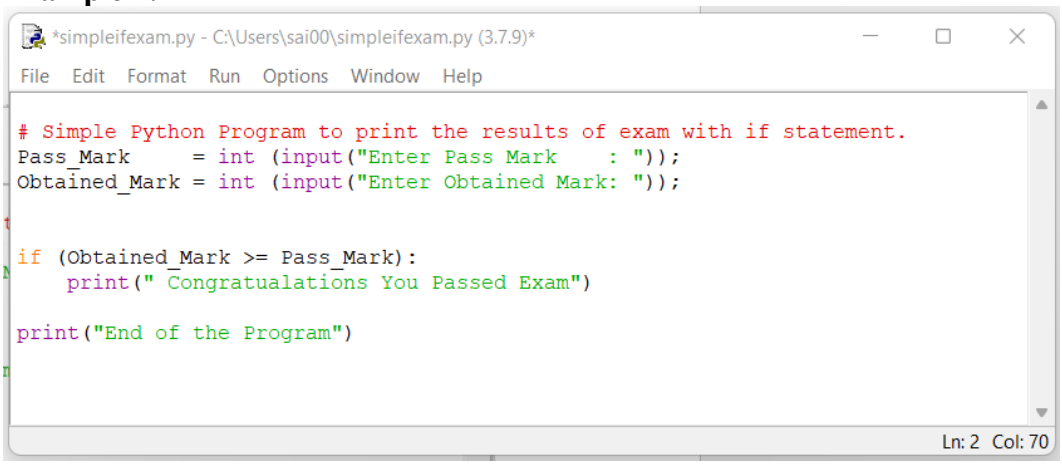


Output:

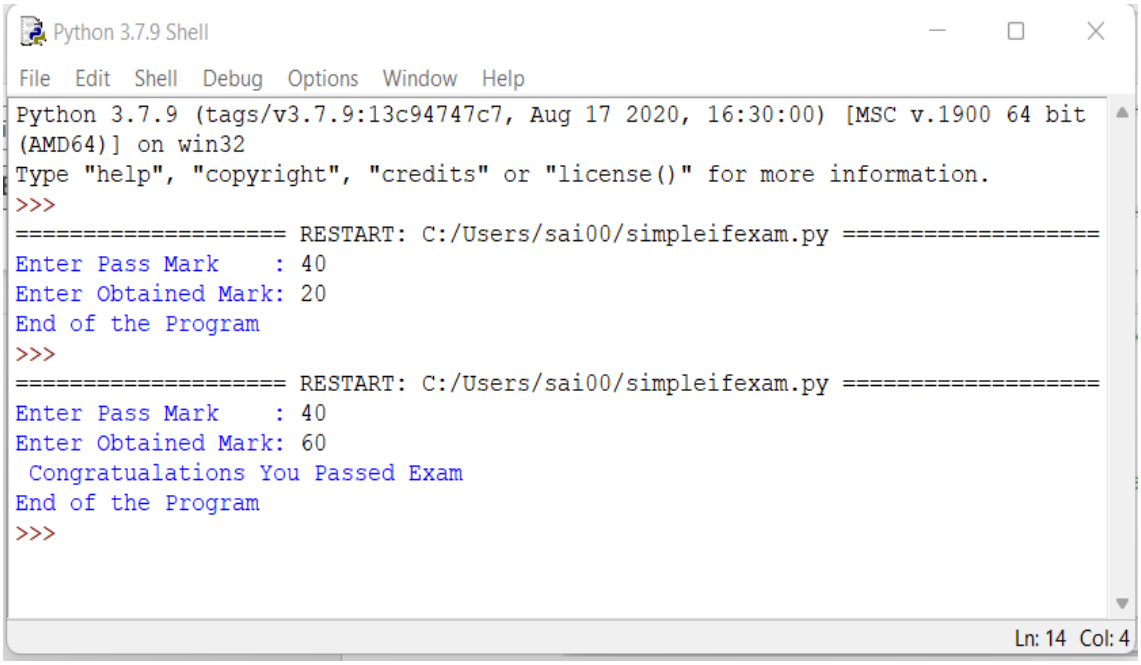


The above code tests the condition "x<20." If the test is successful, a block of code will be executed, as really seen in the output, and finally the last line, "**This statement will always be executed,**" will be executed. This statement is also clearly displayed in the output.

Example 2:



Output:



The code condition (Obtained_Mark > = Pass_Mark) is tested in the previous example; if it passes, the if-block will be executed. The code is executed twice. The first time, the condition is not met (20 < 40), and the final message, "**End of the Program**," is shown. Nevertheless, the second attempt met the success requirement (i.e., 60 > 40), printed "**Congratulations on Passing the Exam**," and showed the final message, "**End of the Program.**"

7.5 IF–ELSE STATEMENT

if-else statements

The Boolean expression is evaluated by the if-else statement. The code in the "if" block will be executed if the condition is TRUE; otherwise, the code in the "else" block will be executed.

Syntax:

If (EXPRESSION == TRUE):

If-Statement (Body of the block)

else:

else-Statement (Body of the block)

When the syntax (EXPRESSION = = TRUE) in the following example is successfully executed, a block of code will be executed if it is not, otherwise it will be executed.

The flow chart of if-else statement is shown in Figure 7.2.

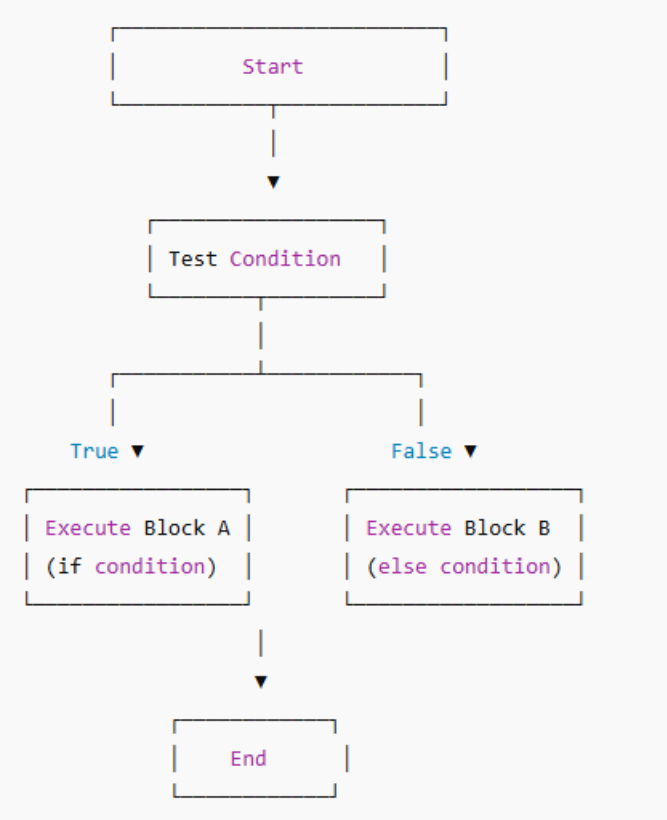
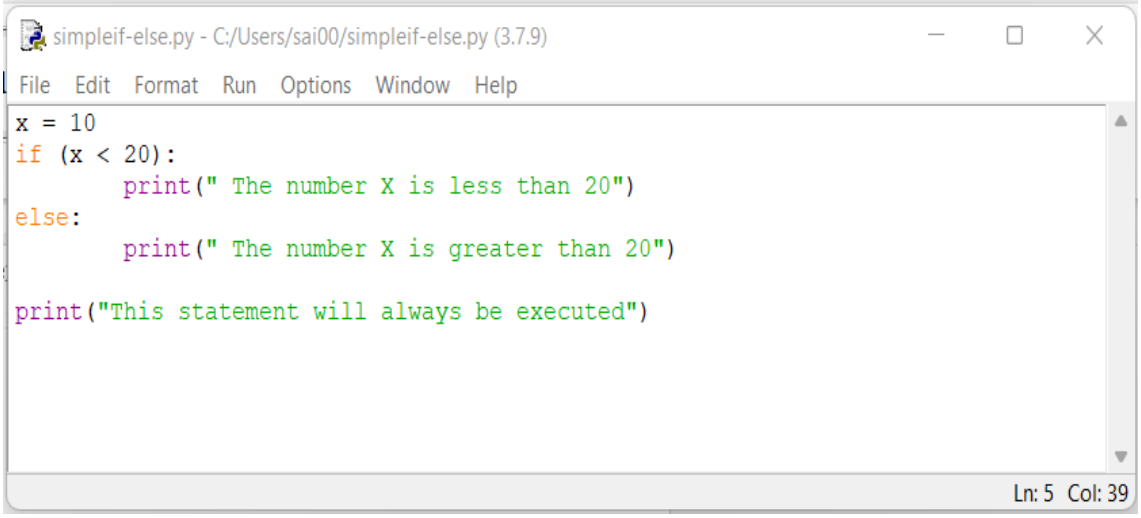


Fig 7.2. Flow Chart of if-else Statement

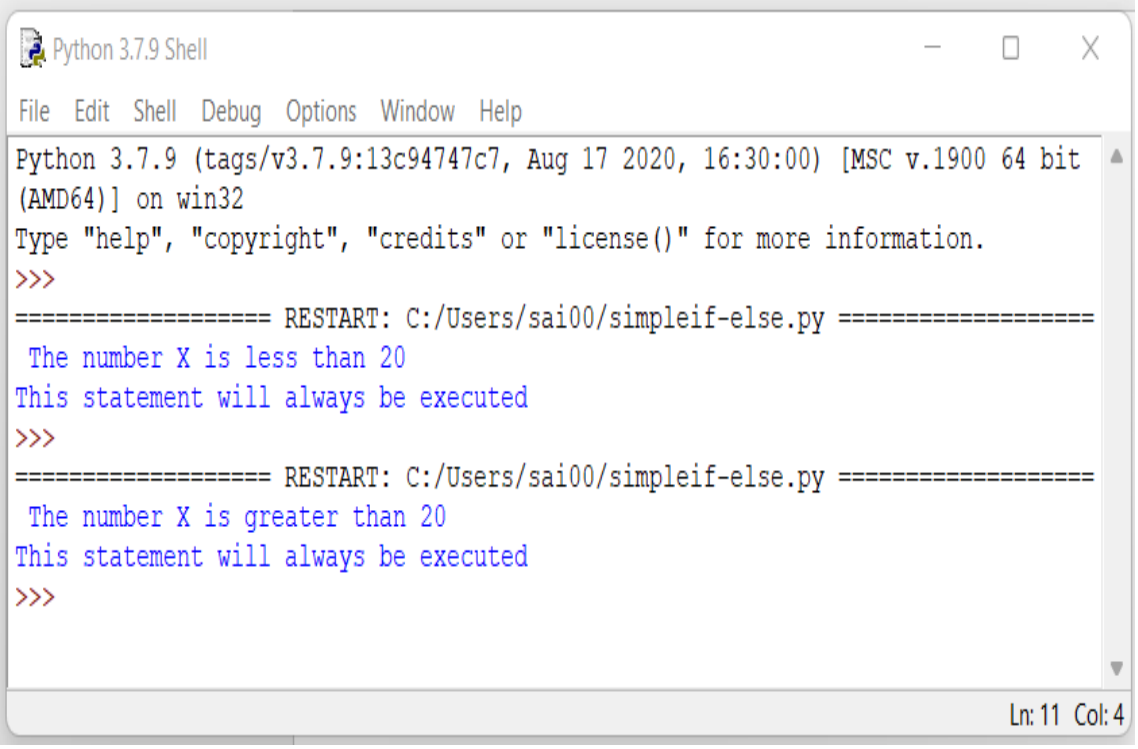
According to the flow chart above, the controller will first reach the if condition and determine if the condition is true. If it is, the statements in the if block will then be run; if not, the "else" block will be executed, and finally the remaining code that is included outside the "if-else" block will be executed.

Example: 1

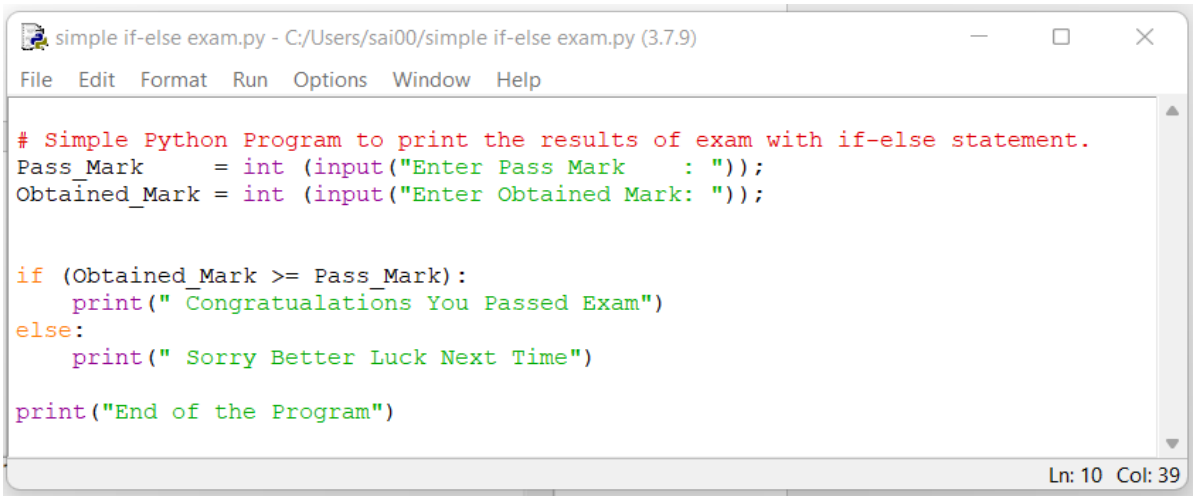


The condition (x<20) is tested twice in the code above. The first time it is run, if it is successful, a block of code will be executed, as we can see in the output. Finally, the final statement, "**This statement will always be executed**," is executed, and this is also clearly displayed in the output. Nevertheless, the second run condition failed by evaluating x=30, executing the else-Block of code, and generating the output "**X is greater than 20**." The final statement, "**This statement will always be executed**," is finally carried out and is likewise displayed in the output.

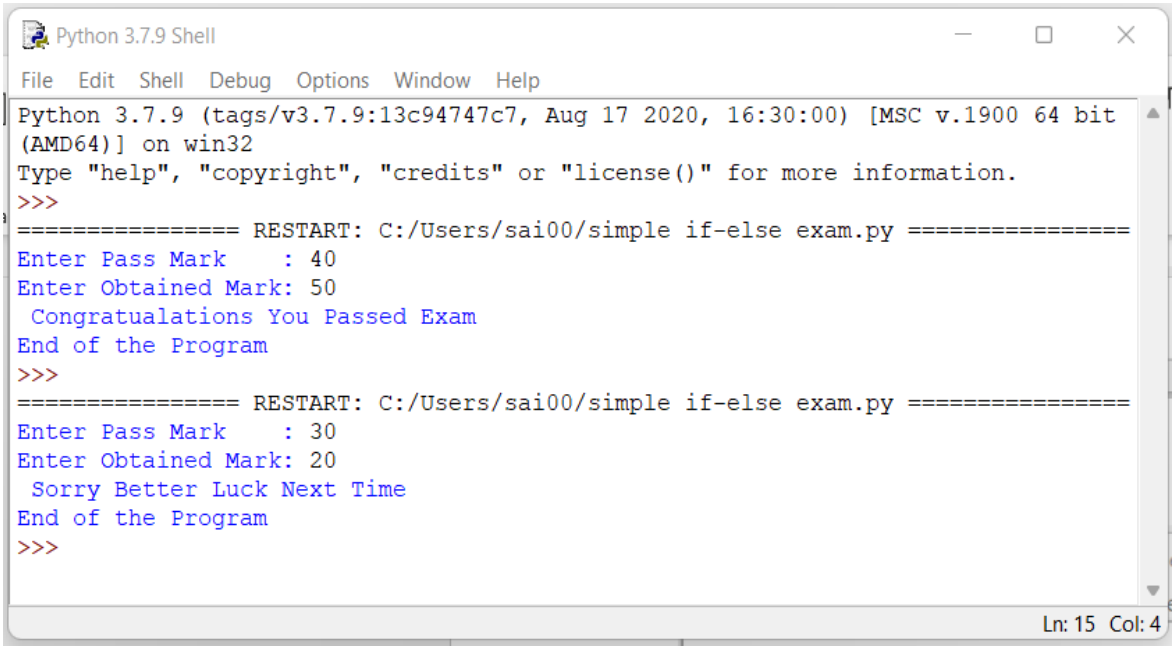
Output:



Example2:



Output:



The code condition (Obtained_Mark > = Pass_Mark) is tested in the previous example; if it passes, the if-block will be executed. The code is executed twice. The first time, if the condition is met (i.e., 50 > 40), the message "Congratulations You Passed Exam" is displayed, and the final phrase, "End of the Program," is printed. Nevertheless, the second time around, the condition failed (20 < 30), printing "Sorry, Better Luck Next Time" and displaying the last sentence, "End of the Program."

7.6 ELIF STATEMENTS

"elif" statements are an additional type of conditional statement in Python. The "elif" statement checks for multiple conditions only in the event that the supplied condition is false. The sole distinction between it and a "if-else" expression is that the condition will be checked in "elif" rather than "else."

Syntax:

if (EXPRESSION-1 = = TRUE):

If-Statement (Body of the block)

elif(EXPRESSION-2 = = TRUE):

elif-Statement (Body of the block)

elif(EXPRESSION-3 = = TRUE):

elif-Statement (Body of the block)

else:

else-Statement (Body of the block)

- In the above syntax (EXPRESSION-1 = = TRUE) is executed successfully then if-Block of code will be executed
- otherwise (EXPRESSION-2 = = TRUE) is tested, if it is executed successfully then elif- Block of code related EXPRESSION-2 will be executed
- otherwise (EXPRESSION-3 = = TRUE) is tested, if it is executed successfully then elif- Block of code related EXPRESSION-3 will be executed otherwise else-Block of code will be executed.

The flow chart of else-if- lader statement is shown in Figure 7.3.

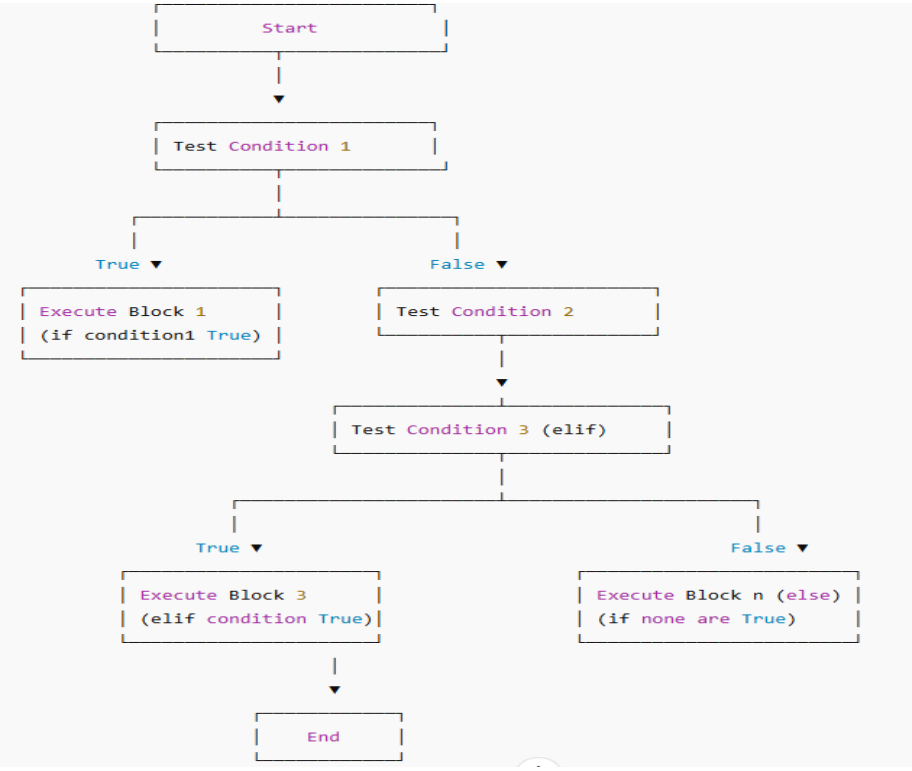
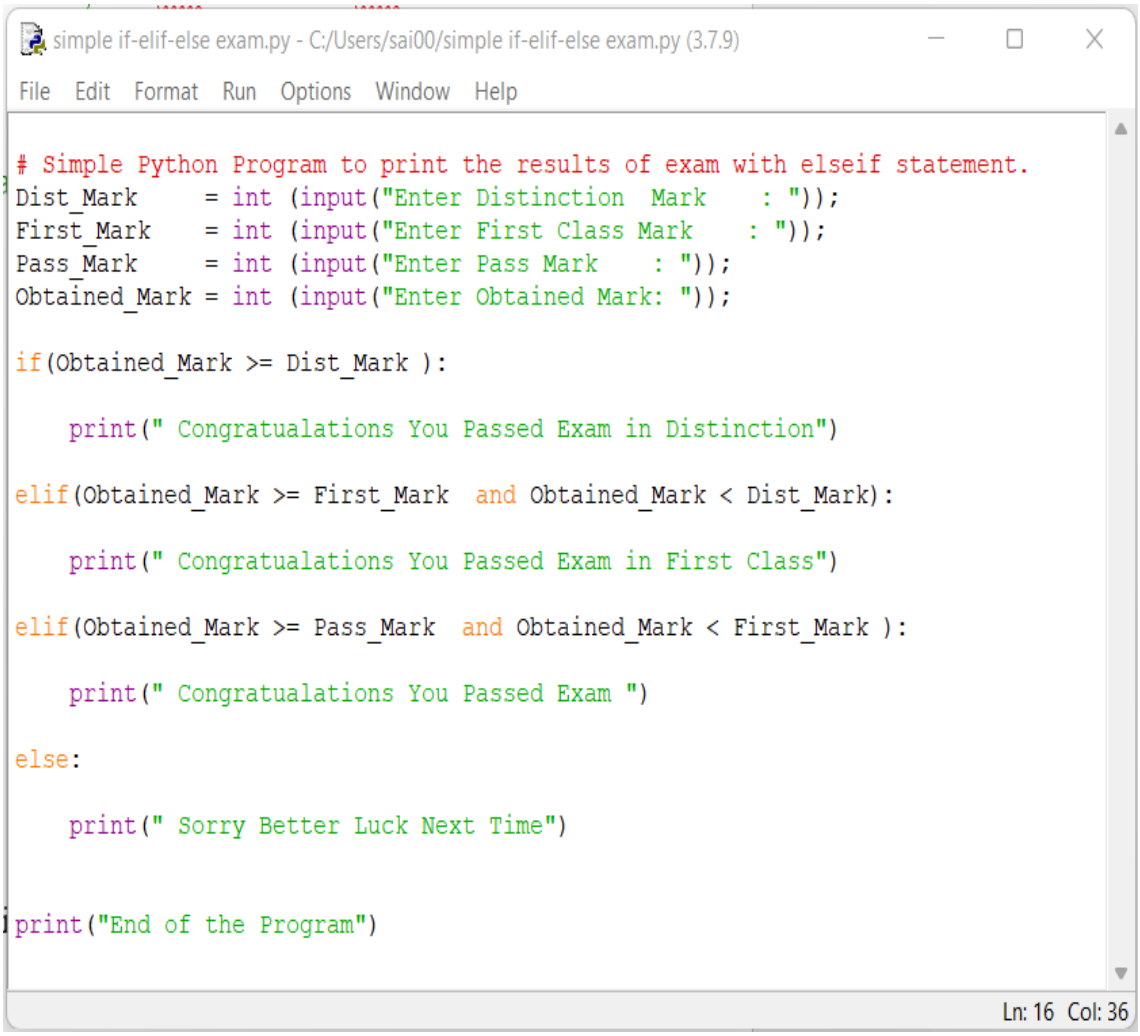


Fig 7.3. Flow Chart of else-if ladder Statement

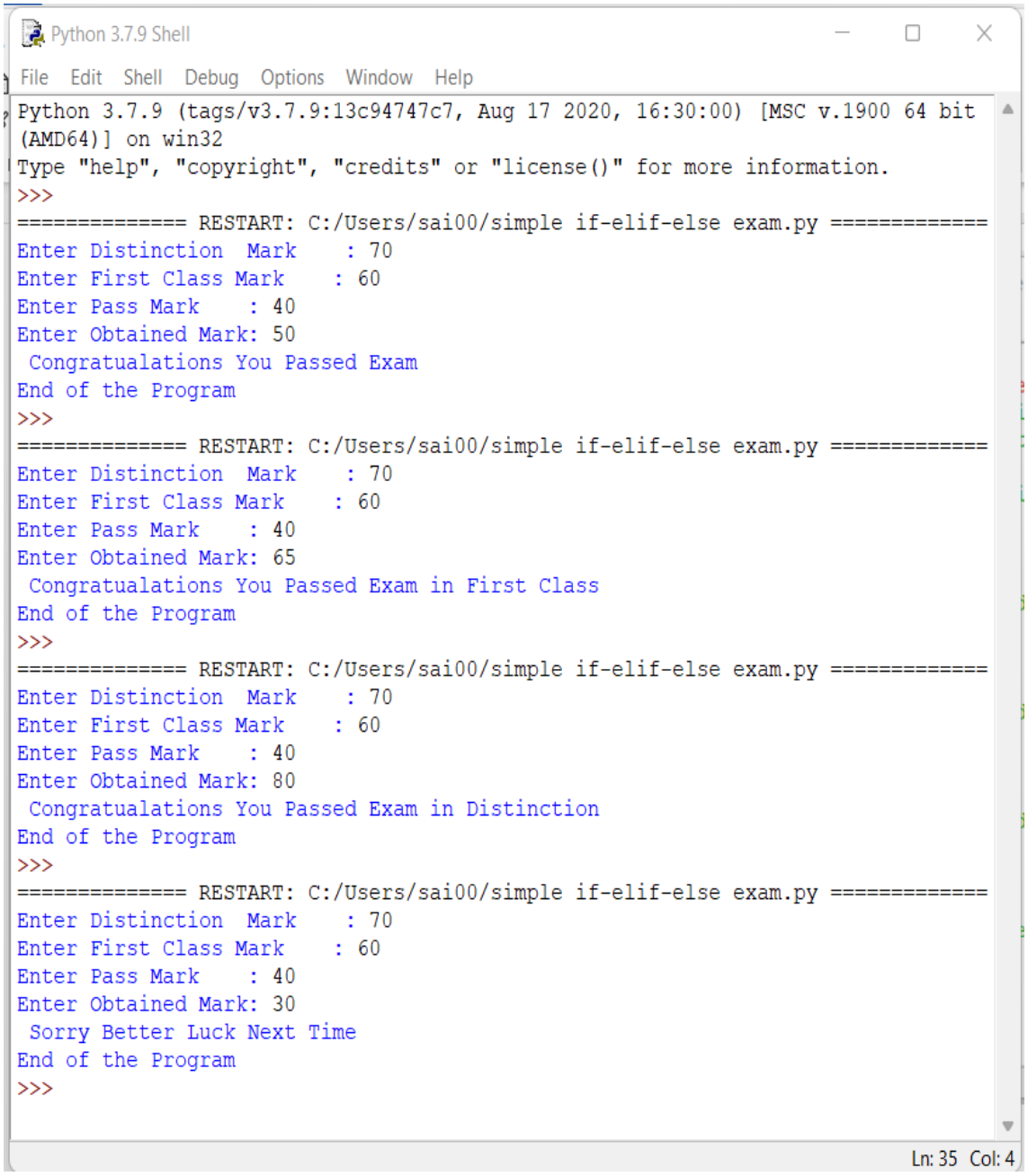
Example:

In the code below, the condition (Obtained_Mark >= Dist_Mark) is tested; if it is successful, the if-block of code is executed; otherwise, the following succeeding blocks are executed based on the criteria; otherwise, the else statement and the end statement are executed. The code is executed four times; the first time the condition is met (i.e., 50 > 40), the message "**Congratulations You Passed Exam**" is displayed, and the last statement, "**End of the Program**", is printed. However, the second time run condition (65 > 60) is successful and prints "**Congratulations You Passed Exam in First Class**" before displaying the last line, "End of the Program". Similarly, in the third run, the requirement (i.e., 80 > 70) is met, and the message "**Congratulations You Passed Exam in Distinction**" is displayed, followed by the final sentence "End of the Program". During the last run, if the condition (i.e., 30 < 40) is not met, the else block is activated and the message "**Sorry, Better Luck Next Time**" is written. The last statement displayed is "**End of the Program**".

Example:



Output:



7.7 NESTED IF-ELSE STATEMENTS

Nested "if-else" statements indicate that one "if" or "if-else" statement is contained within another if or if-else block. Python has this feature as well, which allows us to verify several conditions in a single application.

Syntax:

if (EXPRESSION-1 = = TRUE):
    if (EXPRESSION-2 = = TRUE):
        Inner-If-Statement (Body of the block)

```
        else:
            Inner-else-Statement (Body of the block)
    else:
        Outer-else-Statement (Body of the block)
```

The syntax used above obviously shows that the if block will include another if block, and so on. If block can have 'n' number of if blocks within it.

- In the below flow chart, (EXPRESSION-1 = = TRUE) is executed successfully, then (EXPRESSION-2 = = TRUE) is tested;
- if it is executed successfully, then if- Block of code related EXPRESSION-2 will be executed;
- otherwise, Block of code related EXPRESSION-2 will be executed; otherwise, Block of code related EXPRESSION-1 will be executed.

The Flow Chart of nested if Statement is shown in Figure 7.4.

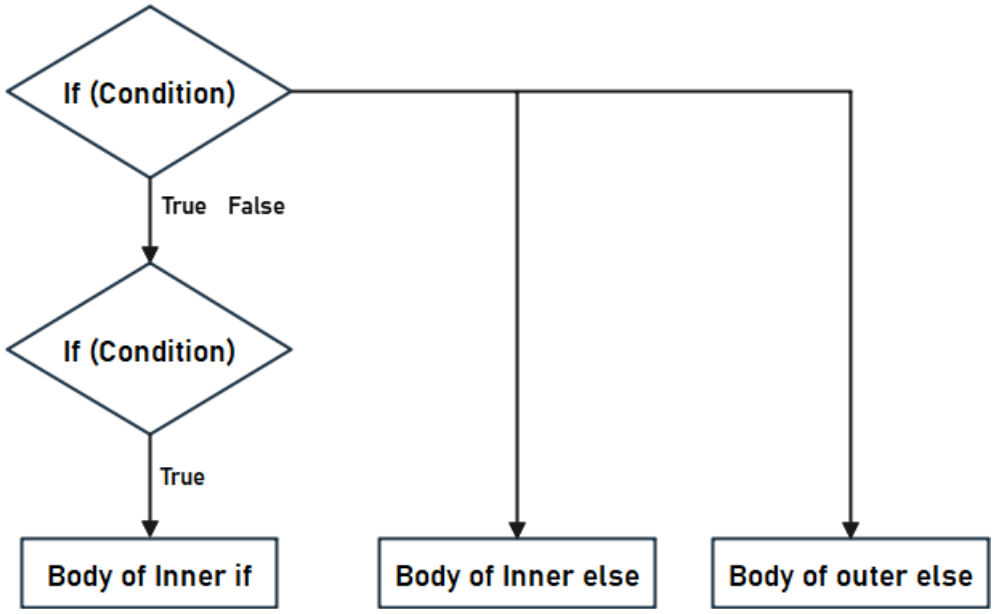
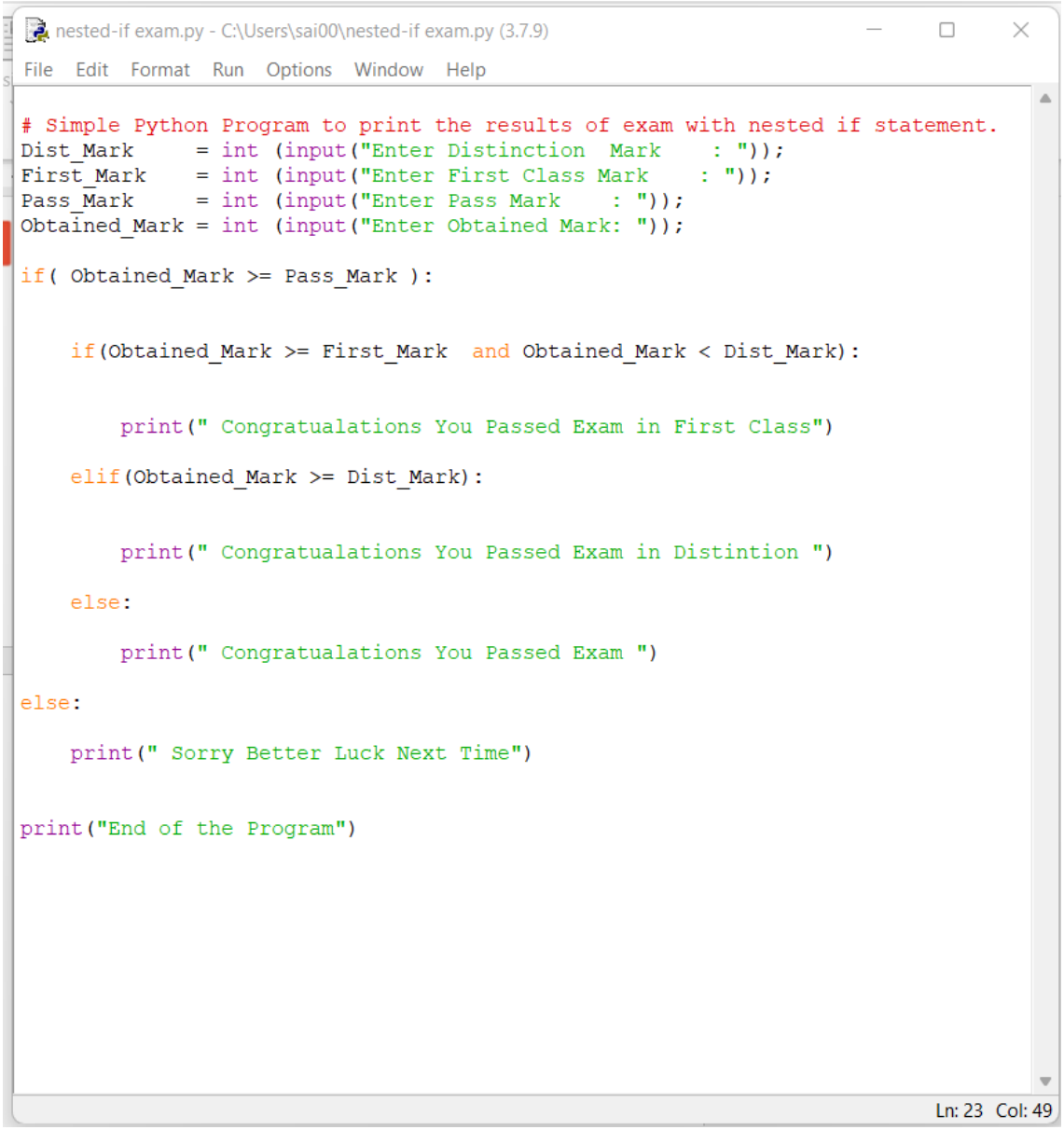


Fig 7.4. Flow Chart for nested-if Statement

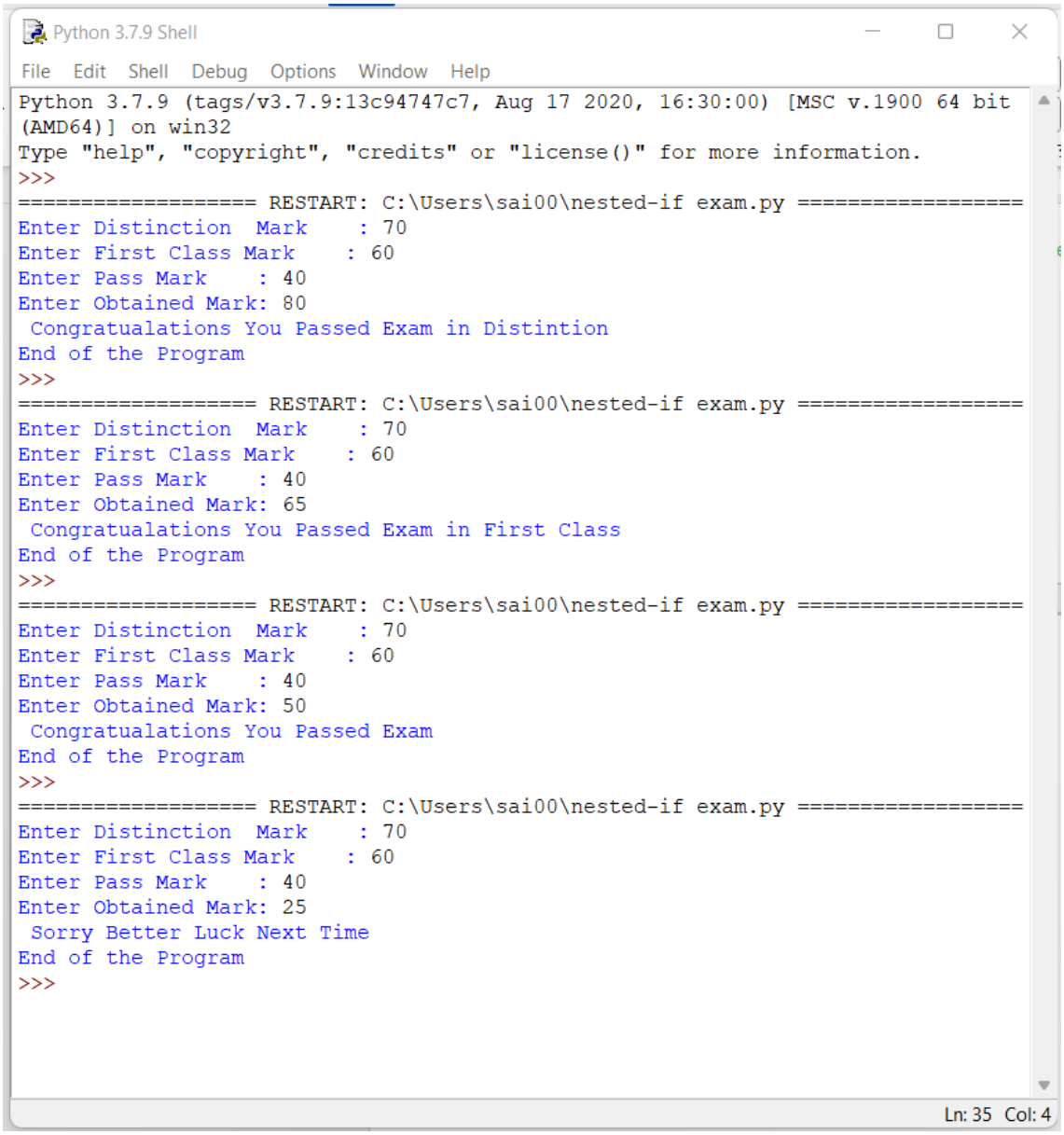
Example:

In the code below, the condition (Obtained_Mark > = Pass_Mask) is tested. If it is successful, the inner if-statement (Obtained_Mark > = First_Mask and Obtained_Mark < Dist_Mask) is tested. If it is successful, the block-related inner condition is executed. Otherwise, the block-related inner condition is executed. Otherwise, the else block from the outer condition is executed. The ensuing blocks are run based on the circumstances; otherwise, the else statement is executed, followed by the end statement.



The code is executed four times. The first time, the condition is successful (i.e., 80 > 40), and the second time, the condition is likewise successful (i.e., 80 > 70), and the message "**Congratulations You Passed Exam in Distinction**" is displayed, followed by the last statement, "**End of the Program**". However, the condition is successful the second time (65>40) and then tested (65>60) and printed "**Congratulations You Passed Exam in First Class**" and displayed the last statement, "End of the Program". Similarly, in the third run, the condition is successful (i.e., 50 > 40), and then tested (i.e., 50 > 40), which is successful and prints "**Congratulations You Passed Exam**" and displays the last statement, "**End of the Program**". If the condition is not met (i.e., 25 < 40), the else block is activated and the message "**Sorry, Better Luck Next Time**" is written. The last statement displayed is "**End of the Program**".

Output:



7.8 CONDITIONAL EXPRESSIONS (TERNARY OPERATOR)

In some situations, it is useful to make a simple decision in **a single line** rather than writing a complete if–else structure.Python provides the **conditional expression**, also known as the **ternary operator**, for such cases.

It evaluates a condition and **returns one of two values** depending on whether the condition is **True** or **False**.

Syntax

<value_if_true> if <condition> else <value_if_false>

This expression is evaluated from left to right.

- If the condition is **True**, Python returns <value_if_true>.

- If the condition is **False**, it returns <value_if_false>.

Example – Finding the Larger Number

```
a = 25
b = 40
larger = a if a > b else b
print("Larger number is:", larger)
```

Output

```
Larger number is: 40
```

Example – Even or Odd

```
num = int(input("Enter a number: "))
print("Even") if num % 2 == 0 else print("Odd")
```

Output

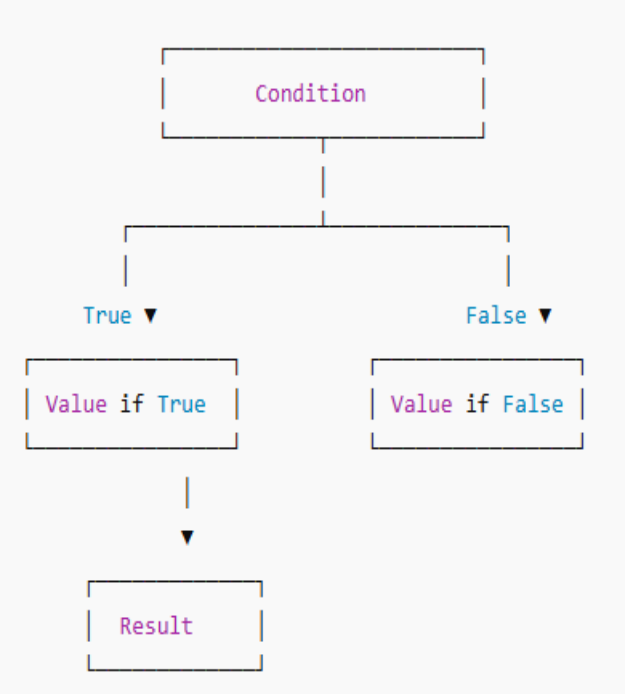
```
Enter a number: 15
Odd
```

Example – Voting Eligibility

```
age = int(input("Enter your age: "))
status = "Eligible" if age >= 18 else "Not Eligible"
print(status, "to vote.")
```

Output

```
Enter your age: 20
Eligible to vote.
```



7.5 Flowchart for Conditional Expression

Advantages

- **Compact and expressive for simple decisions.**
- **Improves code readability when used judiciously.**
- **Can be nested for complex choices, though readability may suffer.**

7.9 PRACTICAL EXAMPLES

This section demonstrates how the various decision-making statements (if, if–else, if–elif–else, and nested if) can be applied to solve real-world problems.

Each example shows both code and expected output to clarify control flow.

BMI (Body Mass Index) Calculator

Objective:

To implement a function myBMI() that takes a person's height (in inches) and weight (in pounds) and computes their Body Mass Index (BMI).

The BMI is then classified as *Underweight*, *Normal*, or *Overweight* based on standard health guidelines.

Formula

$$\text{BMI} = \frac{\text{weight} \times 703}{(\text{height})^2}$$

Where:

- weight → person's weight in pounds
- height → person's height in inches
- 703 → conversion factor for imperial units

Program

```
def myBMI(height, weight):
    """Compute and classify Body Mass Index (BMI)."""
    bmi = (weight * 703) / (height ** 2)
    print("BMI value:", round(bmi, 2))

    if bmi < 18.5:
        print("Underweight")
    elif bmi < 25:
        print("Normal")
    else:
```

```
    print("Overweight")

# Example usage:
h = float(input("Enter height (in inches): "))
w = float(input("Enter weight (in pounds): "))
myBMI(h, w)
```

Explanation

1. The function myBMI() takes two inputs — height and weight.
2. The BMI is calculated using the formula.
3. Using an if–elif–else structure:
    - If BMI < 18.5 → prints Underweight.
    - If 18.5 ≤ BMI < 25 → prints Normal.
    - If BMI ≥ 25 → prints Overweight.
4. The round() function is used to format the BMI to two decimal places.

Sample Runs

Example 1:

```
Enter height (in inches): 65
Enter weight (in pounds): 110
BMI value: 18.3
Underweight
```

Example 2:

```
Enter height (in inches): 68
Enter weight (in pounds): 150
BMI value: 22.8
Normal
```

Example 3:

```
Enter height (in inches): 63
Enter weight (in pounds): 165
BMI value: 29.2
Overweight
```

7.10 SUMMARY

- Decision control statements make a program choose **specific actions** based on conditions.
- The constructs if, if–else, if–elif–else, and nested if allow conditional branching.
- Boolean expressions use **relational** and **logical operators** and evaluate to True or False.
- The **conditional (ternary)** expression provides a concise single-line decision.
- Correct **indentation** defines program structure in Python.
- These statements add **flexibility and intelligence** to programs.

7.11 TECHNICAL TERMS

Decision Control, Conditional Statement, Boolean Expression, Relational Operator, Logical Operator, Indentation, if Statement, Elif, else, Nested if, Conditional Expression / Ternary Operator, Branching, Truth Table, Compound Condition, Control Flow, Condition Testing.

7.12 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the need for decision control in a program with examples.
2. Describe the working of if, if–else, and if–elif–else statements.
3. Write a program to determine whether a year is a leap year.
4. Discuss how logical operators help in forming compound conditions.
5. Explain the role of indentation in decision structures.
6. What is a conditional expression? Give suitable examples.

Short Notes

1. Boolean expressions and relational operators.
2. Range checking using if–else.
3. Flow of control in if–elif–else.
4. Difference between nested if and if–elif–else.
5. Syntax and use of the else block.

Programming Exercises

1. Accept three numbers and display the largest.
2. Write a menu-driven calculator using if–elif–else.
3. Determine whether a character is a vowel, consonant, or symbol.
4. Check if a given number is divisible by 5 and 11.
5. Print student grade based on marks using decision statements.
6. Use a conditional expression to find the smaller of two numbers.

7.13 SUGGESTED READINGS

1. **Ljubomir Perković**, *Introduction to Computing Using Python: An Application Development Focus*, Wiley (2012).
2. **Reema Thareja**, *Python Programming Using Problem-Solving Approach*, Oxford University Press.
3. **Mark Lutz**, *Learning Python*, O'Reilly Media.
4. **Eric Matthes**, *Python Crash Course*, No Starch Press.
5. **Al Sweigart**, *Automate the Boring Stuff with Python*, No Starch Press.

LESSON- 08

CONTROL STRUCTURES

AIMS AND OBJECTIVES

After completing this chapter, students will be able to:

- Understand how Python executes statements repeatedly using loops.
- Distinguish between **definite iteration (for loop)** and **indefinite iteration (while loop)**.
- Use different **loop patterns**: iteration, counter, accumulator, and nested loops.
- Apply loops with **two-dimensional lists**.
- Control iteration flow using **break**, **continue**, and **pass** statements.
- Develop real-world programs involving repetition and data aggregation

STRUCTURE

8.1 INTRODUCTION

Programs often require repeating a sequence of statements multiple times.

For example:

- **Counting from 1 to 10,**
- **Summing a list of numbers,**
- **Printing elements of a list, or**
- **Reading data until the user quits.**

Writing such repetitive code manually is inefficient and error-prone. To handle repetition, Python provides looping structures, which allow a block of code to execute repeatedly until a condition changes.
Python has two major looping statements:

Type	Structure	When Used
for loop	**Iterates over a sequence or range**	**When number of iterations is known**
while loop	**Repeats while a condition is true**	**When number of iterations is unknown**

8.2 THE FOR LOOP AND ITERATION PATTERNS

The for loop is used to execute a block of code a fixed number of times or once for each element in a sequence such as a list, tuple, or string.
Syntax

```
for variable in sequence:
    statement(s)
```

Each element in the sequence is assigned to the loop variable, and the indented block executes once per element.

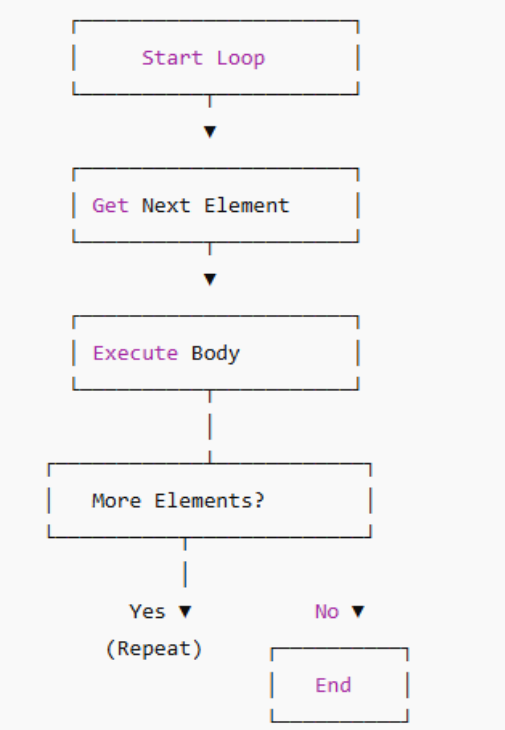


Fig 8.1 Flowchart – Iteration Loop

8.2.1 Loop Pattern – Iteration Loop

The simplest use of the for loop is to iterate through all items of a sequence.

Example – Iterating Through a List

```
for name in ['Ravi', 'Lina', 'Arun']:
    print("Hello,", name)
```

Output

Hello, Ravi

Hello, Lina

Hello, Arun

This pattern is known as an iteration loop — it processes every item in a collection.

Example:

```
animals = ['fish', 'cat', 'dog']
 for animal in animals:
        print(animal)
```

Output

fish

cat

dog

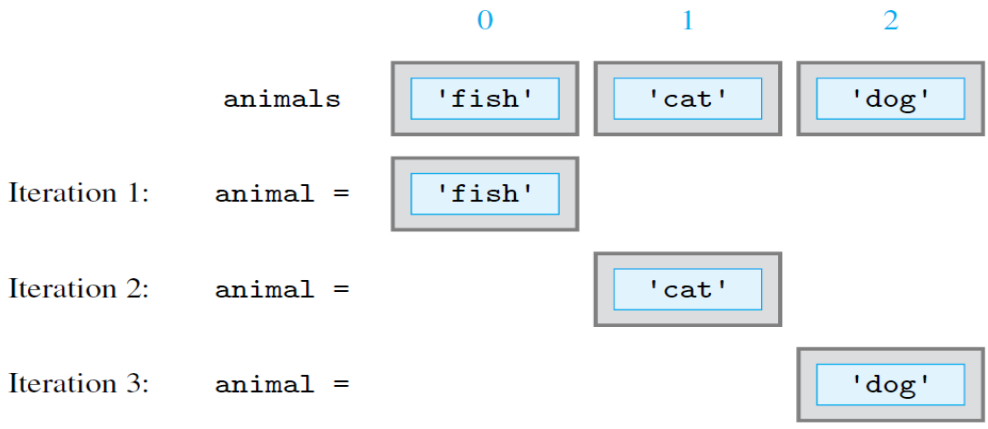


Figure 6.2 Iteration through a list.

8.2.2 Loop Pattern – Counter Loop

A counter loop runs a specific number of times, typically using the range() function.

Syntax of range()

range(start, stop, step)

- **start – beginning value (inclusive)**
- **stop – ending value (exclusive)**
- **step – increment (default 1)**

Example -Counting Iterations

```
for i in range(1, 6):
   print("Iteration number:", i)
```

Output

Iteration number: 1

Iteration number: 2

Iteration number: 3

Iteration number: 4

Iteration number: 5

8.2.3 Loop Pattern – Accumulator Loop

An accumulator loop collects or aggregates data during each iteration.

Example – Sum of Numbers

```
total = 0
for num in range(1, 6):
   total += num
print("Sum =", total)
```

Output

```
Sum = 15
```

The variable total is called the accumulator.

8.2.4 Accumulating Different Data Types

Loops can accumulate strings, lists, or concatenated results.

Example – String Accumulation

```
sentence = ""
for word in ["Python", "is", "powerful"]:
   sentence += word + " "
print(sentence.strip())
```

Output

```
Python is powerful
```

Example 8.5 – List Accumulation

```
squares = []
for i in range(1, 6):
    squares.append(i * i)
print(squares)
```

Output

```
[1, 4, 9, 16, 25]
```

8.2.5 Loop Pattern – Nested Loop

A nested loop is a loop inside another loop.

It is used for two-dimensional data (tables, matrices, grids, etc.).

Example – Multiplication Table

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i * j, end=' ')
    print()
```

Output

```
1 2 3
2 4 6
3 6 9
```

Each iteration of the outer loop triggers a full pass of the inner loop.

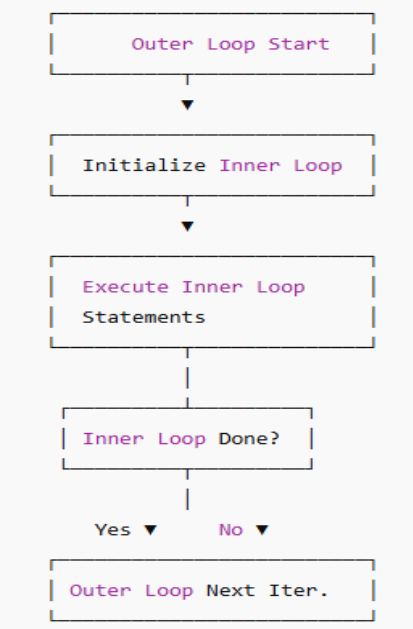


Figure 8.2 Flowchart – Nested Loop

8.3 TWO-DIMENSIONAL LISTS

Python can represent tabular data using lists of lists.

8.3.1 Concept of Two-Dimensional Lists

Example

```
matrix = [
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
]
print(matrix[1][2])
```

Output

```
6
```

8.3.2 Nested Loop Pattern with 2-D Lists

Example

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
   for value in row:
      print(value, end=' ')
   print()
```

Output

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

8.4 THE WHILE LOOP

A while loop executes statements repeatedly as long as a condition is true.

Syntax

```
while condition:
   statement(s)
```

Example – Simple Counting Loop

```
count = 1
while count <= 5:
   print("Count =", count)
```

```
    count += 1
```

Output

```
Count = 1
Count = 2
Count = 3
Count = 4
Count = 5
```

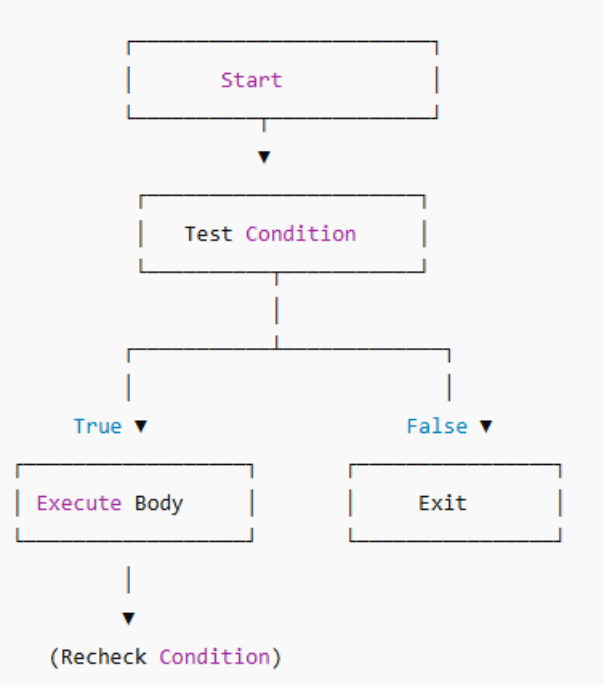


Fig 8.3 Flowchart – while Loop

Example – Factorial Using while

```
n = int(input("Enter a number: "))
fact = 1
i = 1
while i <= n:
    fact *= i
    i += 1
print("Factorial =", fact)
```

8.5 MORE LOOP PATTERNS

8.5.1 Iteration Pattern – Sequence Loop

Iterates directly through each element of a collection.

Example

```
for letter in "PYTHON":
   print(letter)
```

Output

P

Y

T

H

O

N

8.5.2 Infinite Loop

An infinite loop runs forever unless stopped manually or by a break condition.

Example

```
while True:
   name = input("Enter name (type 'stop' to exit): ")
   if name == 'stop':
      break
   print("Hello,", name)
```

8.5.3 Loop-and-a-Half Pattern

Used when the termination condition appears **in the middle** of the loop.

Example

```
while True:
   num = int(input("Enter number (-1 to quit): "))
   if num == -1:
      break
   print("Square:", num ** 2)
```

8.6 ADDITIONAL ITERATION CONTROL STATEMENTS

8.6.1 break Statement

The break statement **terminates** the loop immediately.

Example

```
for i in range(1, 10):
   if i == 5:
```

```
        break
    print(i)
```

Output

1

2

3

4

8.6.2 continue Statement

Skips the current iteration and moves to the next.

Example

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output

1

2

4

5

8.6.3 pass Statement

Used as a **placeholder** when a statement is required syntactically but no action is desired.

Example

```
for i in range(5):
    if i < 3:
        pass
    else:
        print(i)
```

Output

3

4

8.7 SUMMARY

- **Loops** repeat a block of code multiple times.
- The **for loop** handles definite iteration; the **while loop** handles indefinite iteration.
- Common **loop patterns**:
  - *Iteration Loop* – processes each element.
  - *Counter Loop* – runs for a set number of iterations.
  - *Accumulator Loop* – builds up totals or collections.
  - *Nested Loop* – processes multi-dimensional structures.
- **Control statements** (break, continue, pass) modify normal loop flow.
- Two-dimensional lists are handled using **nested loops**.

8.8 TECHNICAL TERMS

Iteration,
Loop,
Counter Variable,
Accumulator,
Nested Loop,
Two-Dimensional List,
Infinite Loop,
Loop-and-a-Half Pattern,
break, continue, pass,
Definite Iteration,
Indefinite Iteration.

8.9 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the difference between for and while loops in Python.
2. Describe and illustrate accumulator and counter loop patterns.
3. Explain nested loops with an example.
4. How are two-dimensional lists processed using nested loops?
5. What is an infinite loop? How can it be avoided?
6. Discuss how break and continue statements affect loop execution.

Short Notes

1. Sequence loop.
2. Loop control statements.
3. Loop and a half pattern.
4. Role of the accumulator variable.
5. Use of range() in for loops.

Programming Exercises

1. Write a program to print the first 10 natural numbers.
2. Compute the factorial of a given number using a while loop.
3. Generate a list of squares using an accumulator loop.
4. Display multiplication tables (1 to 5) using nested loops.

5. Find the sum of digits of a number using a while loop.
6. Write a program to display even numbers from 1 to 50.

8.10 SUGGESTED READINGS

1. **Ljubomir Perković**, *Introduction to Computing Using Python: An Application Development Focus*, Wiley (2012).
2. **Reema Thareja**, *Python Programming: Using Problem-Solving Approach*, Oxford University Press.
3. **Mark Lutz**, *Learning Python*, O'Reilly Media.
4. **Eric Matthes**, *Python Crash Course*, No Starch Press.
5. **Al Sweigart**, *Automate the Boring Stuff with Python*, No Starch Press.

Dr. Neelima Guntupalli

LESSON- 09

PYTHON DICTIONARY

AIMS AND OBJECTIVES

The main aim of this chapter is understanding the concept of dictionary in Python Programming. The discussion related to understand what dictionary and its characteristics. After completion of this chapter, student will be able to know what dictionary, how it is different from other data types. Also able to know operations, functions, and methods in dictionary.

STRUCTURE

9.1. INTRODUCTION

Python, a programming language, is equipped with a wide variety of tools and functions. The dictionary is one example of such a feature. In the Python programming language, a dictionary is a collection of key-value pairs. Uniqueness is required for the dictionary keys. A value of any kind could be assigned to the dictionary. Python's dictionary is a data structure that makes it possible for us to develop code that is both simple and very effective. The fact that the keys of this data structure can be hashed is the reason why it is referred to as a hash table in many different languages. In a moment, we will comprehend the significance of this.

Using Python dictionaries, we can easily obtain a value that has been associated with a specific key and then immediately access that value. It is recommended that we make use of them if we are looking for a certain Python object, also known as a lookup method.

9.2 PYTHON DICTIONARY

A dictionary in Python is a set of objects that let's us store information in key-value pairs. With Python dictionaries, we may rapidly obtain a value by associating it with a distinct key. Using them whenever we need to locate (search for) a certain Python object is a good concept. For this purpose, lists can also be used, but they operate far more slowly than dictionaries.

9.2.1. The Characteristic of Dictionary

- In the first place, the dictionary will have information in the form of key-value pairs.
- A colon ":" sign is used to visually define the key and the values.
- The representation of an item can consist of a single key-value pair.
- It is not permitted to have duplicate keys.
- It is possible to acknowledge duplicate values.
- It is quite OK to use heterogeneous objects for both keys and values.
- The order of the insertion is not maintained.
- a dictionary object that is capable of being altered.
- Dictionary entries behave in a dynamic manner.
- The notions of indexing and slicing are not applicable in this situation.

9.2.2. Creating Python Dictionary

In python, a dictionary is created using the key:value pairs using the curly brackets {} and is separated by commas. The syntax for creating dictionary is shown below:

Syntax:

```
my_dict = {
            "key1":"value1",
            "key2":"value2"
          }
```

In the above syntax my_dict is a dictionary created with two pair of items differentiated with different keys and values.

Example:

```
# creating a dictionary

    country_capitals = {

                        "Germany": "Berlin",

                        "Canada": "Ottawa",

                        "England": "London"

                       }
```

In the above example country_capitals is a dictionary created with three pair of items which includes {"Germany": "Berlin"}, {"Canada": "Ottawa"} and { “England": "London"}.

9.3. ACCESSING DICTIONARY ELEMENTS

To access an element from the dictionary there are three ways and are described below:

- Access by Key
- Access by get () function
- Access of nested dictionary

9.3.1 Access Dictionary by Key

We can access the value of a dictionary item by placing the key inside square brackets. It accesses and prints the values associated with the keys. The keys and values showcasing can be of different data types (string and integer).

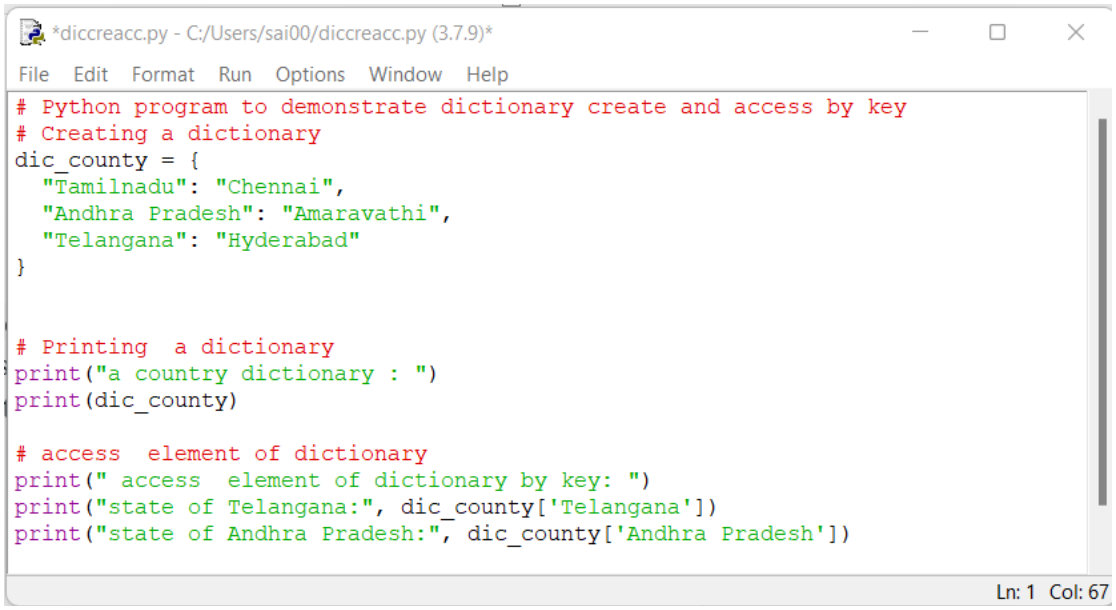
Syntax:

```
Value= dictionary_name[‘Key’]
```

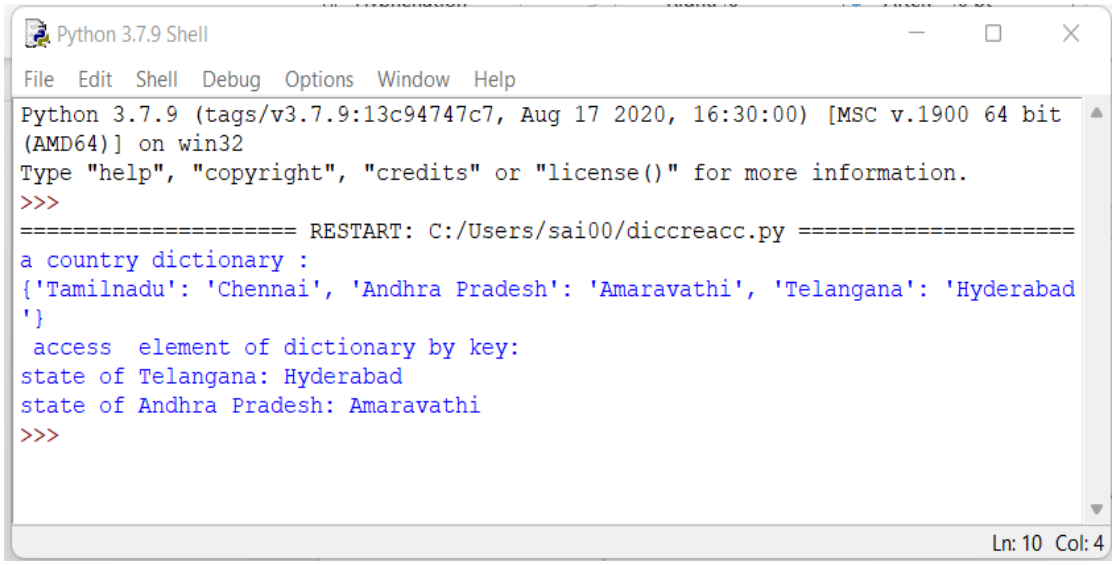
Example:

State= dic_county [‘Andhra Pradesh’]

Example:



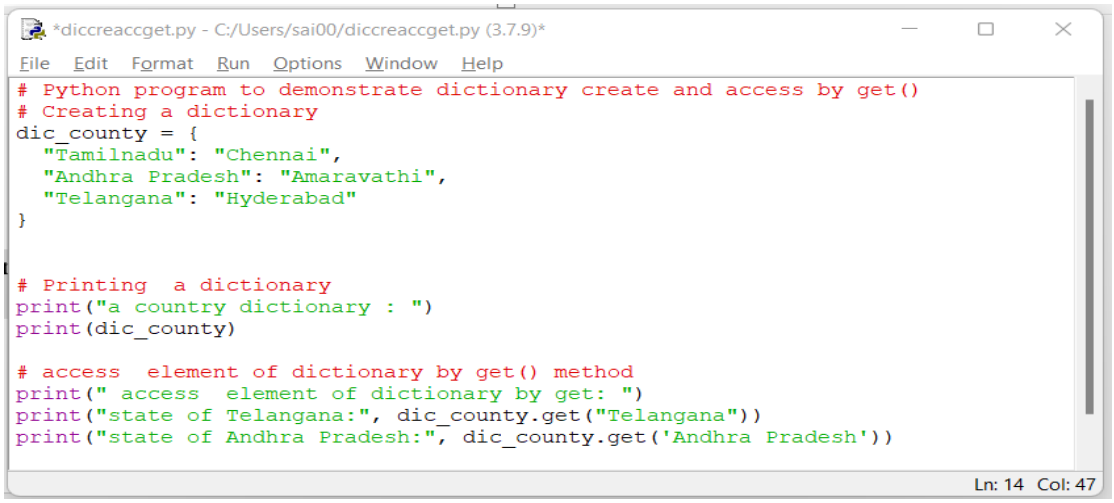
Output:



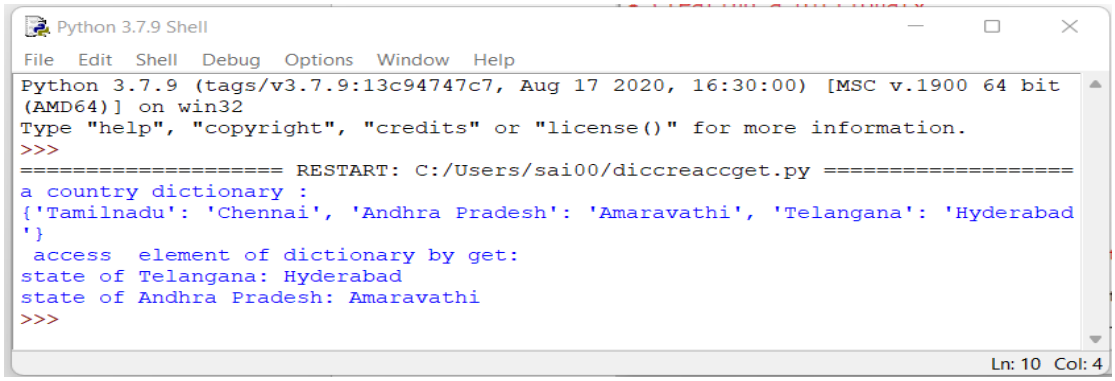
9.3.2 Access dictionary by get() method

The code demonstrates accessing a dictionary element using the get() method. It retrieves and prints the value associated with the key 3 in the dictionary ‘Dict’. This method provides a safe way to access dictionary values, avoiding KeyError if the key doesn’t exist.

Example:



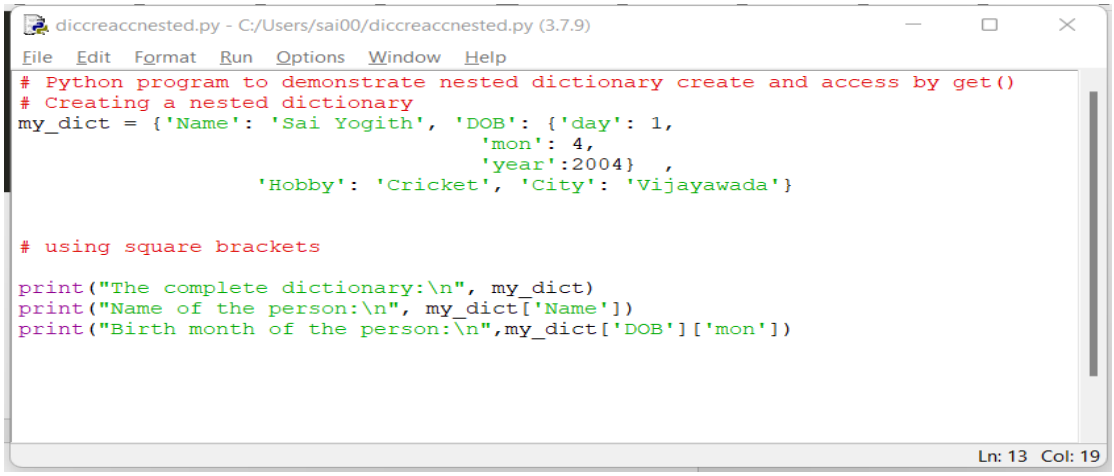
Output:



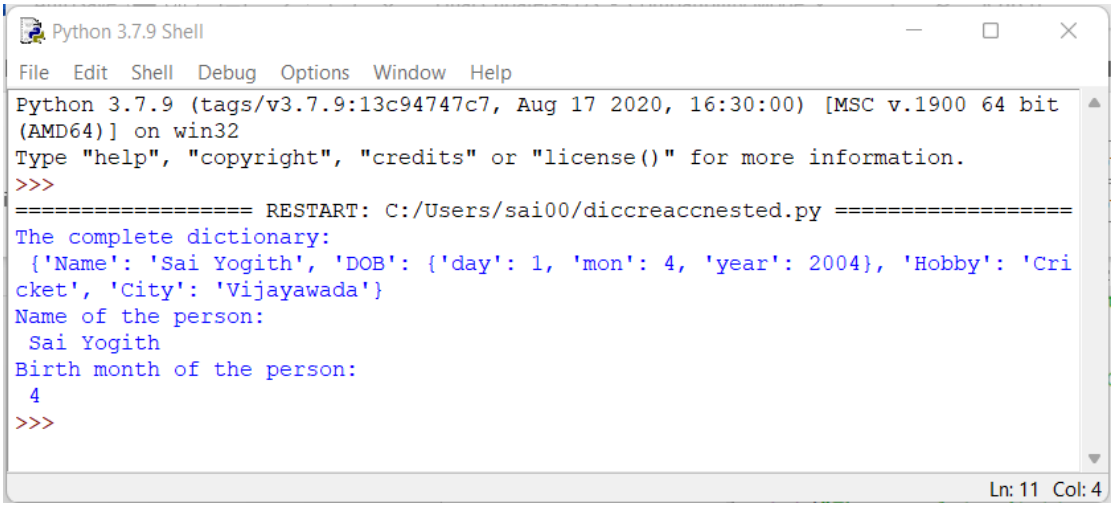
9.3.3 Access of Nested Dictionary

To access the value of any key in the nested dictionary, use indexing [] syntax. It first accesses main dictionary associated with the key and then, it accesses a specific value by navigating through the nested dictionaries.

Example:



Output:



9.4 DICTIONARY METHODS

Dictionary methods are used to perform specific functionality over dictionary that may be updating, adding, extracting a, removing and etc operations on keys and items. Some of the functions includes in given Table 9.1.

Table 9.1. Dictionary Methods

Function	Description
pop()	Removes the item with the specified key.
update()	Adds or changes dictionary items.
clear()	Remove all the items from the dictionary.
keys()	Returns all the dictionary's keys.
values()	Returns all the dictionary's values.
get()	Returns the value of the specified key.
popitem()	Returns the last inserted key and value as a tuple.
copy()	Returns a copy of the dictionary.

9.4.1. Update Elements Methods

Dictionaries are subject to change. Using an assignment operator, we can add new things or change the value of existing items.

Example:

diccreaccchange.py - C:/Users/sai00/diccreaccchange.py (3.7.9)

File Edit Format Run Options Window Help

```
# Python program to demonstrate update dictionary
# Creating a nested dictionary
my_dict = {'Name': 'Sai Yogith', 'DOB': {'day': 1,
                                         'mon': 4,
                                         'year':2004}  ,
                   'Hobby': 'Cricket', 'City': 'Vijayawada'}

# Details before updation
print(" Details before updation \n")
print("The complete dictionary:\n", my_dict)
print("Name of the person:\n", my_dict['Name'])
print("City of the person:\n", my_dict['City'])

#Updating a dictionary

my_dict['Name']='Lakkakul Sai Yogith'
my_dict['City']='Bangalore'

# Details after updation
print("\n Details after updation \n")
print("The complete dictionary:\n", my_dict)
print("Name of the person:\n", my_dict['Name'])
print("City of the person:\n", my_dict['City'])
```

Ln: 13 Col: 0

Output:

Python 3.7.9 Shell

File Edit Shell Debug Options Window Help

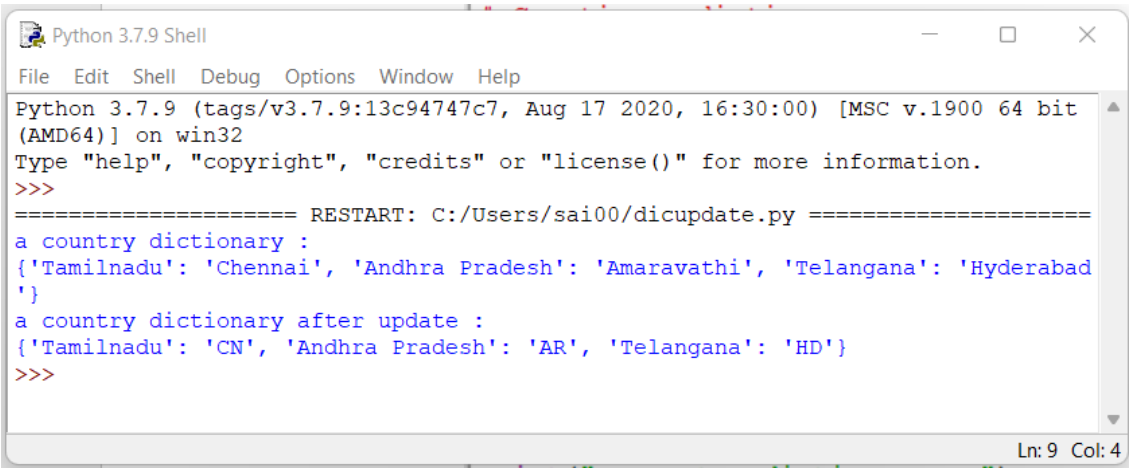
```
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================== RESTART: C:/Users/sai00/diccreaccchange.py ==================
 Details before updation

The complete dictionary:
 {'Name': 'Sai Yogith', 'DOB': {'day': 1, 'mon': 4, 'year': 2004}, 'Hobby': 'Cri
cket', 'City': 'Vijayawada'}
Name of the person:
 Sai Yogith
City of the person:
 Vijayawada

 Details after updation

The complete dictionary:
 {'Name': 'Lakkakul Sai Yogith', 'DOB': {'day': 1, 'mon': 4, 'year': 2004}, 'Hob
by': 'Cricket', 'City': 'Bangalore'}
Name of the person:
 Lakkakul Sai Yogith
City of the person:
 Bangalore
>>>
```

Ln: 19 Col: 20

By declaring value together with the key, for example, Dict[Key] = 'Value', one value at a time can be added to a Dictionary. Another approach is to use Python's update () function. Python's update () method is a built-in dictionary function that updates the key-value pairs of a dictionary using elements from another dictionary or an iterable of key-value pairs. With this method, you can include new data or merge it with existing dictionary entries.

Example:

Output:

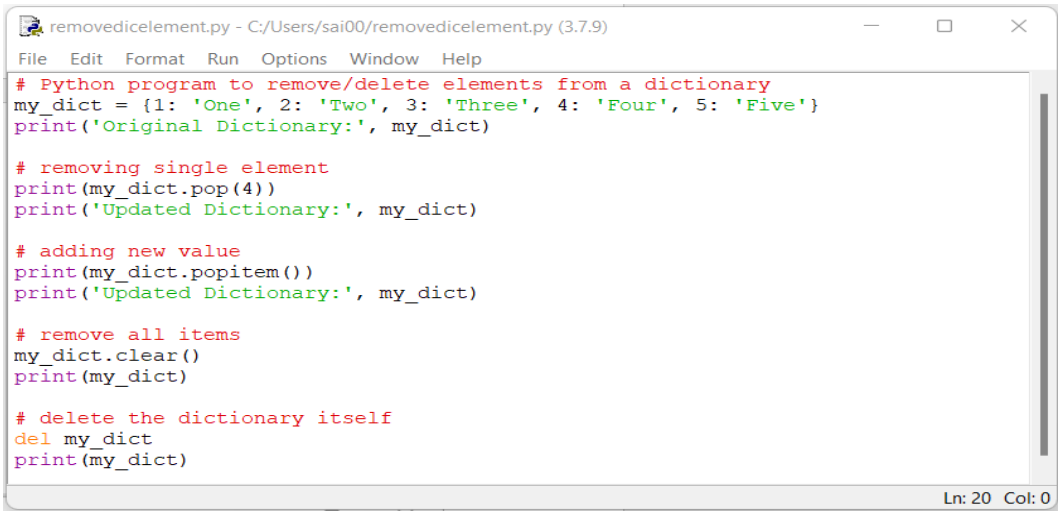


9.4.2. Removing Elements Methods

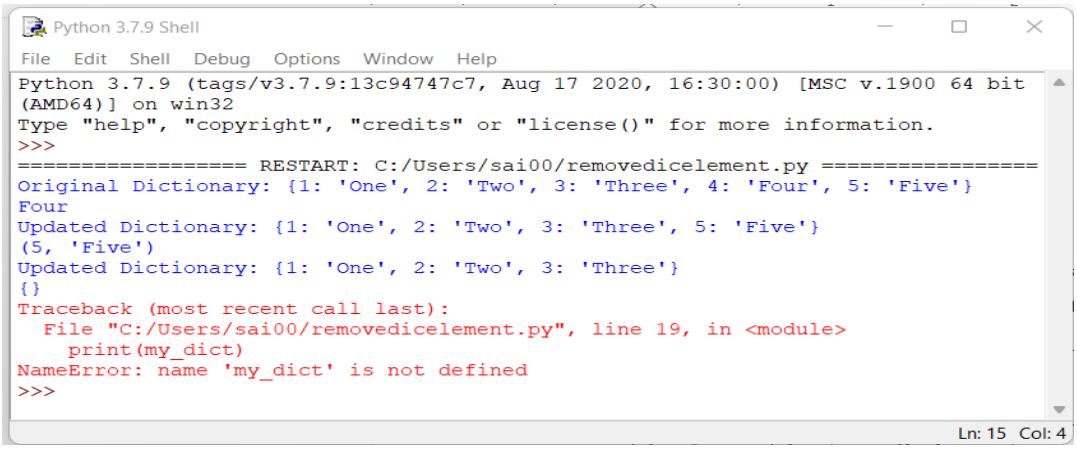
A key can be removed from a dictionary in three ways: from an individual entry, from all entries, or from the entire dictionary.

1. The pop () function can be used to remove a single element. The value of the key that has been specified to be eliminated is returned by the pop () function.
2. To randomly remove any elements (key-value pairs) of the dictionary, we can use the popitem() It returns the arbitrary key-value pair that has been removed from the dictionary.
3. Using the clear () method, all elements can be eliminated at once. The del keyword is used to completely delete the entire dictionary.

Example:



Output:



9.4.3 keys () and values() Methods

In Python, the keys () function returns a view object that contains dictionary keys, which enables quick access and iteration across the dictionary.The values() method in Python returns a view object that contains all of the dictionary values. This view object can be accessed and iterated through in an effective manner within Python.

Syntax:

d = {'key': 'value'}

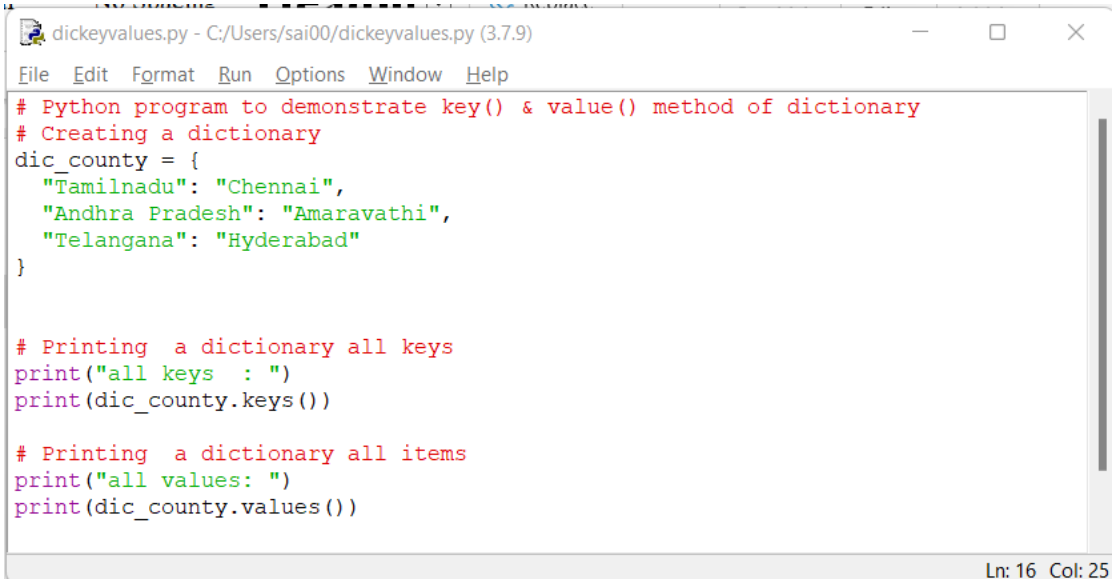
d.keys()

Syntax:

d = {'key': 'value'}

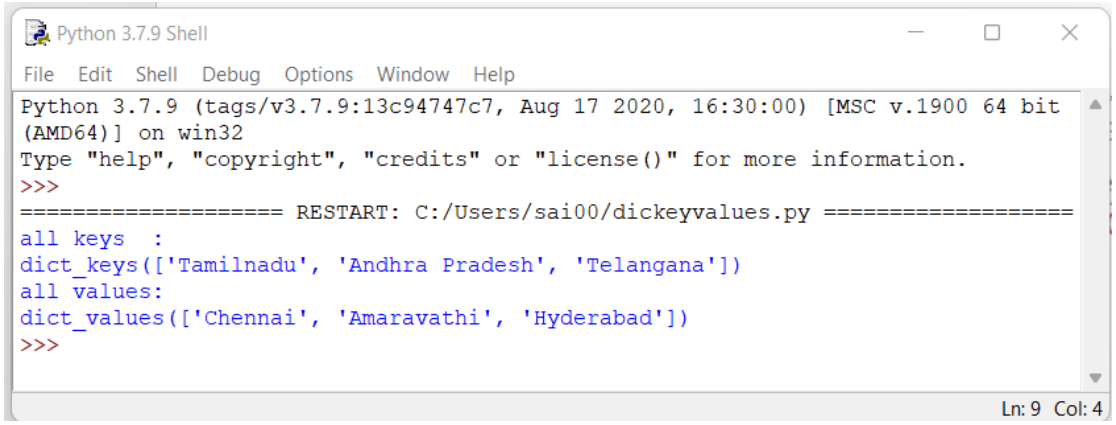
d. values ()

Example:



In the above example , created dictionary called dic_country with three elements with the usage of keys() and values() fucntions displayed the information related every key and values associated with elements stored in dic_county dictionary. The reslut shown in output.

Output:



9.5 PYTHON DICTIONARY FUNCTIONS

The Python dictionary offers a wide range of methods that may be utilized to conduct operations on key-value pairs in an easy and convenient manner. The following is a list of functions using the Python dictionary shown in Table 9.1.

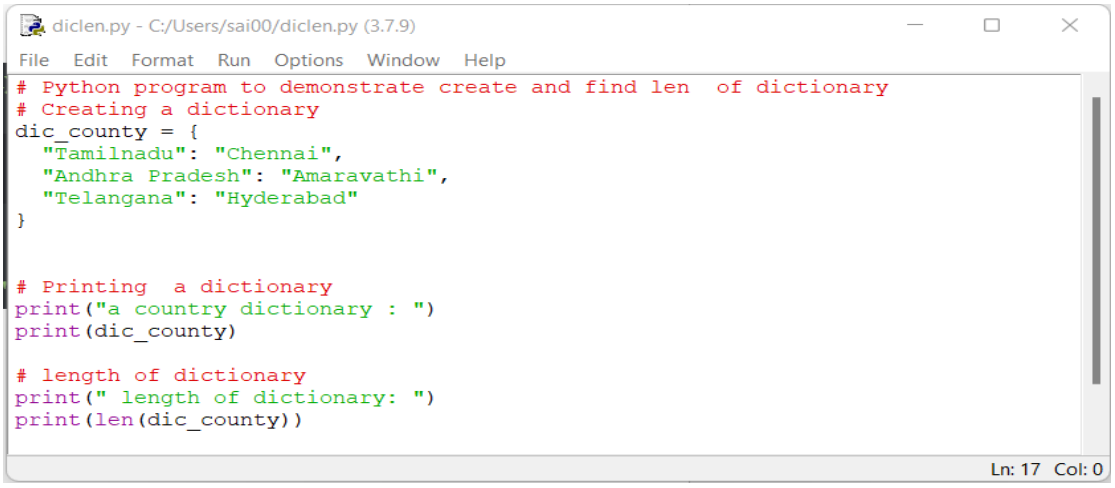
Table 9.1. Python Dictionary Functions

Function	Python Expression	Description
len()	len(my_dictionary)	Returns the length of the dictionary (key count).
sorted ()	sorted (dictionary_name)	Returns the dictionary with keys sorted in ascending order.
all ()	all (dictionary_name)	Returns True if all the keys in the dictionary are True (not 0 and False).
any ()	any(dictionary_name)	Returns True is any of the keys in the dictionary is True.
str ()	str (dictionary_name)	Returns a string representation of the dictionary passed as the argument.

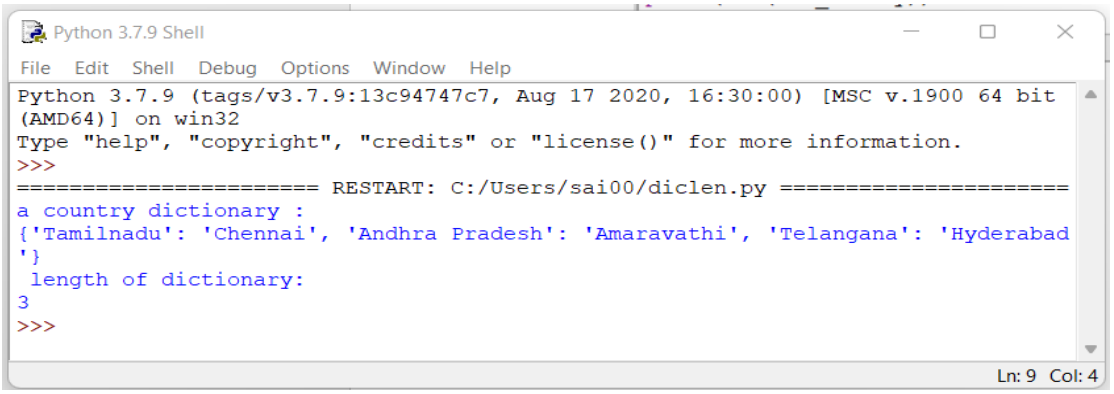
9.5.1 len() function

Using the len() method, which returns the item count, one can determine the length of a dictionary by its use. Printing the length of my dictionary is as follows:

Example:



Output:

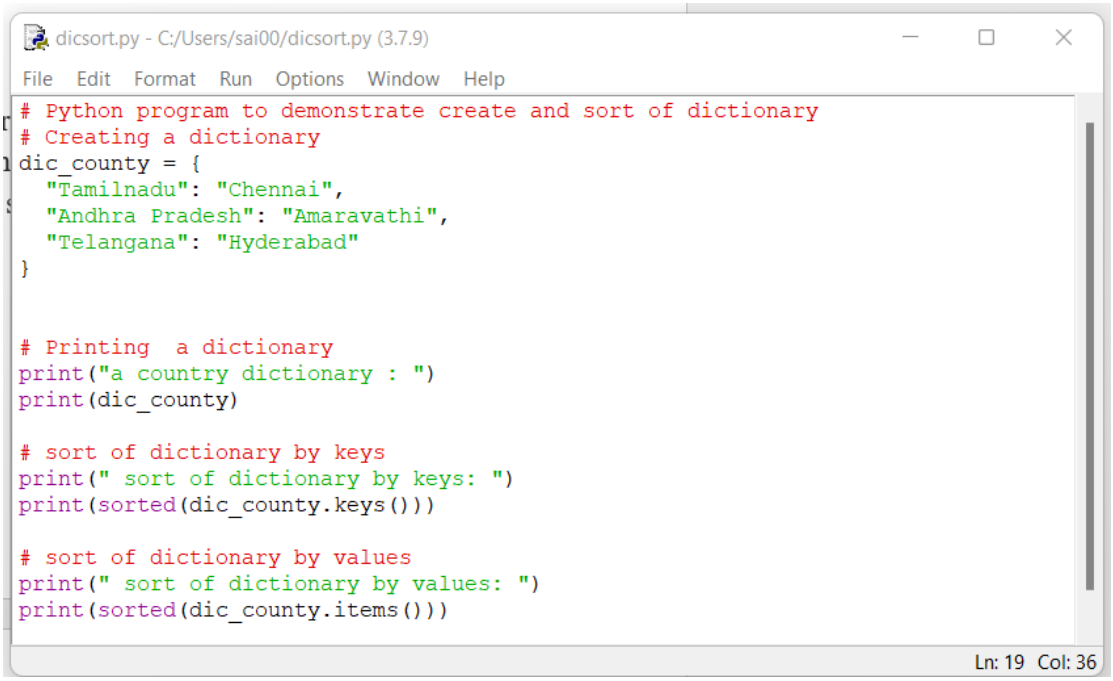


In the above example , dic_country elements count is determined by calling the len() function and displayed lengh of the dictionary 3 it means dictinay holds the three elemtns and reslut shown in output.

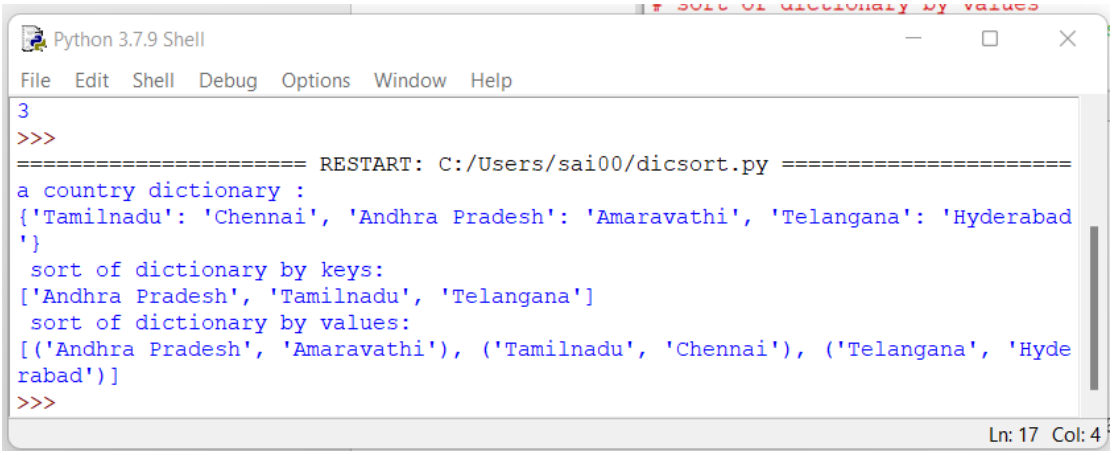
9.5.2 sorted () function.

Sorting the dictionary can be accomplished with Python's built-in keys functions, which include the keys () and values () functions. Any iterable can be used as an argument, and it will return the sorted list of keys you provided. The dictionary can be arranged in ascending order by using the keys to sort the entries. First, let's get familiar with the below example.

Example:



Output:



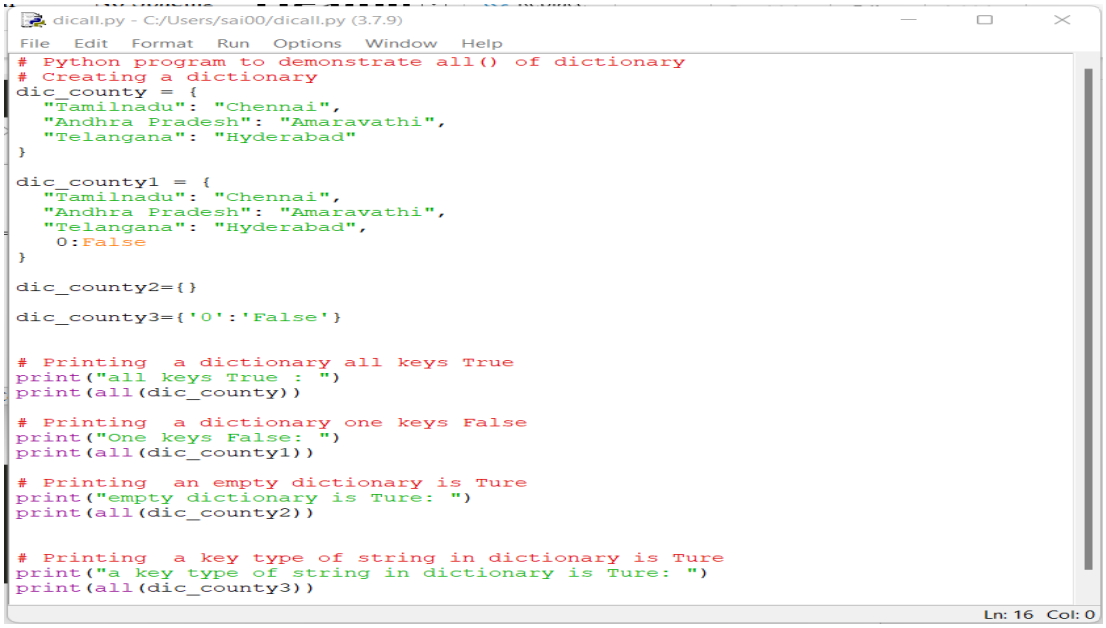
We have declared a dictionary of names in the code that was just presented. We made use of the built-in function in conjunction with the sorted() method, which provided us with a list of

the keys that had been sorted. We then proceeded to utilize the items() function in order to obtain the dictionary in the order that it was sorted.

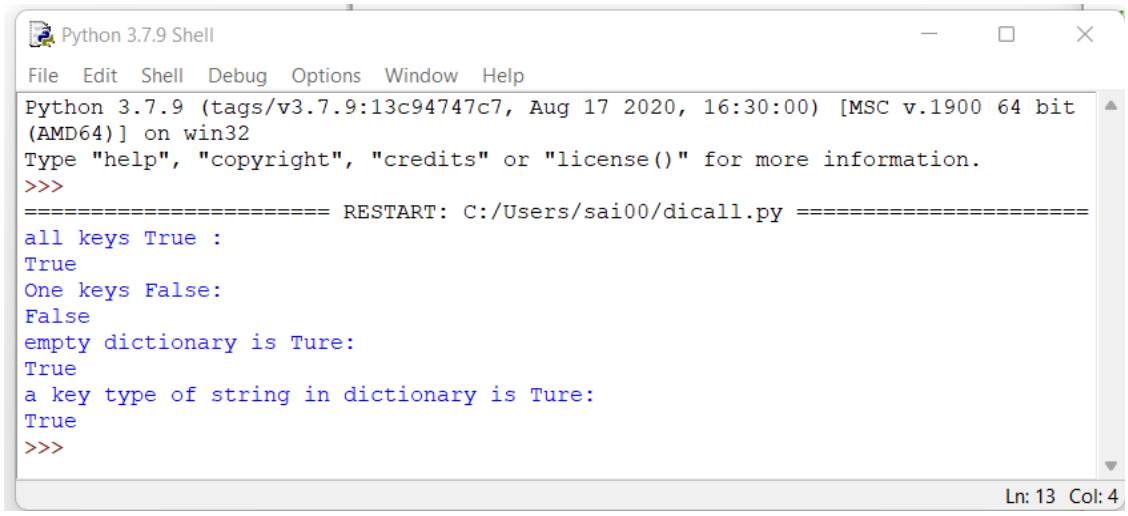
9.5.3 all() function

A dictionary's keys are the only elements that are examined when the all() method is applied to it; the values are not examined. In the event that not all of the keys in the dictionary are true, the all() method will return FALSE; but, if all of the keys are true, it will return false. In the event that the dictionary does not consist of any entries, the all() function also returns a value of TRUE.

Example:



Output:

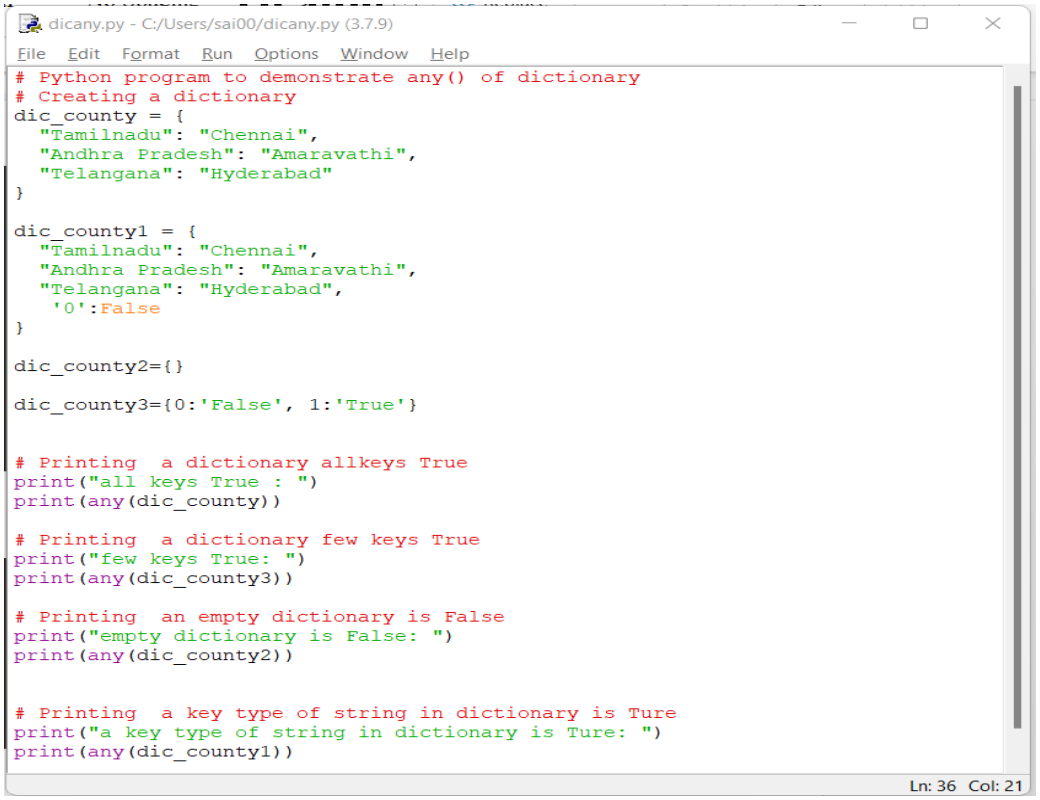


9.5.4 any () function

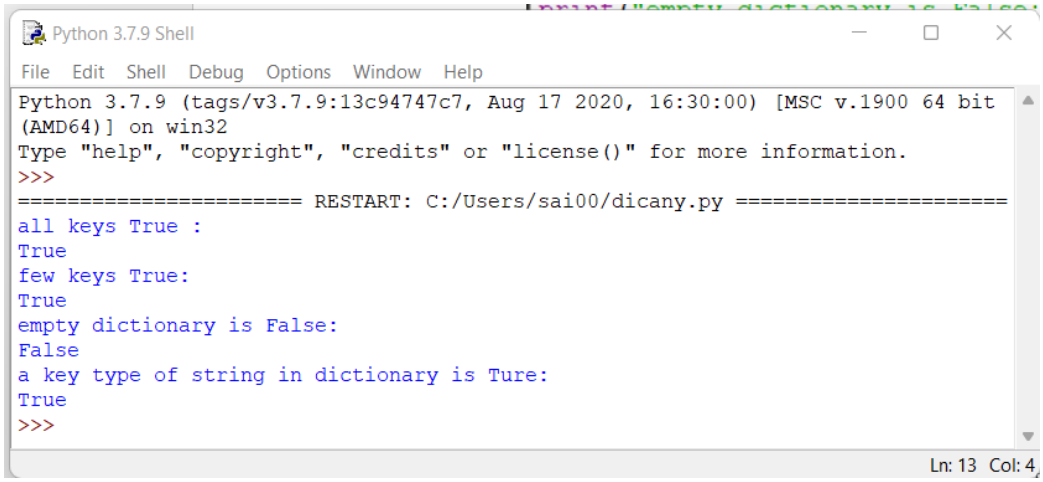
The any () method only verifies the keys of a dictionary when it is applied to a dictionary; it does not verify the values. If any of the keys associated with the dictionary are true, the any () method

will return TRUE; otherwise, it will return FALSE. In the event that the dictionary does not consist of any entries, the any () function also returns FALSE.

Example:



Output:



9.6 A DICTIONARY AS A SUBSTITUTE FOR MULTIWAY CONDITION

When we first introduced dictionaries, we emphasized their ability to store and retrieve data using user-defined keys.

However, dictionaries can also replace long multiway if–elif–else chains where multiple

conditions are used to perform simple lookups or mappings. Let us consider a motivating example.

Example Problem

Suppose we want to write a function complete() that accepts the two-letter abbreviation of a weekday (e.g., 'Tu') and returns the full name of the day (e.g., 'Tuesday').

Desired behavior:

```
>>> complete('Tu')
'Tuesday'
```

Traditional Implementation: Using Multiway if–elif–else

A straightforward (but lengthy) way to implement this is to use a seven-way conditional chain:

```
def complete(abbreviation):
    'Returns the day of the week corresponding to abbreviation'
    if abbreviation == 'Mo':
        return 'Monday'
    elif abbreviation == 'Tu':
        return 'Tuesday'
    elif abbreviation == 'We':
        return 'Wednesday'
    elif abbreviation == 'Th':
        return 'Thursday'
    elif abbreviation == 'Fr':
        return 'Friday'
    elif abbreviation == 'Sa':
        return 'Saturday'
    else:  # abbreviation must be 'Su'
        return 'Sunday'
```

Although correct, this implementation is inefficient, verbose, and difficult to maintain. Each new mapping (for example, adding 'Ho' → 'Holiday') would require adding another elif condition.

Improved Implementation: Using a Dictionary

A more elegant and efficient solution is to represent the mapping between abbreviations and full day names using a dictionary.

```
def complete(abbreviation):
    'Returns the day of the week corresponding to abbreviation'
    days = {
        'Mo': 'Monday',
        'Tu': 'Tuesday',
        'We': 'Wednesday',
        'Th': 'Thursday',
        'Fr': 'Friday',
        'Sa': 'Saturday',
        'Su': 'Sunday'
    }
    return days[abbreviation]
```

Explanation

- The dictionary days maps each two-letter abbreviation (the key) to the corresponding full day name (the value).
- To find the full name, we simply index the dictionary using the abbreviation: days[abbreviation].
- The dictionary performs this lookup instantly, without testing multiple conditions.

Sample Run

```
>>> complete('We')
'Wednesday'
>>> complete('Su')
'Sunday'
```

Benefits of the Dictionary Approach

Aspect	Multiway if–elif–else	Dictionary Mapping
Code length	Long and repetitive	Compact and readable
Efficiency	Each condition is checked sequentially	Direct key lookup (constant time)
Maintainability	Hard to update or extend	Easy to modify or add key–value pairs
Concept	Conditional branching	Key–value mapping

Using dictionaries to replace multiway conditionals demonstrates Python's expressive power and data-oriented programming style.Whenever conditions correspond to a clear mapping between keys and values, a dictionary provides a cleaner, faster, and more scalable solution.

9.7 DICTIONARY AS A COLLECTION OF COUNTERS .

One of the most important and practical applications of dictionaries is counting occurrences — also known as frequency counting.

Many programs, from search engines to data analytics tools, rely on counting how many times specific items occur within a dataset.

Example Problem

Suppose we want to count the number of occurrences of each name in a list of student names.

```
students = ['Cindy', 'John', 'Cindy', 'Adam', 'Adam',
        'Jimmy', 'Joan', 'Cindy', 'Joan']
```

We need a function frequency() that will take such a list and compute how many times each name appears.

Concept

For each distinct item in the list, we want to:

1. Create a counter initialized to zero.
2. Increment the counter each time the item occurs.

The challenge is that we don't know in advance how many distinct items exist.

Solution: Use a Dictionary of Counters

A dictionary is ideal because it can dynamically:

- Create a new key (item) the first time it appears.
- Associate it with a counter value.
- Increment that value each subsequent time the item appears.

Implementation

```
def frequency(items):
   'Counts occurrences of each distinct element in the list items'
   counters = {}
   for item in items:
      if item in counters:
         counters[item] += 1
      else:
         counters[item] = 1
   return counters
```

Example Use

```
students = ['Cindy', 'John', 'Cindy', 'Adam', 'Adam',
        'Jimmy', 'Joan', 'Cindy', 'Joan']

print(frequency(students))
```

Output

{'Cindy': 3, 'John': 1, 'Adam': 2, 'Jimmy': 1, 'Joan': 2}

Explanation

1. The empty dictionary counters = {} starts with no keys.
2. The loop visits each item in the list:
   - If the item already exists in the dictionary, increment its counter.
   - If not, create a new key with initial value 1.
3. After processing all items, the dictionary contains each unique element with its count.

9.8 SUMMARY

Python is an excellent programming language that comes with a wide variety of feature sets. The fact that it provides a structured code makes it much simpler to comprehend. Since Python is currently one of the most widely used programming languages in the modern day, it is essential to have a comprehensive understanding of this programming language. This chapter will provide you with practical experience on how to work with dictionary along methods and functions.

9.9 TECHNICAL TERMS

Dictionary, Update, any, key, value, Get Method, Pop, Clear, and pop Items.

9.10 SELF ASSESSMENT QUESTIONS

Essay questions:

1. How is a dictionary created and called? Explain.
2. What are the various dictionary methods? Explain.
3. Explain about dictionary functions with example.

Short Notes:

1. Write about get () access method.
2. How dictionary is different form the List.

9.11 SUGGESTED READINGS

1. Steven cooper – Data Science from Scratch, Kindle edition.
2. Reemathareja – Python Programming using problem solving approach, Oxford Publication
3. "Python Pocket Reference" by Mark Lutz
4. "Python Essential Reference" by David Beazley
5. "Python Programming: An Introduction to Computer Science" by John Zelle
6. "Introduction to Computation and Programming Using Python" by John Guttag

LESSON- 10

TUPLE

AIMS AND OBJECTIVES

The main aim of this chapter is understanding the concept of tuples in Python Programming. The discussion related to understand what tuple and its characteristics is. After completion of this chapter, student will be able to know what tuple is, how it is different from other data types. Also able to know access tuples by various methods, operations, functions, and methods in tuples.

STRUCTURE

10. 1 INTRODUCTION

Python is a popular high-level, general-purpose programming language that excels at creating graphical user interfaces and web applications. It is also a popular choice for application

development due to its dynamic type and binding features. In this chapter we'll learn about tuples, an important data structure in Python programming.

Python tuples are a data structure that is quite like a list. The primary distinction between the two is that tuples are immutable, which means they cannot be modified once generated. This makes them excellent for storing non-modifiable data, such as database records. A tuple can contain any number of objects of various types, including strings, integers, floats, lists, and so on. Let's look at how to generate and use a tuple to make our programming work easier.

10.2 PYTHON TUPLE

A sequence of any items that are separated by commas and wrapped in parenthesis is referred to as a tuple. We use tuples to represent fixed collections of elements since they are immutable objects, which means they cannot be modified. Tuples are used to carry out this function. Tuple items are placed in a specific order, cannot be altered, and permit duplicate values. When we say that tuples are ordered, we are referring to the fact that the items in the tuple have a predetermined order on which they will remain indefinitely. Tuples and Python lists share some similarities in terms of indexing, nested objects, and repetition; nevertheless, the most significant distinction between the two is that a Python tuple is immutable, whereas a Python list is mutable. Tuples are used in Python programming languages. Since tuples are indexed, the first item has an index of [0], the second item uses an index of [1], and so on.

10.2.1 Creating Python Tuples

It is possible to create a tuple by associated with all of the items (elements) in parentheses () rather than square brackets [], and by separating each element with commas. It is possible for a tuple to include any number of objects of different types, including integers, floats, lists, strings, and so on. In addition, you have the option of specifying nested tuples, which can include one or more items that are either dictionaries, lists, or tuples.

The given example shows how to create simple Tuple in python:

Example:

```
emp_tup = ()    #  Empty Tuple
int_tup = (2, 8, 1, 6, 15, 3)    # A Tuple with integers
mixed_tup = (12, "sai", 81.3)    # A Tuple with mixed data items
nested_tup = ("Python", [8,5,17,6], (2, 6, 1, 20))    # Nested Tuple
```

We produced four different sorts of tuples in the example that was just presented: empty, int-type, mixed type, and nested type. It is after the initialization of data items that the size of the empty tuple is calculated. Nevertheless, the elements that are part of the int type and the mixed type are the numbers 6 and 3. An example of a nested tuple is a special sort of tuple in which each element also contains additional elements. A string, a list, and a tuple were the three elements that were present in the nested tuple that was defined before.

10.2.2 Advantages of Tuple over List

Since they are so comparable, tuples and lists are applied in scenarios that are comparable. On the other hand, there are a few benefits that come along with utilizing a tuple rather than a list.

- In contrast to lists, the Tuples cannot be modified in any way. The addition, removal, or replacement of a tuple is not possible.
- Tuples are often utilized for heterogeneous data kinds, which are distinct from one another, whereas lists are typically utilized for homogeneous data types, which are comparable to one another.
- As a result of the immutability of tuples, iterating through them is a more efficient process than iterating through a list. As a consequence of this, there is a slight improvement in performance.
- Dictionary keys can be derived from tuples that include elements that cannot be changed. When it comes to lists, this is not possible.
- If you have data that does not change, implementing it as a tuple will ensure that it continues to be protected from being written to.
- If you wish to make changes to the information contained in a tuple, we will first need to transform it into a list.

10.3 ACCESSING TUPLE

A tuple's objects can be accessed in 3 different types of ways which includes:

- Indexing
- Negative Indexing
- Slicing

10.3.1 Indexing

Accessing an item within a tuple that has an index that begins at 0 can be accomplished using the index operator []. A tuple that contains five items will have indices that range from 0 to 4, inclusive. An index that is higher than four will be considered out of range.

Example:

tupaccess.py - C:/Users/sai00/tupaccess.py (3.7.9)

File Edit Format Run Options Window Help

```
# Python program to show how to create and access a tuple with indexing
# Creating an empty tuple
emp_tup = ()
print("Empty tuple: ", emp_tup)

# A Tuple with integers
int_tup = (2, 8, 1, 6, 15, 3)
print("\nTuple with integers: ", int_tup)

# A Tuple with different data types
mixed_tup = (12, "sai", 81.3)
print("\n Tuple with different data types: ", mixed_tup)

#  A nested tuple
nested_tup = ("Python", [8,5,17,6], (2, 6, 1, 20))
print("\n A nested tuple: ", nested_tup )

print("\n 3 element of a integer tuple: ",int_tup[3])

print("\n 2 element of a mixed tuple: ",mixed_tup[2])

print("\n 1 element of  2nd element of a nested  tuple: ",nested_tup[1][2])

print("\n 2 element of  3rd element of a nested  tuple: ",nested_tup[2][3])
```

Ln: 1 Col: 71

The four types of tuples that have previously been constructed in the example above—empty, int-type, mixed type, and nested type—are accessed using an index. This operator is quite helpful in accessing particular elements from the tuple. Different elements are accessible from different types of tuples in the code above. Three elements from the integer tuple and two from the mixed tuple, for instance. Similar access is made using the indexing method in nested tuples.

Output:

Python 3.7.9 Shell

File Edit Shell Debug Options Window Help

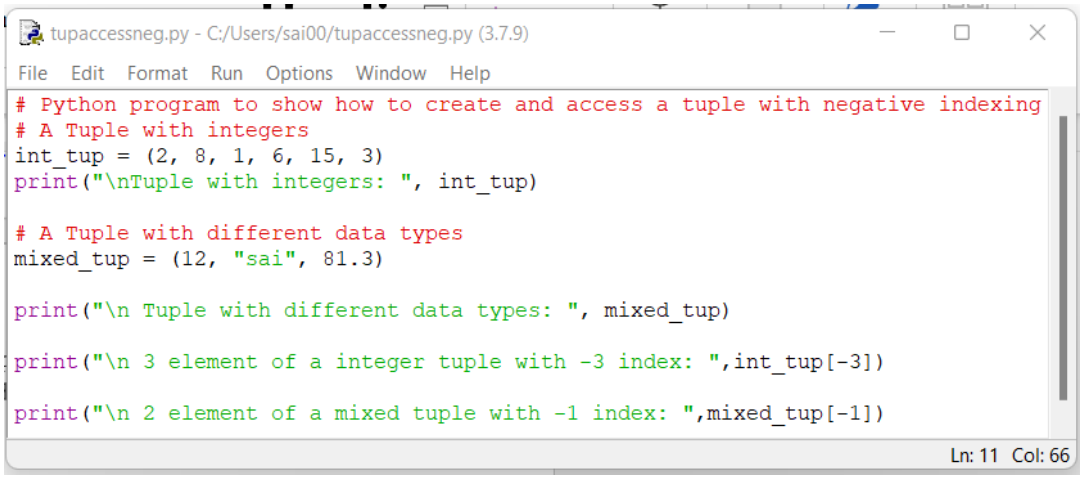
```
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/tupaccess.py ==========
Empty tuple:  ()

Tuple with integers:  (2, 8, 1, 6, 15, 3)

 Tuple with different data types:  (12, 'sai', 81.3)

 A nested tuple:  ('Python', [8, 5, 17, 6], (2, 6, 1, 20))

 3 element of a integer tuple:  6

 2 element of a mixed tuple:  81.3

 1 element of  2nd element of a nested  tuple:  17

 2 element of  3rd element of a nested  tuple:  20
>>>
```

Ln: 20 Col: 4

10.3.2 Negative Indexing

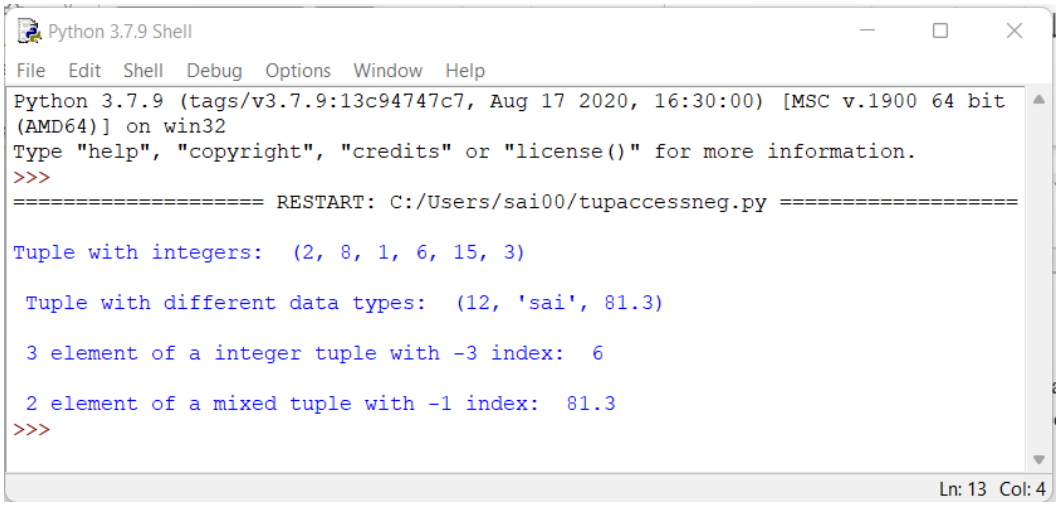
Tuple, a type of sequence object in Python, also allows negative indexing. -1 addresses the final item in the selection, -2 addresses the second-to-last item, and so on.

Example:



Two sorts of tuples, empty and int-type, are already constructed in the example above and are accessed using a negative index. For instance, the -3 and -1 indexes are used to retrieve the elements 3 and 2 of the integer and mixed tuples, respectively. In a similar vein, nested tuples can also use this type of access.

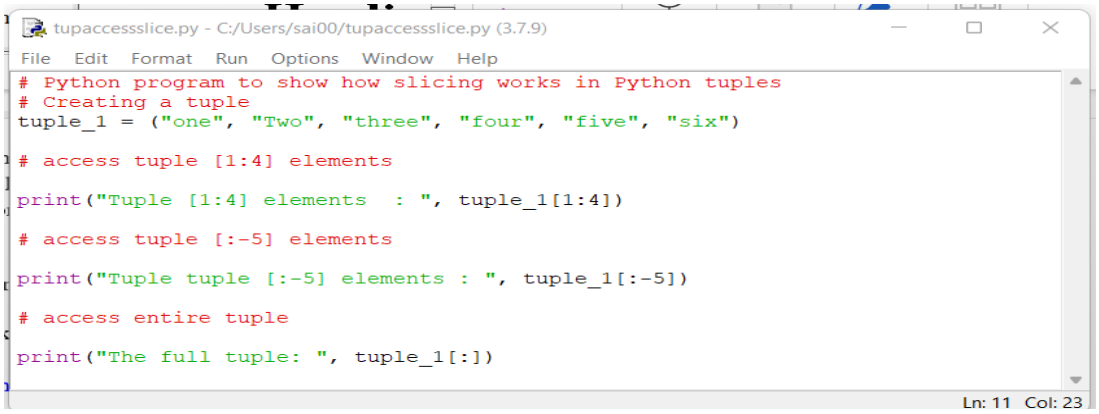
Output:



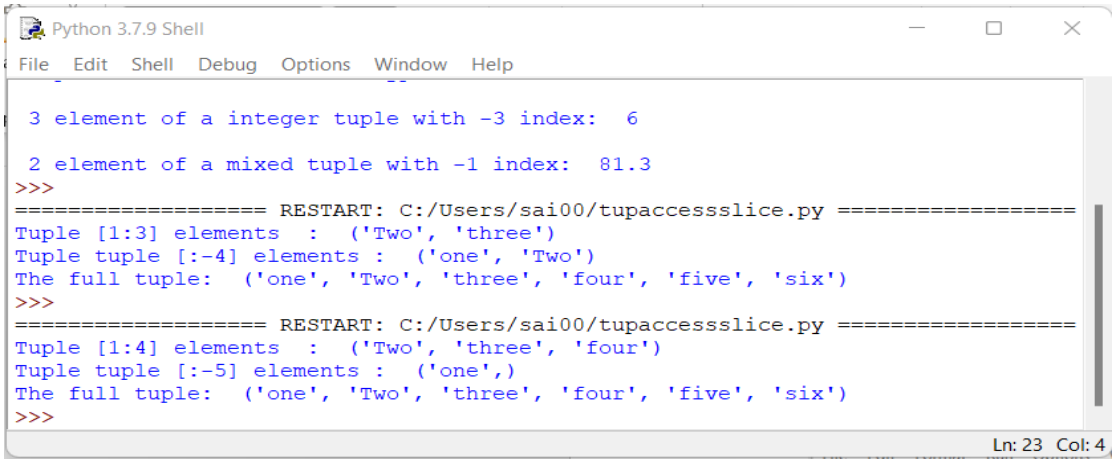
10.3.3. Slicing

In Python, tuple slicing is a widely used technique that programmers use to solve real-world problems. Examine a Python tuple. To access a range of a tuple's elements, slice it. One method is to use the colon as a simple slicing operator (:).We can use the slicing operator colon (:) to access different tuple components.

Example:



Output:



10.4 PYTHON TUPLE OPERATIONS

Tuple is a sequence in Python. As a result, we can use the + operator to concatenate two tuples and the "*" operator to concatenate many copies of a tuple. Tuple objects are used by the membership operators "in" and "not in."

Table 8.1 Python Tuple Operations

Python Expression	**Results**	**Description**
len((1, 2, 3))	**3**	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

10.4.1 Concatenation of Tuples

The process of connecting two or more tuples is called concatenation. The operator "+" is used for concatenation. Tuple concatenation is always performed starting at the end of the original tuple. On tuples, other arithmetic operations are not applicable. Concatenation can only be used to join datatypes that are the same, joining a list and a tuple result in an error. The idea of tuple concatenation is shown in Figure 8.1.

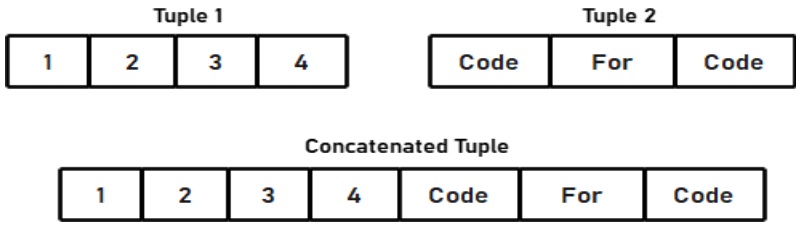


Fig 8.1 Concatenation of Tuples in Python

Example:

```
tupadd.py - C:/Users/sai00/tupadd.py (3.7.9)
File  Edit  Format  Run  Options  Window  Help
# Python program to add tuples
# Creating a tuples
tuple_1 = ("one", "Two", "three", "four", "five", "six")

tuple_2 = (4,9,3,8,5,1)

# Printing first Tuple
print("Tuple 1: ")
print(tuple_1)

# Printing first Tuple
print("\n Tuple 2: ")
print(tuple_2)
# add 2 tuples

print("\n Tuple after addition : ", tuple_1 + tuple_2)
                                                    Ln: 2  Col: 19
```

Two tuples of the types character and integer were constructed in the example above, designated as tuple_1 and tuple_2. The outcome of later addition operations applied to two tuples is reported. The result, which combines the contents of tuples 1 and 2 into a single tuple, is displayed on screen.

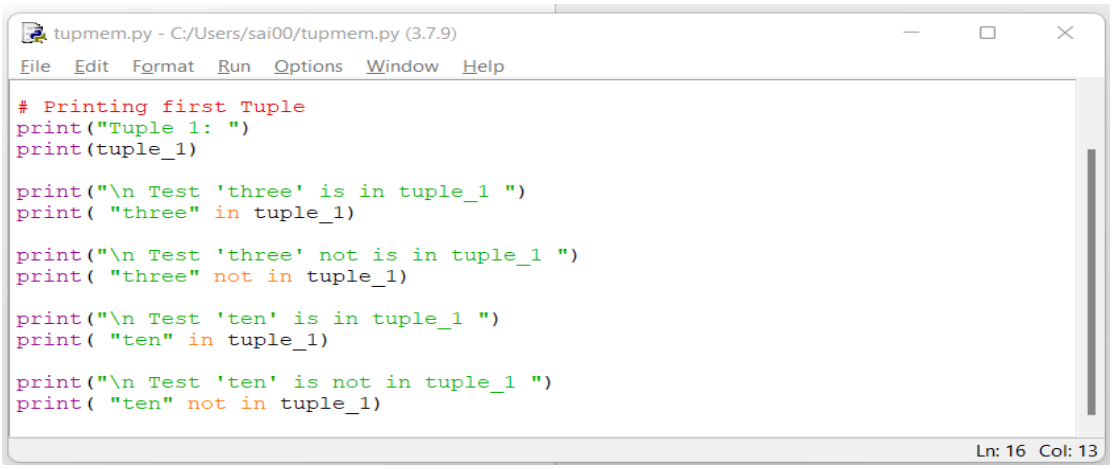
Output:

```
Python 3.7.9 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/sai00/tupadd.py ==========
Tuple 1:
('one', 'Two', 'three', 'four', 'five', 'six')

 Tuple 2:
(4, 9, 3, 8, 5, 1)

 Tuple after addition :  ('one', 'Two', 'three', 'four', 'five', 'six', 4, 9, 3,
 8, 5, 1)
>>>
                                                    Ln: 12  Col: 4
```

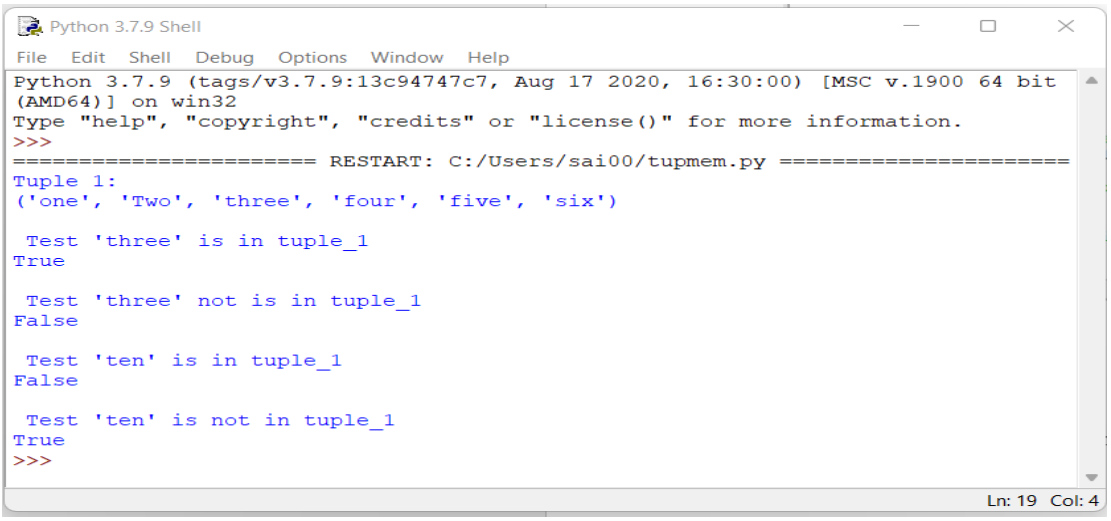
10.4.2 Tuple Membership

The existence of an item in a tuple can be ascertained by using the in and not in keywords.

Example:



Output:



Applying membership procedures on the two produced tuples, tuple_1 and tuple_2, as demonstrated in the preceding example. The results of these membership operations, such as is and is not, are TRUE or FALSE. Verified whether the term "there" is available in the case above. In a same manner, look up further words.

10.5 PYTHON TUPLE FUNCTIONS

Python offers a variety of functions for carrying out tasks. Functions such as cmp(), max(), min(), and so forth are used to carry out particular tasks. Each function's explanation can be found in Table 8.1.

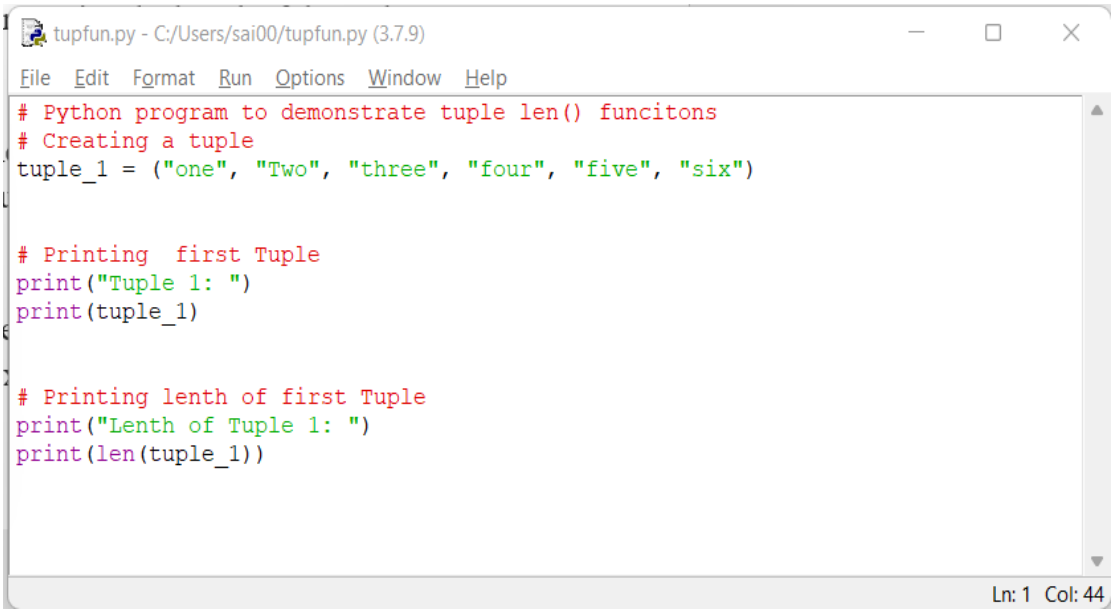
Table 8.1 Python Tuple Functions

Function	Description
cmp(tuple1, tuple2)	Compares elements of both the tuples
len(tuple)	Returns the total length of the tuple
max(tuple)	Returns the largest element from the tuple
min(tuple)	Returns the smallest element from the tuple
tuple(seq)	Converts a list into tuple

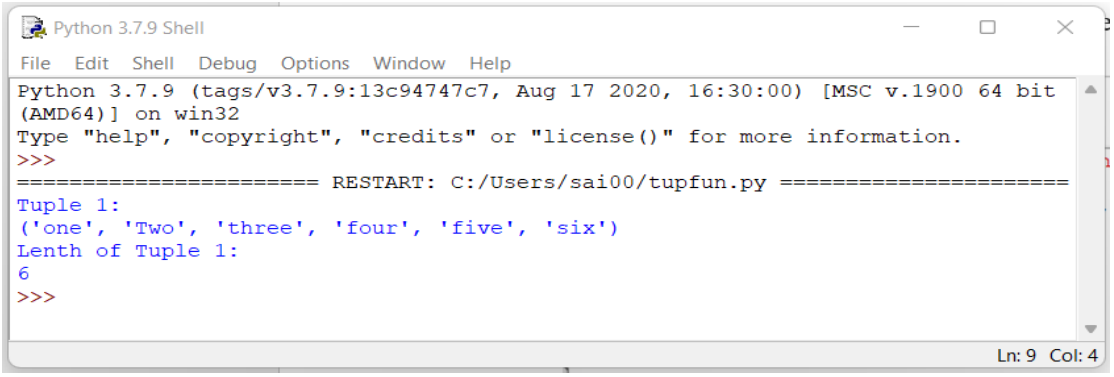
10.5.1 len()

The number of elements in a tuple can be obtained using the len() method. It accepts a tuple as an input and outputs an integer number that is the tuple's length.

Example:



Output:

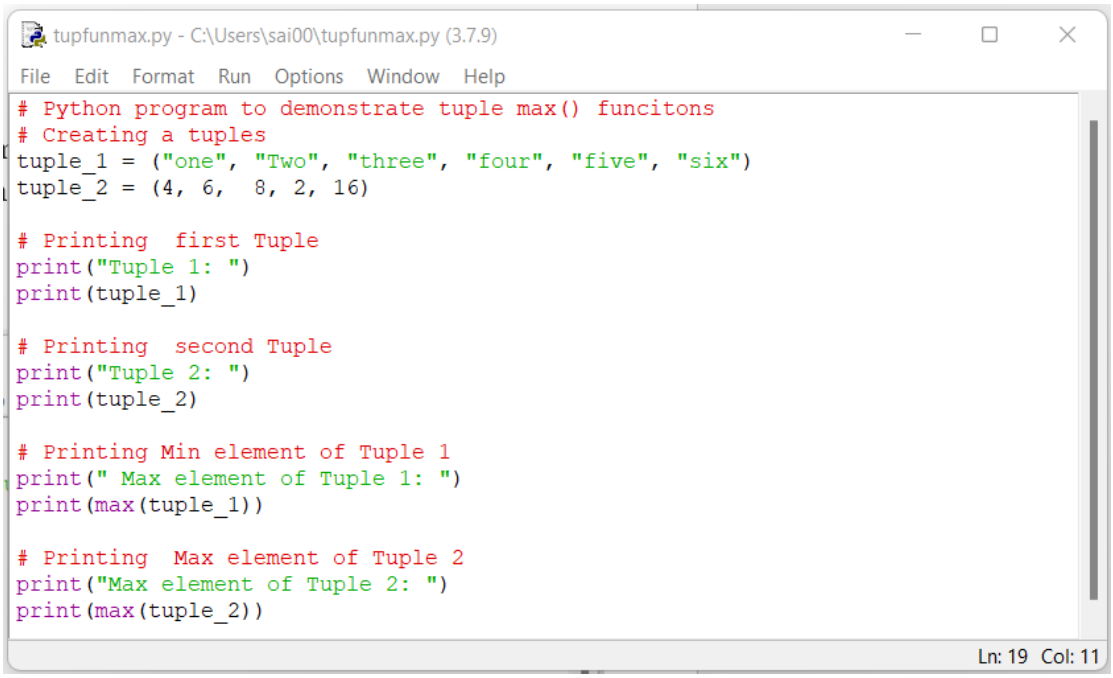


We have defined a tuple called my_tuple with five items in the example above. The length of the tuple, which is 5, was then obtained using the len() method.

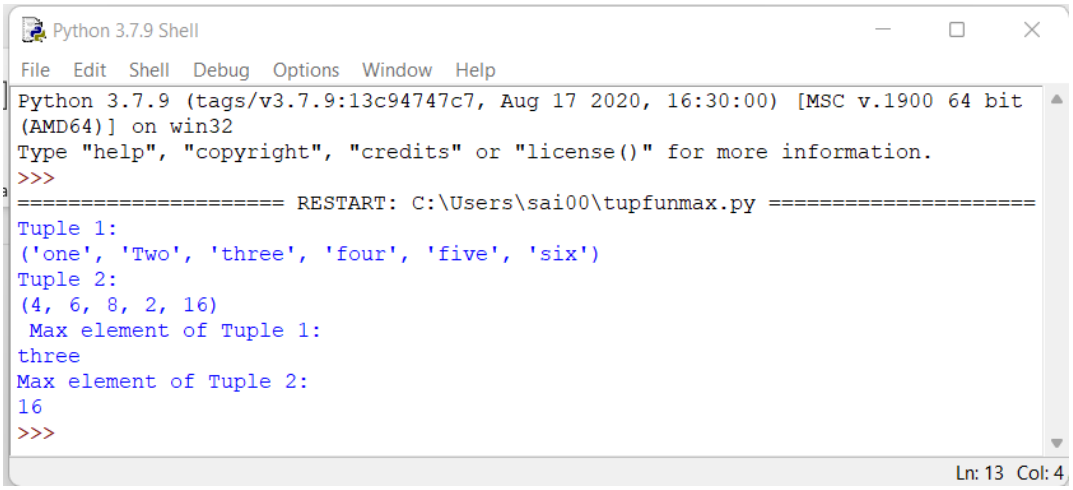
10.5.2 max ()

To get the maximum value in a tuple, use the max () function. It accepts a tuple as an input and outputs the tuple's maximum value.

Example:



Output:

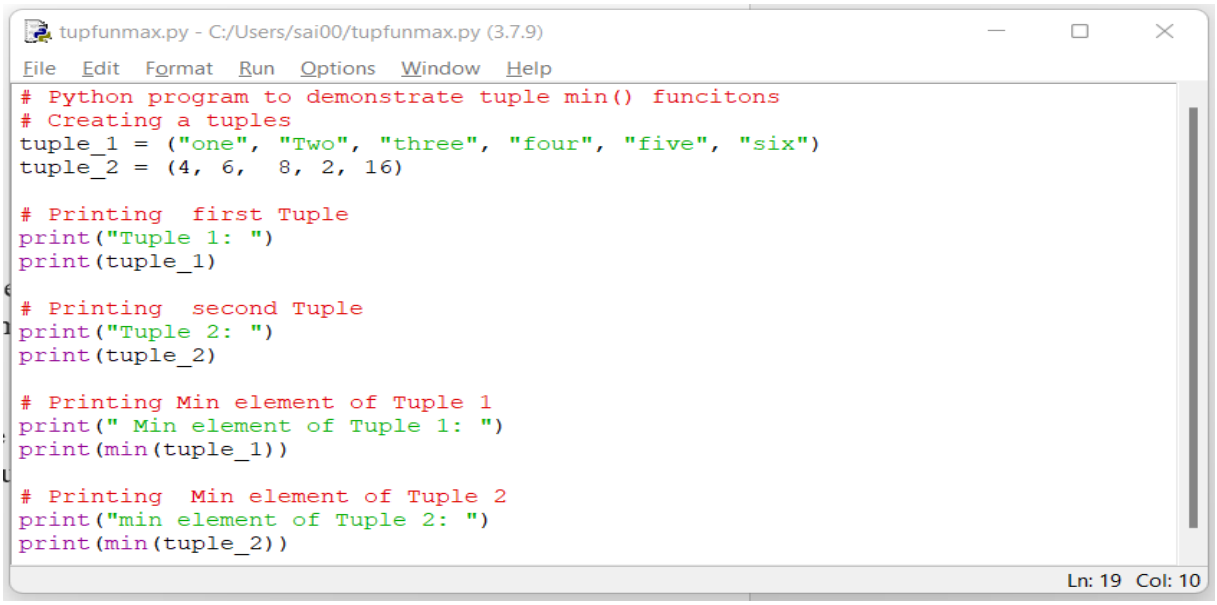


We have defined a tuple called my_tuple with five items in the example above. The maximum value in the tuple, which is 9, was then obtained using the max() method.

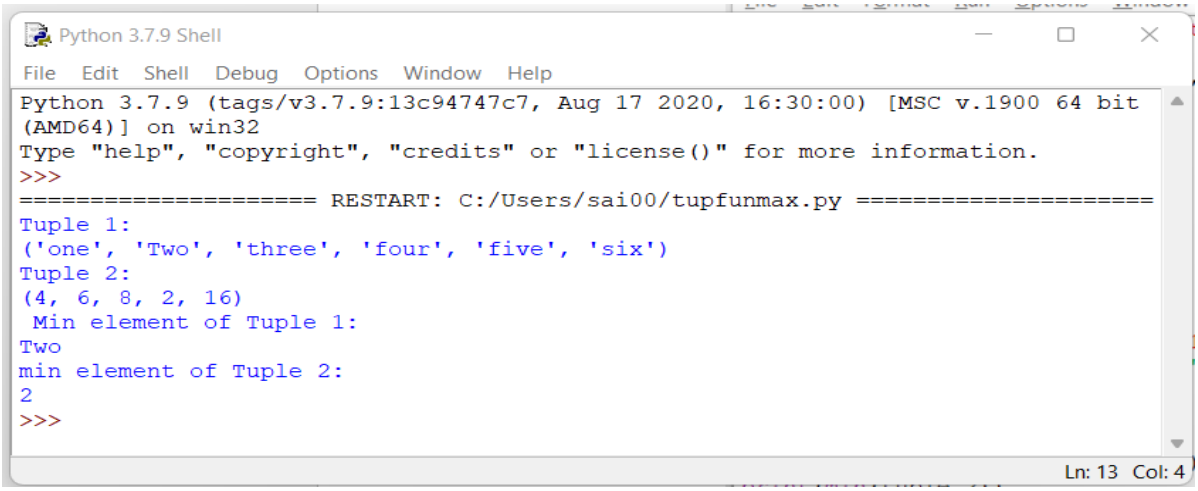
10.5.3 min ()

To get the lowest value in a tuple, use the min () function. It accepts a tuple as an input and outputs the tuple's minimal value.

Example:



Output:

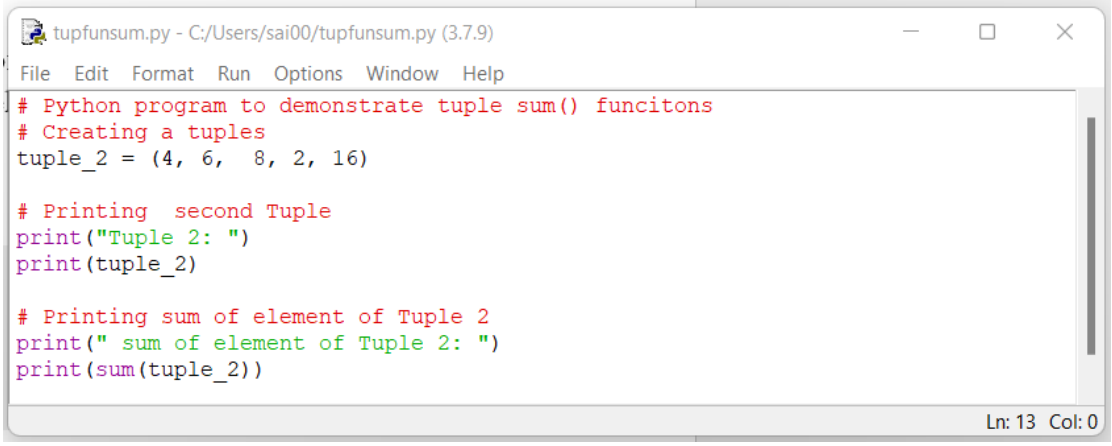


We have defined two tuples tuple_1 and tuple_2 with six and five items in the example above. Next, we obtained the tuple's minimal values Two, and 2 by using the min() function.

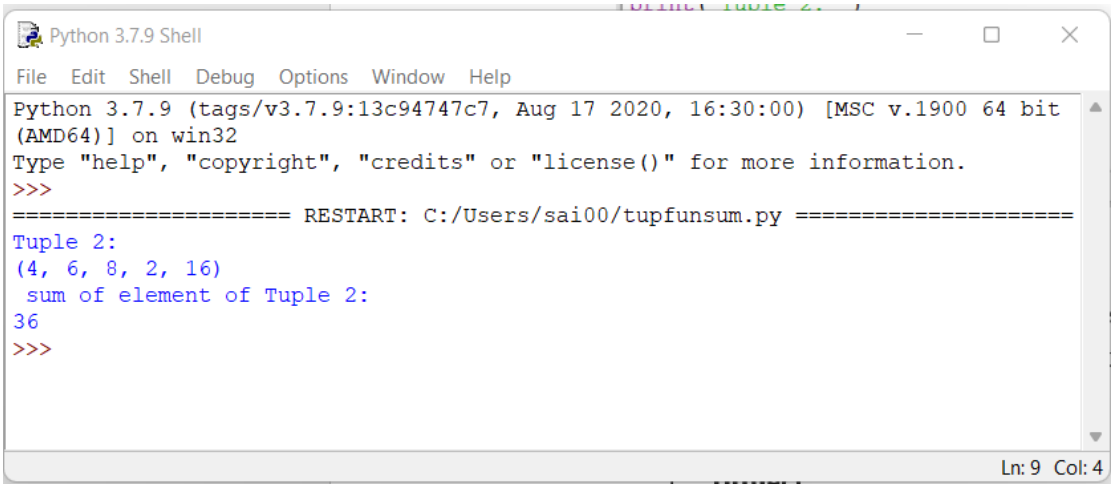
10.5.4 sum ()

The sum of each element in a tuple can be obtained using the sum () function. It accepts a tuple as an input and outputs the total of each tuple's elements.

Example:



Output:



We have defined a tuple called tuple_2 with five items in the example above. The total of all the elements in the tuple, which is 36, was then obtained using the sum () method.

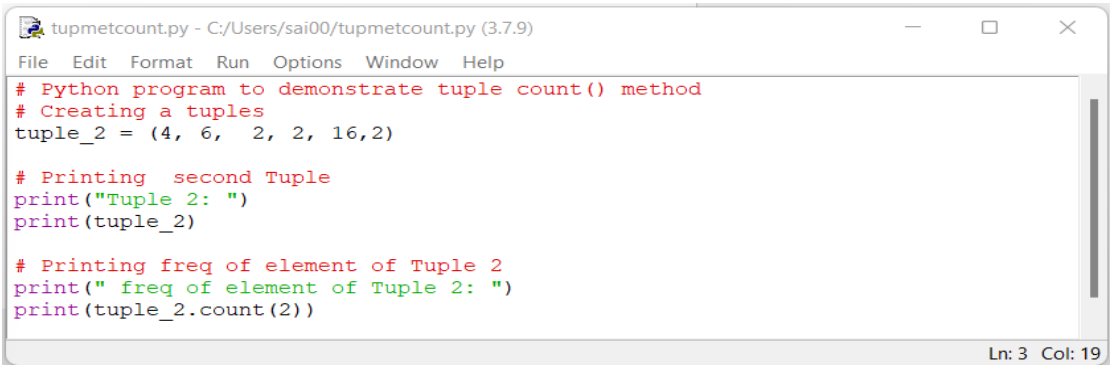
10.6 TUPLE METHODS

Python's tuple routines offer an extensive range of functionalities for working with tuples. Programmers can find the length, maximum or minimum value, total of all items, and create tuples from iterables using these functions. Easy finding and counting of particular elements within tuples is also made possible by the index() and count() operations.

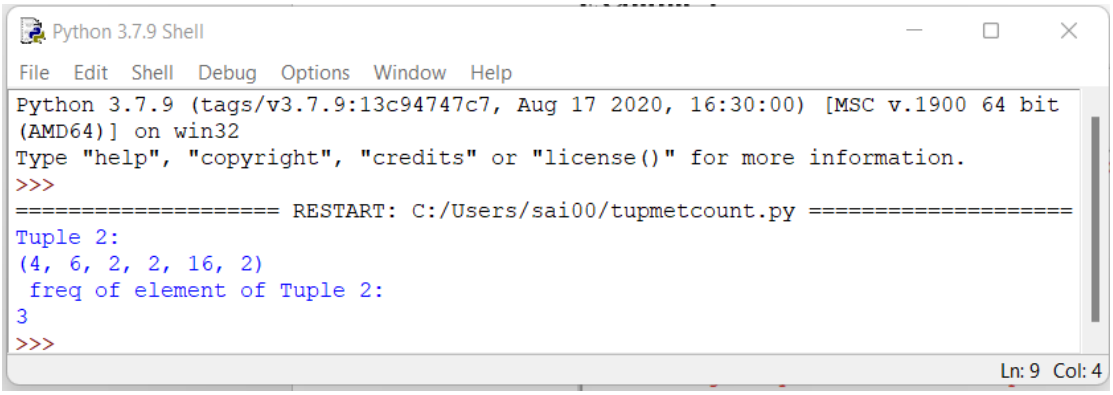
10.6.1 Count () Method

A built-in Python function called count () can be used to determine how many times a certain element appears in a tuple. The value to be counted is the only input that the method accepts.

Example:



Output:

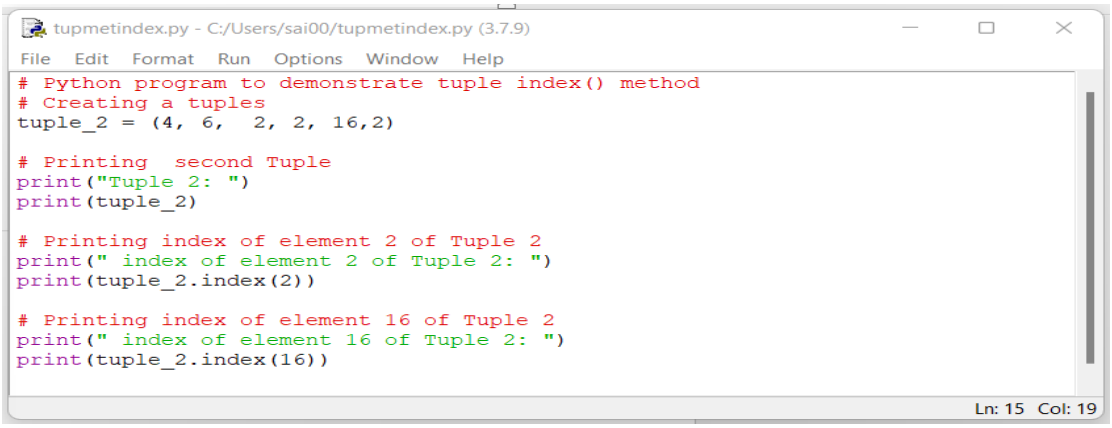


In the above example, we first create a tuple tuple_2 with some elements. Then we use the count () method to count the number of occurrences of the value 2 in the tuple. The method returns the count of 2 which is 3. Finally, we print the count.

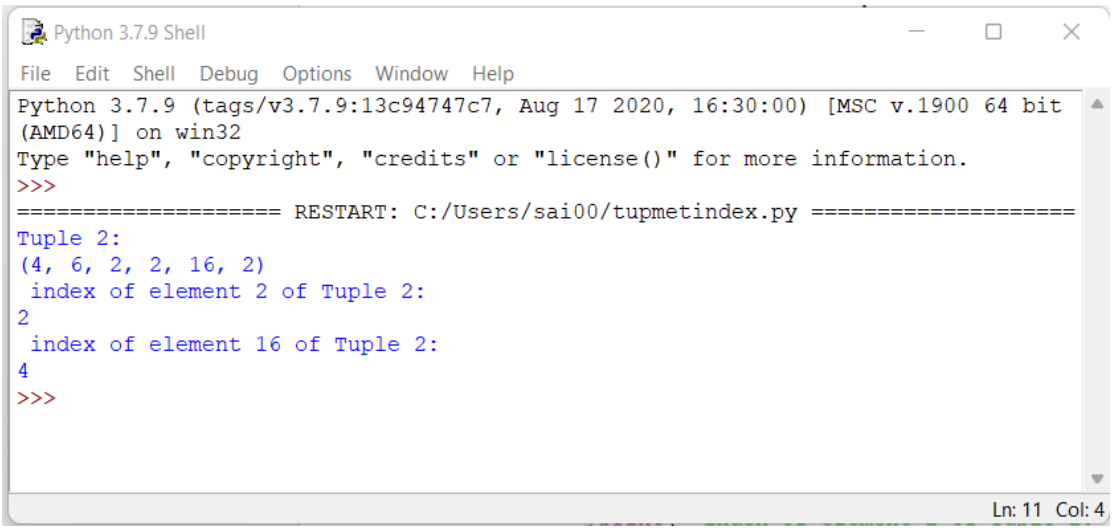
10.6.2. Index () Method

A built-in Python function called index () can be used to determine the index of a given element's first instance in a tuple. The value to be searched in the tuple is the only input required by the method.

Example:



Output:



In the preceding example, we first create a tuple, tuple_2, with certain elements. Then we use the index () method to discover the index of the tuple's first occurrence of the value 2. The method returns the index of the first occurrence of 2 (which is 1). Finally, we will print the index.

Tuples are widely used in Python for a variety of purposes, including returning multiple values from a function, representing fixed groupings of data, and serving as keys in dictionaries.

The methods discussed above make it simple to interact with tuples in Python, allowing you to extract and change their contents. The count () function in Python is useful for determining the number of repetitions of a certain element in a tuple. The index () function in Python is useful for determining the index of the first occurrence of a certain element in a tuple.

10.7 CLASS TUPLE

Tuples are implemented in Python as objects of the built-in class tuple. A tuple behaves like a list in almost every way, except that it is immutable — once created, its contents cannot be modified.

Tuples can hold heterogeneous data, can be nested, and support operations such as indexing, iteration, slicing, comparison, and packing/unpacking.

Tuples are commonly used to group related pieces of data.

Example

```
student = ('A102', 'Priya', 21, 'BCA')

print("ID:", student[0])

print("Name:", student[1])
```

```
print("Age:", student[2])

print("Course:", student[3])
```

Output

```
ID: A102

Name: Priya

Age: 21

Course: BCA
```

Here, the tuple represents a single structured record, similar to a row in a database.

- Tuples are **instances of class tuple**.
- They are **immutable**, ordered, and can hold mixed data.
- They are **faster** and **more memory efficient** than lists.
- Their immutability allows them to be **used as dictionary keys**.

10.8 TUPLE OBJECTS CAN BE DICTIONARY KEYS

A major advantage of tuples is that, unlike lists, they can be used as **keys in a dictionary**. This is because tuples are **immutable and hashable**, while lists are **mutable and unhashable**.

Let us understand this through an example.

Example – Invalid: Lists as Dictionary Keys

Suppose we want to create a phonebook where each key is a **person's name** (first and last), and the value is their phone number.

If we try to use lists as keys:

```
phonebook = {
    ['Anna', 'Karenina']: '(123)456-78-90',
    ['Yu', 'Tsun']: '(901)234-56-78',
    ['Hans', 'Castorp']: '(321)908-76-54'
}
```

Output

TypeError: unhashable type: 'list'

Explanation:

Lists are **mutable**, and mutable objects cannot be hashed.
A dictionary requires its keys to be immutable (unchanging), so lists are not valid keys.

Example 10.8.2 – Correct: Tuples as Dictionary Keys

We can solve this by using tuples instead of lists.

```
phonebook = {
    ('Anna', 'Karenina'): '(123)456-78-90',
    ('Yu', 'Tsun'): '(901)234-56-78',
    ('Hans', 'Castorp'): '(321)908-76-54'
}
print(phonebook)
```

Output

```
{
 ('Hans', 'Castorp'): '(321)908-76-54',
 ('Yu', 'Tsun'): '(901)234-56-78',
 ('Anna', 'Karenina'): '(123)456-78-90'
}
```

Each **tuple key** uniquely identifies a person by their first and last name.

Accessing a Value Using Tuple Key print(phonebook[('Hans', 'Castorp')])

Output

(321)908-76-54

Adding New Entries

```
phonebook[('Leo', 'Tolstoy')] = '(444)222-33-11'
```

```
print(phonebook)
```

Output

```
{
 ('Hans', 'Castorp'): '(321)908-76-54',
 ('Yu', 'Tsun'): '(901)234-56-78',
 ('Anna', 'Karenina'): '(123)456-78-90',
 ('Leo', 'Tolstoy'): '(444)222-33-11'
}
```

Why Tuples Work as Keys

- A dictionary key must be **immutable and hashable**.
- Since tuples cannot change once created, they qualify as valid keys.
- Lists fail this property because their contents can be modified at any time.

10.9 DICTIONARY METHOD ITEMS(), REVISITED

The items() method is an important dictionary function that returns a **view object** containing all **key–value pairs** as **tuples**.

Syntax

```
dictionary.items()
```

Returns:
A view of all (key, value) pairs in the dictionary.

Example – Viewing Key–Value Tuples

```
for entry in phonebook.items():
    print(entry)
```

Output

```
(('Hans', 'Castorp'), '(321)908-76-54')
(('Yu', 'Tsun'), '(901)234-56-78')
(('Anna', 'Karenina'), '(123)456-78-90')
```

(('Leo', 'Tolstoy'), '(444)222-33-11')

Each dictionary entry is represented as a **tuple** containing the key and its corresponding value.

Example – Iterating and Unpacking Key–Value Tuples

We can extract the key and value separately during iteration.

```
for (first, last), number in phonebook.items():
    print(f"{first} {last}: {number}")
```

Output

Hans Castorp: (321)908-76-54

Yu Tsun: (901)234-56-78

Anna Karenina: (123)456-78-90

Leo Tolstoy: (444)222-33-11

Explanation

- Each element of phonebook.items() is a **tuple (key, value)**.
- The key itself is also a **tuple (first, last)**.
- Unpacking allows direct access to both the first and last name components.

Example 10.9.3 – Converting Items to a List of Tuples

```
print(list(phonebook.items()))
```

Output

```
[
 (('Hans', 'Castorp'), '(321)908-76-54'),
 (('Yu', 'Tsun'), '(901)234-56-78'),
 (('Anna', 'Karenina'), '(123)456-78-90'),
 (('Leo', 'Tolstoy'), '(444)222-33-11')
]
```

This representation shows how the dictionary internally stores key–value mappings as tuples.

Example 10.9.4 – Dictionary from a List of Tuples

A dictionary can also be **created** directly from a list of key–value tuples.

```
data = [
    (('John', 'Keats'), '(222)333-44-55'),
    (('Percy', 'Shelley'), '(111)555-99-00')
]
new_phonebook = dict(data)
print(new_phonebook)
```

Output

```
{
 ('John', 'Keats'): '(222)333-44-55',
 ('Percy', 'Shelley'): '(111)555-99-00'
}
```

Key Points

Feature	Description
items()	Returns key–value pairs as tuples.
Unpacking	Enables simultaneous access to keys and values.
Tuple as Key	Tuples used as immutable dictionary keys.
Tuple in items()	Each pair is represented as a tuple (key, value).

Practical Use Case

You can easily **iterate**, **search**, or **export** structured dictionary data:

```
for (first, last), number in sorted(phonebook.items()):
    print(f"{last}, {first} — {number}")
```

Output

Castorp, Hans — (321)908-76-54
Karenina, Anna — (123)456-78-90
Tolstoy, Leo — (444)222-33-11
Tsun, Yu — (901)234-56-78

10.10 SUMMARY

Tuples enable integer-based indexing and duplicate elements, which improves data organization and retrieval. They can be defined with or without parentheses; however, without parentheses, a following comma is required to represent a tuple. Tuples are best used for their original purpose; misapplication can result in inefficiencies, such as substituting lists, sets, or dictionaries.

To ensure efficient data processing and manipulation, choose the suitable data structure after carefully considering the use cases.

10.11 TECHNICAL TERMS

Tuple, Indexing, Negative Indexing, Max, Min , Count,.Index, and Slicing

10.12 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the concept of tuples and how they differ from lists.
2. Discuss the advantages of using tuples in Python.
3. Why can tuples be dictionary keys, but lists cannot?
4. Explain the items() method of a dictionary with an example.
5. Write a program to create a phone directory using tuple keys and string values.

Short Notes

1. Write about Class tuple in Python.
2. Explain about Immutability and hashability.
3. How can Tuples as dictionary keys.
4. Describe about Tuple functions and methods.
5. Discuss about the role of items() in dictionary iteration.

10.13 SUGGESTED READINGS

1. **Ljubomir Perković**, *Introduction to Computing Using Python: An Application Development Focus*, Wiley, 2012.
2. **Reema Thareja**, *Python Programming Using Problem-Solving Approach*, Oxford University Press.
3. **Mark Lutz**, *Learning Python*, O'Reilly Media.
4. **Eric Matthes**, *Python Crash Course*, No Starch Press.
5. **Al Sweigart**, *Automate the Boring Stuff with Python*, No Starch Press.

Dr. Kampa Lavanya

LESSON- 11

SET

AIMS AND OBJECTIVES

After completing this chapter, the learner will be able to:

1. Explain the concept and features of the Python set data type.
2. Create sets using curly braces {} and the set() constructor.
3. Apply set operators and methods to perform mathematical and logical operations.
4. Differentiate between mutable sets and immutable frozensets.
5. Use sets to remove duplicates, test membership, and compare data collections.
6. Implement real-world problem solutions using set operations and functions in Python.

STRUCTURE

11.1 INTRODUCTION

Python's set data type is another powerful built-in collection class used to store unordered, unique, and immutable items.

It represents the mathematical concept of a *set* — a group of elements with no repetitions.

Sets are very useful in programs that involve:

- Duplicate removal (e.g., cleaning data),
- Membership testing (checking if an element exists),
- Mathematical set operations (union, intersection, etc.),
- Fast lookups using hashing.

A set in Python is an unordered collection of unique, immutable objects enclosed in curly braces {}.

Syntax:

set_name = {element1, element2, element3, ...}

Unlike lists or tuples, sets cannot contain duplicate elements or mutable items (like lists or dictionaries).

Example – Creating a Simple Set

```
phonebook1 = {'123-45-67', '234-56-78', '345-67-89'}
print(phonebook1)
print(type(phonebook1))
```

Output

```
{'123-45-67', '234-56-78', '345-67-89'}
<class 'set'>
```

Explanation:

- The curly braces {} indicate a set.
- The order of items may differ since sets are *unordered*.
- The type() function confirms that it's a set object.

Handling Duplicates

If a set is defined with duplicate items, Python automatically removes them.

Example

```
phonebook1 = {'123-45-67', '234-56-78', '345-67-89',
        '123-45-67', '345-67-89'}
print(phonebook1)
```

Output

{'123-45-67', '234-56-78', '345-67-89'}

Why Use Sets?

Purpose	Advantage
Duplicate removal	Automatically removes repeated entries
Mathematical operations	Built-in support for union, intersection, etc.
Fast lookup	Membership check is faster than lists
Hashable keys	Can use immutable sets (frozensets) as dictionary keys

11.2 CHARACTERISTICS OF SETS

- Unordered: The elements have no fixed order; indexing and slicing are not supported.
- No Duplicates:Each element appears only once.
- Mutable Container:You can add or remove elements after creation.
- Immutable Elements:Each element inside the set must be an immutable object such as integers, strings, or tuples.
- Efficient Membership Testing:Checking if an element exists is very fast due to internal hashing.

11.3 CREATING SETS

Python provides multiple ways to create sets.

11.3.1 Using Curly Braces {}

```
A = {10, 20, 30}
print(A)
```

Output

{10, 20, 30}

11.3.2 Using set() Constructor

```
B = set(['apple', 'banana', 'cherry'])
print(B)
```

Output

{'banana', 'cherry', 'apple'}

11.3.3 Creating an Empty Set

Empty sets must be created using the set() function, not {}.

```
empty = set()
print(empty)
print(type(empty))
```

Output

```
set()
<class 'set'>
```

{} creates an empty dictionary, not a set.

11.3.4 Creating a Set from a String

```
chars = set("banana")
print(chars)
```

Output

```
{'b', 'n', 'a'}
```

11.4 ACCESSING ELEMENTS

Since sets are unordered, we **cannot use indexing or slicing**. We can access elements only through:

- **Iteration** using loops, or
- **Membership testing** using in / not in.

Example – Iterating Over a Set

```
colors = {'red', 'green', 'blue'}
for c in colors:
    print(c)
```

Output

```
red
blue
green
```

Example – Membership Testing

```
colors = {'red', 'green', 'blue'}
print('red' in colors)
print('yellow' not in colors)
```

Output

True

True

11.5 SET OPERATORS

Python implements classical set theory operations using operators.

Operation	Operator	Description
Membership	in, not in	Check element presence
Length	len()	Number of elements
Equality	==, !=	Compare sets
Subset/Superset	<, <=, >, >=	Relationship between sets
Union	`	`
Intersection	&	Common elements
Difference	-	Elements in one but not the other
Symmetric Difference	^	Elements in one or the other, not both

Example – Membership and Length

```
phonebook1 = {'123-45-67', '234-56-78', '345-67-89'}
print('123-45-67' in phonebook1)
print('456-78-90' not in phonebook1)
print(len(phonebook1))
```

Output

True

True

3

Example – Equality and Comparison

```
A = {'a', 'b', 'c'}
B = {'a', 'b', 'c'}
C = {'a', 'b'}
print(A == B)
print(C < A)
print(A > C)
```

Output

True

True

True

Example – Union, Intersection, Difference, Symmetric Difference

```
A = {1, 2, 3}
B = {3, 4, 5}
print(A | B)
print(A & B)
print(A - B)
print(A ^ B)
```

Output

```
{1, 2, 3, 4, 5}
{3}
{1, 2}
{1, 2, 4, 5}
```

11.6 SET METHODS

In addition to operators, Python provides rich methods for manipulating sets.

Method	Description
add(x)	Adds an element x.
update(iterable)	Adds multiple elements.
remove(x)	Removes x; error if not found.
discard(x)	Removes x if present, no error otherwise.
pop()	Removes and returns a random element.
clear()	Removes all elements.

Example – add()

```
phonebook3 = {'345-67-89', '456-78-90'}

phonebook3.add('123-45-67')

print(phonebook3)
```

Output

{'123-45-67', '345-67-89', '456-78-90'}

Example – remove() and discard()

```
colors = {'red', 'green', 'blue'}
colors.remove('green')
colors.discard('yellow')  # No error even if not present
print(colors)
```

Output

{'red', 'blue'}

Example 11.6.3 – clear()

```
phonebook3.clear()
print(phonebook3)
```

Output

set()

Example – update() and pop()

```
S = {10, 20}
S.update([30, 40, 50])
print(S)
print(S.pop())
print(S)
```

Output

{40, 10, 50, 20, 30}

40

{10, 50, 20, 30}

11.7 SET RELATIONSHIP METHODS

Method	Description
issubset(t)	True if all elements of s are in t
issuperset(t)	True if s contains all elements of t
isdisjoint(t)	True if sets have no elements in common

Example

```
A = {1, 2, 3}

B = {1, 2, 3, 4}

print(A.issubset(B))

print(B.issuperset(A))

print(A.isdisjoint({5, 6}))
```

Output

```
True

True

True
```

11.8 BUILT-IN FUNCTIONS

Function	Purpose
len(s)	Number of elements
max(s)	Largest element
min(s)	Smallest element
sum(s)	Sum of numeric elements
sorted(s)	Returns sorted list

Example

```
S = {10, 2, 8, 4}
print(len(S))
print(max(S))
print(min(S))
print(sum(S))
print(sorted(S))
```

Output

```
4
10
2
24
[2, 4, 8, 10]
```

11.9 FROZEN SETS

A frozenset is the immutable version of a set.

Elements cannot be added or removed after creation.

Example

```
A = frozenset([1, 2, 3])
B = frozenset([3, 4, 5])
print(A | B)
print(A & B)
```

Output

```
frozenset({1, 2, 3, 4, 5})
frozenset({3})
```

11.10 APPLICATIONS OF SETS

1. Removing Duplicates

```
numbers = [1, 2, 2, 3, 3, 4]
print(set(numbers))
```

2. Common Elements

```
A = {'apple', 'banana'}
B = {'banana', 'mango'}
print(A & B)
```

3. Filtering Data

```
text = "Python is powerful and Python is easy"
unique_words = set(text.split())
print(unique_words)
```

4. Fast Membership Checking

```
vowels = {'a', 'e', 'i', 'o', 'u'}
print('e' in vowels)
```

11.11 SUMMARY

- Set is an unordered collection of unique immutable elements.
- Supports mathematical operations like union, intersection, difference, and symmetric difference.
- Provides methods for addition, removal, and clearing.
- frozenset is the immutable counterpart of set.
- Sets are ideal for duplicate removal and membership testing.

11.12 TECHNICAL TERMS

Set, Unique Elements, Unordered, Mutable, Immutable Element, Membership Testing, Union, Intersection, Difference, Symmetric Difference, frozenset, Hashing, Subset, Superset, Disjoint Set.

11.13 SELF-ASSESSMENT QUESTIONS

Essay Type Questions

1. Explain the concept of a **set** in Python. How does it differ from other collection data types such as lists and tuples?
2. Discuss the **characteristics and properties** of sets in Python with suitable examples.
3. Describe the various **set operators** available in Python. Illustrate each with a program example.
4. Explain in detail the **methods** supported by the set class for adding, removing, and updating elements.
5. What are **frozen sets**? How do they differ from normal sets? Give examples.

6. Demonstrate the use of **set relationships** such as subset, superset, and disjoint sets with code snippets.
7. Discuss the **advantages and applications** of sets in Python programming. Provide at least three real-life examples.
8. Write a Python program to perform all **set operations** (union, intersection, difference, and symmetric difference) on two given sets.
9. Compare the use of sets for **duplicate removal** versus using lists.
10. Explain how **membership testing** works in sets and why it is more efficient compared to lists or tuples.

Short Answer Questions

1. What are the key properties of a Python set?
2. Write the syntax to create an empty set and a set from a list.
3. Mention any four methods of the set class.
4. Differentiate between the remove() and discard() methods.
5. What does the clear() method do?
6. How is the len() function used with sets?
7. What is the difference between a set and a frozenset?
8. What is the purpose of the isdisjoint() method?
9. Explain the function of the ^ (symmetric difference) operator.
10. What happens when duplicate elements are inserted into a set?

11.14 SUGGESTED READINGS

1. **Ljubomir Perković**, *Introduction to Computing Using Python: An Application Development Focus*, Wiley, 2012.
2. **Reema Thareja**, *Python Programming: Using Problem-Solving Approach*, Oxford University Press, 2018.
3. **Mark Lutz**, *Learning Python*, 5th Edition, O'Reilly Media, 2013.
4. **Eric Matthes**, *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, No Starch Press, 2019.
5. **Al Sweigart**, *Automate the Boring Stuff with Python*, No Starch Press, 2020.
6. **Allen B. Downey**, *Think Python: How to Think Like a Computer Scientist*, Green Tea Press, 2015.
7. **Charles Severance**, *Python for Everybody: Exploring Data in Python 3*, CreateSpace Independent Publishing, 2016.

Dr. Kampa Lavanya

LESSON- 12
RANDOMNESS

AIMS AND OBJECTIVES

After studying this chapter, the learner will be able to:

- Explain what character encoding means and why it is essential in digital text representation.
- Differentiate between ASCII, Unicode, and UTF-8 encoding systems.
- Demonstrate how Python internally stores and manipulates strings using Unicode.
- Apply string encoding and decoding methods in Python.
- Describe the purpose of the random module in Python.
- Generate random integers, floating-point numbers, and sequences.
- Implement randomization in simulations, games, and sampling problems.

STRUCTURE

12.1 INTRODUCTION

Two fundamental areas are explored in this chapter:

1. **Character Encoding**, which defines how text characters are represented as numbers inside a computer.
2. **Randomness and the random Module**, which allows programs to behave unpredictably or simulate random events.

Every program that processes text must deal with **encodings**, and every simulation or game that imitates chance must use **randomness**. Python provides robust support for both its built-in Unicode string system and the standard random library.

12.2 CHARACTER ENCODINGS AND STRINGS

A computer can store and manipulate only numbers. To represent letters, digits, and symbols, each character must be **encoded** as a numeric value. This numeric representation is called a **character code**.

When we write:

```
message = "Hello"
```

Python internally converts each letter of "Hello" into numerical values according to a specific **character encoding scheme**.

12.2.1 Character Encodings

A **character encoding** is a mapping between characters and the numeric values (code points) that represent them.

For example:

Character	Decimal Code	Binary (8 bits)
A	65	01000001
B	66	01000010
C	67	01000011

Each character corresponds to a unique binary pattern stored in memory or transmitted between systems.

Without a standard encoding, computers could not exchange text reliably.

12.2.2 ASCII (American Standard Code for Information Interchange)

ASCII was one of the earliest and most influential encoding standards, developed in the 1960s. It uses **7 bits** to represent **128 characters**, covering:

- Upper- and lower-case English letters
- Digits 0–9
- Basic punctuation symbols
- Control characters (e.g., newline, tab)

Example – ASCII Codes

Character	Decimal	Binary	Hexadecimal
A	65	01000001	0x41
a	97	01100001	0x61
0	48	00110000	0x30
Space	32	00100000	0x20

Python's ord() function returns the code point of a character, and chr() converts a number back to a character.

```
print(ord('A'))     # 65
print(chr(65))      # 'A'
```

ASCII served well for English text but failed to handle accented letters or non-Latin scripts.

Example – Viewing Character Codes in Different Number Systems

To understand how Python represents characters internally, we can write a small function encoding() that accepts a string and prints each character's **ASCII (decimal)**, **hexadecimal**, and **binary** code values.

```
def encoding(s):
   print(f"{'Char':<6}{'Decimal':<10}{'Hex':<8}{'Binary'}")
   for ch in s:
      dec = ord(ch)                  # Decimal (Unicode code point)
      hx = format(dec, '02x')        # Hexadecimal representation
      bin_val = format(dec, '08b')   # Binary (8-bit) representation
      print(f"{ch:<6}{dec:<10}{hx:<8}{bin_val}")
```

Program Execution

```
>>> encoding('dad')
```

Output

Char	Decimal	Hex	Binary
d	100	64	1100100
a	97	61	1100001
d	100	64	1100100

Explanation

- **ord()** returns the **Unicode code point** (integer) for each character.
  For example, ord('d') = 100 and ord('a') = 97.
- **format(x, '02x')** converts the integer x into a **2-digit hexadecimal** string.
  - The value 100 in decimal equals 64 in hexadecimal.
- **format(x, '08b')** converts the integer into an **8-bit binary** string.
  - For d, this is 1100100.

Each character in 'dad' is thus represented numerically in the computer's memory, and these numbers correspond to the **ASCII/Unicode** encoding values.

Python provides two complementary built-in functions for working with character codes:

Function	**Description**
ord(char)	Returns the numeric Unicode code point of the character char.
chr(number)	Returns the character that corresponds to the Unicode code point number.

These functions are exact inverses of each other.

That is:

chr(ord('A')) → 'A'

ord(chr(65)) → 65

Example

```
>>> chr(97)
'a'
>>> chr(65)
'A'
>>> chr(8364)
'€'
>>> chr(937)
'Ω'
```

Output

```
'a'
'A'
'€'
'Ω'
```

Explanation

- The integer **97** corresponds to the lowercase letter **'a'** in the Unicode (and ASCII) table.
- **65** corresponds to **'A'**, the uppercase letter.
- **8364** represents the **Euro symbol (€)**.
- **937** corresponds to the **Greek capital letter Omega (Ω)**.

These code points are defined by the **Unicode standard**, which allows Python to support characters from virtually every language.

Function	Example	Output	Purpose
ord('A')	Returns Unicode code point	65	Character → Code
chr(65)	Returns character	'A'	Code → Character

You can easily build small encoding-decoding utilities in Python:

```
text = "ABC"
codes = [ord(c) for c in text]
decoded = ''.join(chr(i) for i in codes)
print(codes)
print(decoded)
```

Output

```
[65, 66, 67]
ABC
```

12.2.3 Unicode

To overcome ASCII's limitations, Unicode was introduced as a universal standard that assigns a unique number to every character in every language and symbol set.

- Each character has a code point, written as U+XXXX (hexadecimal).
- Unicode currently defines over 140,000 characters, covering scripts worldwide.

Example – Unicode Code Points

Character	Unicode Code Point	Name
A	U+0041	Latin Capital Letter A
Ω	U+03A9	Greek Capital Letter Omega
中	U+4E2D	CJK Unified Ideograph
😊	U+1F60A	Smiling Face Emoji

Python 3 strings (str) are Unicode by default, meaning they can represent any text from any language.

```
s = "Ωmega 😊"
print(s)
print(len(s))
```

Output

```
Ωmega 😊
7
```

12.2.4 UTF-8 Encoding for Unicode Characters

Unicode specifies code points, but the computer still needs a binary representation for storage and transmission.

UTF-8 (Unicode Transformation Format – 8-bit) is the most common encoding form used today.

Features of UTF-8:

1. Variable-length encoding using **1 to 4 bytes**.
2. Backward compatible with ASCII (0–127).
3. Efficient for English text, flexible for global scripts.

Character	Code Point	UTF-8 Bytes (Hex)
A	U+0041	41
ñ	U+00F1	C3 B1
€	U+20AC	E2 82 AC
😊	U+1F60A	F0 9F 98 8A

Example – Encoding and Decoding Strings

```
text = "Python is fun 😊"
encoded = text.encode('utf-8')
print(encoded)
decoded = encoded.decode('utf-8')
print(decoded)
```

Output

```
b'Python\xe2\x80\xa2fun\xf0\x9f\x98\x8a'
Python is fun 😊
```

12.3 WORKING WITH ENCODED STRINGS IN PYTHON

Python provides built-in methods to handle different encodings.

Method	Purpose
encode(encoding)	Converts a string into bytes.
decode(encoding)	Converts bytes back into a string.

Example

```
msg = "Café"
b = msg.encode('utf-8')
print(b)
print(b.decode('utf-8'))
```

Output

```
b'Caf\xc3\xa9'
Café
```

If the wrong encoding is used while decoding, Python raises a UnicodeDecodeError.

12.4 THE RANDOM MODULE

Programs often require random behavior—rolling dice, shuffling cards, generating random IDs, or simulating uncertain events.

Python provides these capabilities in the **random** module.

To use it:

```
import random
```

All random values are **pseudorandom**—they come from deterministic algorithms but appear random for most applications.

12.4.1 Choosing a Random Integer

Use random.randint(a, b) to return an integer N such that $a \leq N \leq b$.

```
import random
num = random.randint(1, 6)
print("Dice rolled:", num)
```

Output

```
Dice rolled: 4
```

Other related functions:

Function	Description
randrange(start, stop, step)	Choose integer from a range.
getrandbits(k)	Return integer with k random bits.

Example

```
print(random.randrange(0, 10, 2))   # Even numbers 0–8
print(random.getrandbits(8))        # Random 8-bit number
```

Example:

```
import random
print("Simulating 10 dice rolls:")
for i in range(10):
  print(random.randrange(1, 7))
```

Output

Simulating 10 dice rolls:

3

5

6

2

4

1

2

6

5

3

Example – Implementing a Number Guessing Game

The following program implements a simple **interactive number guessing game** using Python's random module.

The program randomly chooses a number between 0 and n - 1 and repeatedly asks the user to guess it.

Each time the player guesses incorrectly, the program prints a **hint**:

- "Too low." if the guess is smaller than the secret number.
- "Too high." if the guess is larger.

When the player guesses correctly, the program prints "You got it." and stops.

Program: guess() Function

```
import random
def guess(n):
    """Interactive number guessing game."""
    # Step 1: Choose a random number
    secret = random.randrange(0, n)
        print(f"I'm thinking of a number between 0 and {n - 1}. Can you guess it?")

    while True:
        # Step 2: Ask user for a guess
        user_input = input("Enter your guess: ")

        # Validate input
        if not user_input.isdigit():
            print("Please enter a valid integer.")
            continue

        guess_num = int(user_input)

        # Step 3: Compare with secret number
        if guess_num < secret:
            print("Too low.")
        elif guess_num > secret:
            print("Too high.")
        else:
            print("You got it!")
            break
```

Program Execution Example

```
>>> guess(10)
I'm thinking of a number between 0 and 9. Can you guess it?
Enter your guess: 5
Too high.
Enter your guess: 2
Too low.
```

Enter your guess: 3

You got it!

Step	Operation	Description
1	random.randrange(0, n)	Selects a random number between 0 and n–1.
2	input()	Prompts user to enter a guess.
3	Comparison	If guess < secret → "Too low."; if guess > secret → "Too high."; else "You got it."
4	Loop	Continues until the correct number is guessed.

12.4.2 Choosing a Random "Real" (Floating Point Value)

For fractional random values:

Function	Description
random()	Returns float $0.0 \leq x < 1.0$
uniform(a, b)	Returns float $a \leq x \leq b$
triangular(low, high, mode)	Weighted random float

Example x = random.random()

y = random.uniform(1.5, 6.5)

print("Random fraction:", x)

print("Random real number:", y)

12.4.3 Shuffling, Choosing, and Sampling at Random

The random module also handles random operations on sequences.

Function	Description
choice(seq)	Returns one random element.
choices(seq, k=n)	Returns list of n elements (with replacement).
sample(seq, k)	Returns k unique elements (without replacement).
shuffle(seq)	Randomly reorders elements of a list in place.

Example – Random Choice and Shuffle

```
names = ['Alice', 'Bob', 'Charlie', 'Diana']
print(random.choice(names))     # One name
random.shuffle(names)
print(names)                    # Shuffled order
```

Example – Sampling

```
lottery = list(range(1, 51))
```

```
winners = random.sample(lottery, 6)
print("Winning numbers:", winners)
```

Output

```
Winning numbers: [7, 18, 25, 33, 42, 49]
```

Seeding the Random Number Generator

To reproduce the same random sequence, use random.seed(value).

```
random.seed(10)
print(random.randint(1, 100))
```

Each run with the same seed yields identical output—useful for testing and debugging.

12.5 APPLICATIONS OF RANDOMNESS

1. **Games and Simulations**
   Rolling dice, card games, and random moves in games use random integers.
2. **Monte Carlo Methods**
   Estimating π or probabilities through repeated random sampling.
3. **Random Sampling in Statistics**
   Selecting random subsets from data for analysis.
4. **Security and Token Generation**
   Creating random passwords or identifiers.

Example – Estimating π using Monte Carlo Simulation

```
import random, math
inside = 0
n = 100000
for i in range(n):
   x = random.random()
   y = random.random()
   if x**2 + y**2 <= 1:
      inside += 1
pi_estimate = 4 * inside / n
print("Estimated π:", pi_estimate)
```

Output

```
Estimated π: 3.1416
```

Example – Random Password Generator

```
import random, string
chars = string.ascii_letters + string.digits + "!@#$%"
password = "".join(random.choice(chars) for _ in range(10))
print("Random Password:", password)
```

Output

```
Random Password: aX4!qM8zT@
```

12.6 SUMMARY

- **Character encoding** maps characters to numeric code points.
- **ASCII** encodes 128 characters using 7 bits.
- **Unicode** extends this to global scripts; **UTF-8** is the common binary encoding.
- Python 3 strings are **Unicode** by default.
- The **random** module generates pseudorandom integers, floats, and selections.
- Functions such as randint(), random(), choice(), and shuffle() provide flexible randomization.
- Randomness supports games, simulations, and statistical modeling.

12.7 TECHNICAL TERMS

Character Encoding, ASCII, Unicode,UTF-8,Code Point, Byte Sequence, Encoding / Decoding, Random Number Generator, Seed, Uniform Distribution, Sampling, Monte Carlo Simulation, Random Shuffle, Deterministic Algorithm.

12.8 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Define character encoding. Describe the differences between ASCII, Unicode, and UTF-8.
2. How does Python represent Unicode characters internally?
3. Explain the importance of encoding and decoding operations with suitable examples.
4. What is the random module? Describe at least five of its functions with examples.
5. Discuss the role of random numbers in simulations and games.
6. Write a Python program to generate a random password of given length.
7. Explain how seeding affects random number generation.
8. Demonstrate how to shuffle and sample random elements from a list.
9. Compare pseudorandom and true random numbers.
10. Explain how UTF-8 ensures compatibility with ASCII.

Short Notes

1. Code Points and Bytes
2. ord() and chr() functions
3. Unicode in Python 3
4. random.randint() vs random.random()
5. random.choice() and random.sample()

6. Random Seed and Reproducibility
7. UTF-8 Variable Length Encoding
8. Monte Carlo Method

12.9 SUGGESTED READINGS

1. **Ljubomir Perković**, *Introduction to Computing Using Python: An Application Development Focus*, Wiley, 2012.
2. **Reema Thareja**, *Python Programming: Using Problem-Solving Approach*, Oxford University Press.
3. **Mark Lutz**, *Learning Python*, O'Reilly Media.
4. **Eric Matthes**, *Python Crash Course*, No Starch Press.
5. **Al Sweigart**, *Automate the Boring Stuff with Python*, No Starch Press.
1. **David Beazley and Brian Jones**, *Python Cookbook*, O'Reilly Media.

Dr. Kampa Lavanya

LESSON- 13

OBJECT ORIENTED PROGRAMMING

AIMS AND OBJECTIVES

After studying this chapter, the learner will be able to:

- Explain the fundamental concepts of object-oriented programming (OOP).
- Define new Python classes and understand class structure.
- Create and use user-defined classes with attributes and methods.
- Apply constructors, instance variables, and class variables.
- Implement operator overloading to make custom classes behave like built-ins.
- Design new container classes such as a deck of cards or queue.
- Apply inheritance to derive new classes and reuse existing functionality.
- Define and use user-defined exceptions to handle program-specific errors.

STRUCTURE

13.1 INTRODUCTION – FUNDAMENTAL CONCEPTS

Object-Oriented Programming (OOP) models programs as a collection of objects—entities that combine data (attributes) and behavior (methods).

Instead of writing functions that act on global data, OOP organizes related data and operations within classes.

Core OOP Concepts

Concept	Description
Class	Blueprint defining the structure and behavior of objects.
Object (Instance)	Individual entity created from a class.
Encapsulation	Bundling of data and related methods into one unit.
Abstraction	Hiding implementation details, showing only relevant features.
Inheritance	Creating new classes that reuse attributes and methods of existing ones.
Polymorphism	Ability to use the same operation on objects of different types.

13.2 DEFINING A NEW PYTHON CLASS

A Python class is defined using the keyword class, followed by the class name and a colon.

```
class Point:
    """Represents a point in 2D space."""
```

```
def __init__(self, x, y):
    self.x = x
    self.y = y
```

Here, __init__() is the constructor—a special method that runs when an object is created.

Creating Instances

```
p1 = Point(2, 3)
p2 = Point(5, 6)
print(p1.x, p1.y)
```

Output

```
2 3
```

13.2.1 Methods of Class Point

Methods are functions defined within a class and automatically receive the instance (self) as the first parameter.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def display(self):
        print(f"({self.x}, {self.y})")
p = Point(2, 3)
p.move(1, 2)
p.display()
```

Output

```
(3, 5)
```

13.2.2 A Class and Its Namespace

Each class defines a namespace, a mapping of names to objects—variables, constants, and methods—local to the class.

13.2.3 Every Object Has an Associated Namespace

Every object (instance) maintains its own namespace for storing instance variables. Accessing attributes uses the dot operator: **object. attribute.**

13.2.4 Implementation of Class Point

```
class Point:
    count = 0                # Class variable
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        Point.count += 1


    def __repr__(self):
        return f"Point({self.x}, {self.y})"
p1 = Point(1, 1)
p2 = Point(3, 4)
print(p1, p2)
print("Number of points:", Point.count)
```

Output

```
Point(1, 1) Point(3, 4)
Number of points: 2
```

13.2.5 Instance and Class Variables

- Instance variables (self.x) belong to individual objects.
- Class variables (Point.count) belong to the class as a whole.

Instance variables are attributes that are unique to each object created from a class. They are defined using the prefix self. inside class methods (most commonly within the constructor __init__()).

Example

```
class Student:
    def __init__(self, name, rollno):
        self.name = name        # instance variable
        self.rollno = rollno    # instance variable
```

```
s1 = Student("Asha", 101)
s2 = Student("Rahul", 102)

print("Student 1:", s1.name, s1.rollno)
print("Student 2:", s2.name, s2.rollno)
```

Output

```
Student 1: Asha 101
Student 2: Rahul 102
```

Explanation:

- self.name and self.rollno are **instance variables**.
- Each object (s1, s2) has its **own copy** of these variables.
- Changing s1.name does not affect s2.name.

```
s1.name = "Anita"
print("Updated s1:", s1.name)
print("Unchanged s2:", s2.name)
```

Output

```
Updated s1: Anita
Unchanged s2: Rahul
```

Class variables are attributes that belong to the **class itself**, not to any individual instance. They are declared **outside all methods** but **inside the class definition**.

Example

```
class Student:
    school_name = "Greenwood High"     # class variable

    def __init__(self, name, rollno):
        self.name = name               # instance variable
        self.rollno = rollno           # instance variable

s1 = Student("Asha", 101)
s2 = Student("Rahul", 102)
```

```
print("Student 1 School:", s1.school_name)
print("Student 2 School:", s2.school_name)
print("Accessing through class:", Student.school_name)
```

Output

Student 1 School: Greenwood High

Student 2 School: Greenwood High

Accessing through class: Greenwood High

Explanation:

- school_name is a **class variable**, shared by all objects.
- Any change made through the class name affects all objects.

```
Student.school_name = "Sunrise Academy"
print(s1.school_name)
print(s2.school_name)
```

Output

Sunrise Academy

Sunrise Academy

Table 13.1 Comparative Summary

Feature	**Instance Variable**	**Class Variable**
Defined in	Inside methods using self.	Inside class, outside methods
Belongs to	Each object (instance)	The class (shared by all instances)
Accessed using	object_name.variable	ClassName.variable or object_name.variable
Storage	Separate copy for every object	Single shared copy for all
Use case	To store unique attributes for each instance	To store common attributes across all objects

Example

```
class Point:
    count = 0                # class variable (shared)

    def __init__(self, x, y):
        self.x = x           # instance variable
```

```
        self.y = y              # instance variable
        Point.count += 1        # modify class variable

p1 = Point(1, 2)
p2 = Point(3, 4)

print("p1:", p1.x, p1.y)
print("p2:", p2.x, p2.y)
print("Total Points:", Point.count)
```

Output

```
p1: 1 2
p2: 3 4
Total Points: 2
```

Explanation

- x and y → Instance variables (unique to each object).
- count → Class variable (common counter shared by all instances).
- Each time a new object is created, the constructor increases Point.count by 1.

This pattern is commonly used to track:

- The **number of objects** created from a class,
- Or any **aggregate data** shared among instances.

Best Practices

1. Use **instance variables** for per-object data (e.g., student names, coordinates, employee salaries).
2. Use **class variables** for shared data (e.g., school name, total object count, global configuration).
3. Access class variables through the **class name** (not self) when updating them to avoid shadowing.

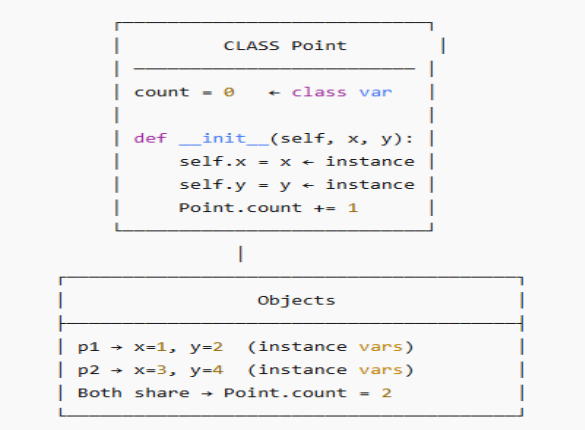


Fig 13.1 Instance and Class Variables

13.2.6 Class Definition, More Generally

A typical class contains:

```
class ClassName:
    """Docstring describing purpose."""
    class_variable = value

    def __init__(self, parameters):
        # initialize instance variables

    def method1(self):
        # perform operation
```

13.2.7 Documenting a Class

Python's docstring (""" ... """) provides built-in documentation accessible via help(ClassName).

13.3 EXAMPLES OF USER-DEFINED CLASSES

13.3.1 Overloaded Constructor

An Overloaded Constructor refers to the concept of using a single constructor method to handle multiple forms of object initialization, depending on the arguments passed when creating an object.

Unlike languages such as C++ or Java, Python does not support multiple constructors (i.e., multiple __init__ methods with different signatures).

However, constructor overloading can be simulated by providing default arguments, variable-length arguments, or conditional logic within a single __init__() definition.

Purpose

The goal of an overloaded constructor is to:

- Allow flexible object creation, depending on the data available at runtime.
- Enable objects to be initialized with different numbers or types of parameters.
- Simplify class usage by adapting to various initialization contexts.

General Syntax

```
class ClassName:
    def __init__(self, param1=None, param2=None, ...):
```

```
    # initialization code
```

Here:

- Default values (None) make parameters optional.
- The constructor adapts based on which arguments are provided

Example – Constructor with Default Parameters

```
class Circle:
    def __init__(self, radius=1):
        self.radius = radius
c1 = Circle()
c2 = Circle(5)
print(c1.radius, c2.radius)
```

Output

```
1 5
```

Example – Constructor with Conditional Logic

In some situations, the constructor must behave differently based on **argument type** or **number**.

```
class Student:
    def __init__(self, name=None, marks=None):
        if name is not None and marks is not None:
            self.name = name
            self.marks = marks
        elif name is not None:
            self.name = name
            self.marks = 0
        else:
            self.name = "Unknown"
            self.marks = 0
    def display(self):
        print(f"Name: {self.name}, Marks: {self.marks}")
```

Program Execution

```
s1 = Student("Asha", 85)
```

```
s2 = Student("Rahul")
s3 = Student()


s1.display()
s2.display()
s3.display()
```

Output

```
Name: Asha, Marks: 85
Name: Rahul, Marks: 0
Name: Unknown, Marks: 0
```

Explanation:

- The same constructor handles **three different initialization cases**.
- The if-elif-else structure allows **overloaded behavior** within a single __init__() method

Example – Constructor Using Variable-Length Arguments

You can also simulate overloading using *args (for positional arguments) and **kwargs (for keyword arguments).

```
class Rectangle:
    def __init__(self, *args):
        if len(args) == 0:
            self.length = 1
            self.breadth = 1
        elif len(args) == 1:
            self.length = self.breadth = args[0]
        elif len(args) == 2:
            self.length, self.breadth = args
        else:
            raise TypeError("Too many arguments")


    def area(self):
        return self.length * self.breadth
```

Program Execution

```
r1 = Rectangle()      # 1x1
```

```
r2 = Rectangle(4)      # 4x4
r3 = Rectangle(4, 6)   # 4x6


print(r1.area(), r2.area(), r3.area())
```

Output

1 16 24

Explanation:

- The same constructor supports multiple ways of initializing a rectangle:
  - No argument → default size
  - One argument → square
  - Two arguments → custom dimensions

Advantages of Overloaded Constructor

1. **Flexibility** – Allows different initialization formats for the same class.
2. **Convenience** – Reduces need for multiple specialized constructors.
3. **Readability** – Keeps initialization logic centralized.
4. **Error Reduction** – Avoids code duplication across constructors.

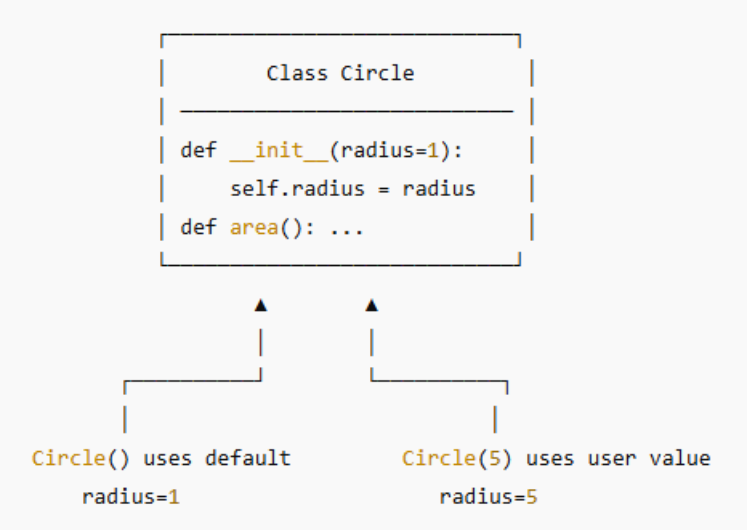


Fig An **Overloaded Constructor**

An **Overloaded Constructor** in Python:

- Is a single constructor method (__init__) that can handle **different argument lists**.
- Provides **multiple ways to initialize** objects using **default**, **optional**, or **variable arguments**.
- Is a key tool for building **flexible and reusable classes**.

13.3.2 Default Constructor

If no constructor is defined, Python provides a default one that does nothing.

Example A Class Without Constructor

```
class Student:
    def display(self):
        print("This is a student object.")
s1 = Student()   # Python calls the default constructor
s1.display()
```

Output

```
This is a student object.
```

Explanation

- The Student class does not define an __init__() constructor.
- When we write s1 = Student(), Python automatically calls the default constructor.
- The object is created successfully, and we can still access its methods.

Difference Between Default and User-Defined Constructors

Feature	Default Constructor	User-Defined Constructor
Defined by	Python automatically	Programmer explicitly
Takes Parameters	No	Yes (optional parameters possible)
Purpose	Creates object but does not initialize attributes	Initializes object data and attributes
Overridden by	User-defined __init__()	Not applicable
Use Case	When no special initialization is required	When attributes must be set during creation

13.3.3 Playing Card Class

```
class Card:
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']


    def __init__(self, value, suit):
        self.value = value
```

```
        self.suit = suit


    def __repr__(self):
        return f"{self.value} of {Card.suits[self.suit]}"
c = Card('Ace', 0)
print(c)
```

Output

```
Ace of Hearts
```

13.4 DESIGNING NEW CONTAINER CLASSES

Container classes store multiple objects and provide methods to manipulate them.

A **Container Class** is a class designed to **store multiple objects** and provide methods to **access, add, remove, or manipulate** those objects efficiently.

In other words, a container acts as a **collection or data structure** that holds other objects as its elements.

In Python, built-in container types include:

- **list**
- **tuple**
- **set**
- **dict**

However, programmers can design **user-defined container classes** to implement **custom data structures** (e.g., Deck of Cards, Queue, Stack, Bag, etc.) that meet specific requirements.

Purpose

Container classes:

- Organize data into **structured collections**.
- Allow **batch operations** on groups of items.
- Promote **data abstraction** by hiding internal details of how items are stored.
- Offer **methods** for adding, removing, searching, or iterating over elements.
- Simplify complex problems that involve managing multiple related objects.

Concept Illustration

Think of a **container** as a box that holds multiple items. Each item can be:

- A number,

- A string,
- Or even another object (instance of a class).

We don't interact with individual items directly; instead, we interact with the **container** using its **methods**.

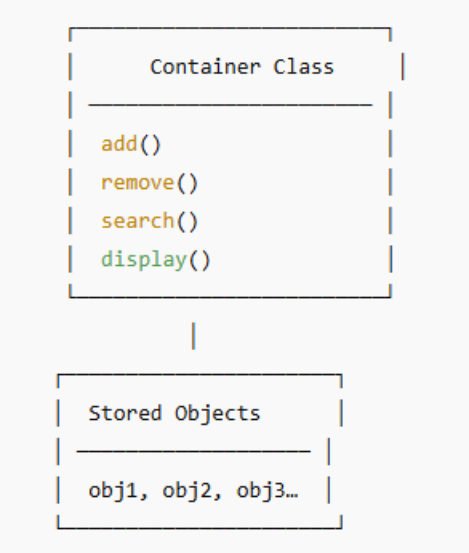


Fig 13. 3 Concept Illustration : Container Classes

Key Characteristics

Feature	Description
Aggregation	Stores multiple objects (instances of possibly different classes).
Encapsulation	Manages internal data privately, accessed only through methods.
Iteration	Often supports looping or traversal through stored objects.
Manipulation	Provides operations such as insertion, deletion, search, and retrieval.
Reusability	Can be generalized and reused in many programs (e.g., Stack, Queue).

Example – A Simple Container Class

Let's design a simple class to hold a collection of integers.

```
class NumberContainer:
    def __init__(self):
        self.numbers = []   # internal list container


    def add(self, num):
```

```
        self.numbers.append(num)

    def remove(self, num):
        if num in self.numbers:
            self.numbers.remove(num)
        else:
            print("Number not found.")

    def display(self):
        print("Numbers in container:", self.numbers)
```

Program Execution

```
c = NumberContainer()
c.add(10)
c.add(20)
c.add(30)
c.display()
c.remove(20)
c.display()
```

Output

```
Numbers in container: [10, 20, 30]
Numbers in container: [10, 30]
```

Explanation

- The class **NumberContainer** maintains a **list of numbers** internally (self.numbers).
- Methods such as add(), remove(), and display() allow controlled access to that list.
- Users of the class don't directly manipulate the list — they call methods instead, achieving **encapsulation**.

13.4.1 Deck of Cards

```
import random
class Deck:
    def __init__(self):
        self.cards = [Card(value, suit)
```

```
                for suit in range(4)
                for value in ['Ace','2','3','4','5','6','7','8','9','10','Jack','Queen','King']]
        random.shuffle(self.cards)

    def draw(self):
        return self.cards.pop()
deck = Deck()
print(deck.draw())
```

Output

7 of Clubs

13.4.2 Queue Container Class

```
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.items:
            print("Queue empty.")
            return None
        return self.items.pop(0)
```

13.5 OVERLOADED OPERATORS

Python allows classes to overload operators by defining special methods (dunder methods).

Operator	Method	Example
+	__add__	a + b
==	__eq__	a == b
str()	__str__	print(a)
repr()	__repr__	For debugging
<	__lt__	Comparison

13.5.1 Operators Are Class Methods

```
class Point:
    def __init__(self, x=0, y=0):
        self.x, self.y = x, y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
p1 = Point(2, 3)
p2 = Point(1, 1)
print(p1 + p2)
```

Output

```
Point(3, 4)
```

13.5.2 Making Class Point User Friendly

Adding a readable string form:

```
def __str__(self):
    return f"({self.x}, {self.y})"
```

13.5.3 Contract between Constructor and repr()

repr() should produce a string that can recreate the object:

```
def __repr__(self):
    return f"Point({self.x}, {self.y})"
```

13.5.4 Making the Queue Class User Friendly

```
def __repr__(self):
    return f"Queue({self.items})"
```

13.6 INHERITANCE

13.6.1 Inheriting Attributes of a Class

A subclass inherits attributes and methods from its superclass.

```
class Animal:
    def speak(self):
        print("Animal speaks")
```

```
class Dog(Animal):
    def bark(self):
        print("Woof!")
d = Dog()
d.speak()
d.bark()
```

Output

```
Animal speaks
Woof!
```

13.6.2 Overriding Superclass Methods

Method overriding occurs when a subclass (derived class) provides a new implementation of a method that already exists in its superclass (base class).

The method in the subclass has the same name, same parameters, and same return type as the one in the superclass, but performs a different or extended action.

When an overridden method is called on a subclass object, Python executes the version defined in the subclass, not the superclass.

Purpose of Method Overriding

Method overriding allows subclasses to:

1. **Modify or customize** behavior inherited from a parent class.
2. **Replace** general methods in the superclass with **specific** ones in the subclass.
3. Implement **polymorphism**, where the same method name behaves differently depending on the object type.
4. **Reuse code** by building on the base class functionality while changing only what's needed.

Example – Basic Method Overriding

```
class Animal:
    def speak(self):
        print("The animal makes a sound.")


class Dog(Animal):
```

```
    def speak(self):        # overriding superclass method
        print("The dog barks.")

class Cat(Animal):
    def speak(self):        # overriding superclass method
        print("The cat meows.")
```

Program Execution

```
a = Animal()
d = Dog()
c = Cat()

a.speak()
d.speak()
c.speak()
```

Output

```
The animal makes a sound.
The dog barks.
The cat meows.
```

13.6.3 Extending Superclass Methods

```
class Cat(Animal):
    def speak(self):
        super().speak()
        print("Cat meows")
```

Output

```
        Animal speaks
        Cat meows
```

13.6.4 Implementing Queue by Inheriting from list

```
class Queue(list):
    def enqueue(self, item):
        self.append(item)
```

```
def dequeue(self):
    if len(self)==0:
        raise IndexError("Empty queue")
    return self.pop(0)
```

13.7 USER-DEFINED EXCEPTIONS

13.7.1 Raising an Exception

```
raise ValueError("Invalid value")
```

13.7.2 Defining User-Defined Exception Classes

```
class QueueEmpty(Exception):
    """Raised when dequeue is attempted on an empty queue."""
    pass


class Queue(list):
    def dequeue(self):
        if not self:
            raise QueueEmpty("Cannot dequeue from empty queue")
        return self.pop(0)
```

Example

```
q = Queue()
try:
    q.dequeue()
except QueueEmpty as e:
    print("Error:", e)
```

Output

```
Error: Cannot dequeue from empty queue
```

13.8 SUMMARY

- OOP organizes code around objects that contain data and methods.
- Classes define the blueprint; objects are instances.
- Constructors (__init__) initialize object state.
- Operator overloading enables intuitive behavior (+, ==, etc.).
- Inheritance promotes code reuse and hierarchy.
- User-defined exceptions provide customized error handling.

13.9 TECHNICAL TERMS

Object, Class, Instance, Constructor, Method, Namespace, Encapsulation, Inheritance, Polymorphism, Operator Overloading, Superclass, Subclass, Docstring, Exception Handling, Custom Exception, Container Class.

13.10 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the fundamental principles of OOP and their importance in software design.
2. Define a Python class with example code and explain its components.
3. What is the difference between class variables and instance variables?
4. How can we overload operators in Python? Illustrate with an example.
5. Discuss inheritance and method overriding with code examples.
6. Explain the design of a Queue or Deck container class.
7. Define and raise a user-defined exception.
8. What is the relationship between repr() and the constructor?

Short Answer Questions

1. What is encapsulation?
2. Give syntax of a Python class.
3. What are dunder (double-underscore) methods?
4. What is the purpose of __init__()?
5. Define polymorphism in your own words.
6. What is the output of repr() vs str()?
7. How is inheritance implemented in Python?
8. Difference between built-in and user-defined exceptions.

13.11 SUGGESTED READINGS

1. Ljubomir Perković, *Introduction to Computing Using Python: An Application Development Focus*, Wiley, 2012.
2. Reema Thareja, *Python Programming: Using Problem-Solving Approach*, Oxford University Press.
3. Mark Lutz, *Learning Python*, O'Reilly Media.
4. Allen B. Downey, *Think Python*, Green Tea Press.
5. David Beazley & Brian Jones, *Python Cookbook*, O'Reilly Media.

Dr. Vasantha Rudramalla

LESSON- 14

OBJECTS AND THEIR USES

AIMS AND OBJECTIVES

After completing this chapter, learners will be able to:

- Explain what software objects are and how they are used in Python.
- Understand object references, mutability, and garbage collection.
- Utilize the turtle module to create visual simulations.
- Apply the principles of modular design in Python programs.
- Use Python's module system to organize programs logically.

STRUCTURE

14.1 INTRODUCTION

In imperative programming, *functions* are the basic building blocks of a program. In object-oriented programming (OOP), however, objects become the fundamental units of design, combining both data (attributes) and behavior (methods).

The concept of "objects" originated in computer simulation, where real-world entities such as cars, students, or bank accounts were modeled in software.

In the early 1970s, Alan Kay at Xerox PARC developed the programming language Smalltalk, introducing object-oriented programming as we know it. This idea later inspired the development of graphical user interfaces (GUIs) and languages such as Python, Java, and C++.

Programming Language	Programming Paradigm Supported	
	Procedural	Object-oriented
C (early 1970s)	X	
Smalltalk (1980)		X
C++ (mid 1980s)	X	X
Python (early 1990s)	X	X
Java (1995)		X
Ruby (mid 1990s)	X	X
C # (2000)	X	X

Fig 14.1 some common used programming languages

14.2 SOFTWARE OBJECTS

In object-oriented programming, a software object is a self-contained entity that combines both data (attributes or properties) and behavior (methods or functions). It is modeled after *real-world objects* that have characteristics and actions. For example, a *student object* might have data such as name, roll number, and marks, and behaviors such as register() or calculate_grade().

Objects are created (or *instantiated*) from classes, which serve as *blueprints* defining what attributes and methods an object will have. Once created, each object maintains its own copy of data, but all objects of a class share the same structure and behavior.

In Python, every data type — integers, strings, lists, even functions — is implemented as an object. This is why we can perform actions like "hello".upper() or [1,2,3].append(4): these are method calls acting on objects.

Objects interact with one another by sending and receiving messages (method calls), allowing complex systems to be built from smaller, reusable components.

Thus, software objects make programs more modular, maintainable, and intuitive, reflecting the real-world relationships between entities and their actions.

14.2.1 What is an Object?

An object in Python is a software entity that bundles:

- Attributes – data stored in *instance variables*, and
- Methods – functions that define its *behavior*.

Every object in Python (even numbers, strings, and lists) is an instance of some class.

Example:

names_list = ['Alice', 'Bob', 'Carol']

names_list.sort()

Here:

- names_list is a list object.
- sort() is a method that operates on the list.

Calling names_list.sort() sends a message to the object saying, "Sort yourself."

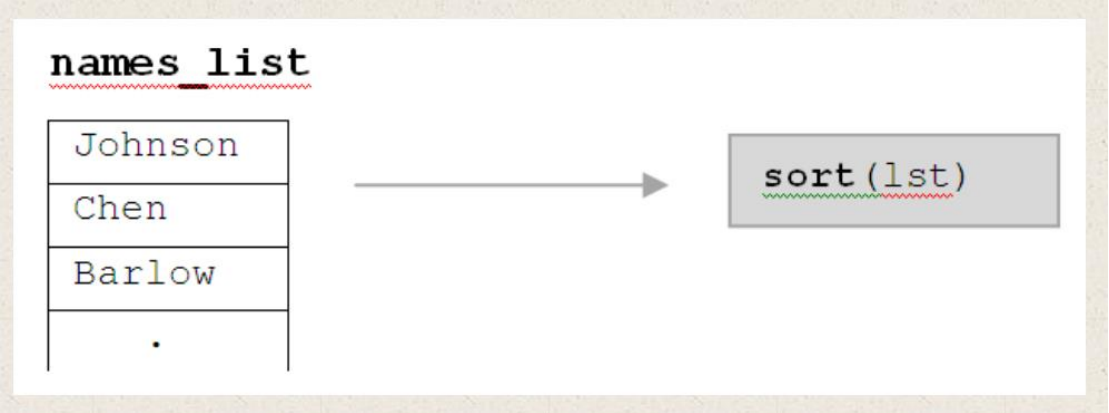


Fig 14.2 example software object name_list

14.2.2 Object References

In Python, variables do not hold actual data values — they hold references (memory addresses) to objects stored elsewhere in memory.

```
n = 10
k = n
```

Both n and k reference the same object (10) in memory.

We can verify this using the built-in id() function:

```
>>> id(n)
    505498136
>>> id(k)
```

505498136

Both have the same memory location, showing that n and k refer to the same object.

14.2.3 Memory Management and Garbage Collection

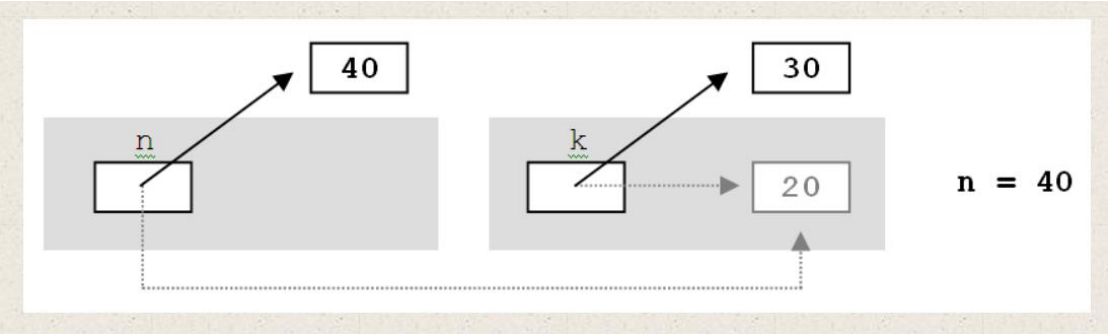
When no variable references an object anymore, Python automatically deallocates its memory through a process called garbage collection.

Example:

n = 20

n = 40 # old value 20 no longer referenced

After this, the object 20 is marked for garbage collection and its memory becomes reusable.



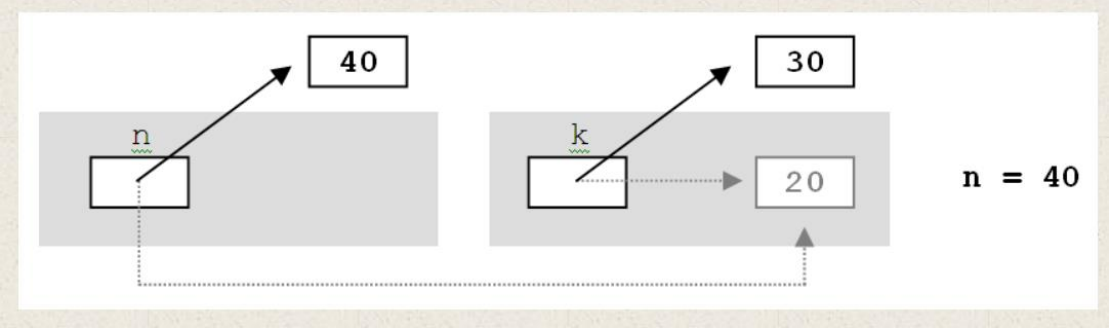


Fig 14.3 memory deallocation

Garbage collection is a method of automatically determining which locations in memory are no longer in use and deallocating them. The garbage collection process is ongoing during the execution of a Python program.

14.2.4 List Assignment and Copying

Assigning one list to another creates a reference, not a copy:

list1 = [10, 20, 30]

list2 = list1

Changing list1[0] also affects list2[0] because both reference the same object.

To make an actual copy, use the list constructor:

list2 = list(list1)

To copy nested lists completely, use:

import copy

list3 = copy.deepcopy(list1)

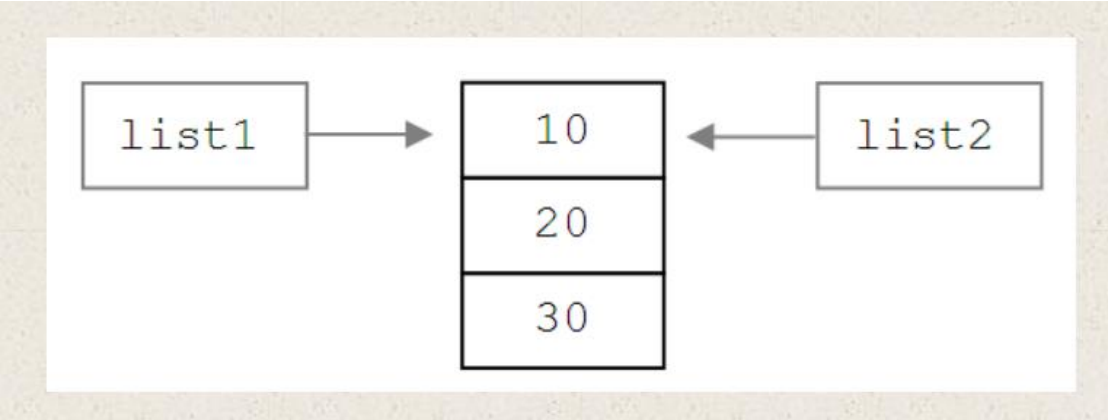


Fig 14.4 List assignment and copy

The situation is different if the list contains sublists, however.

list1 = [[10, 20] , [20, 30],[30,40]]

list2 = list1

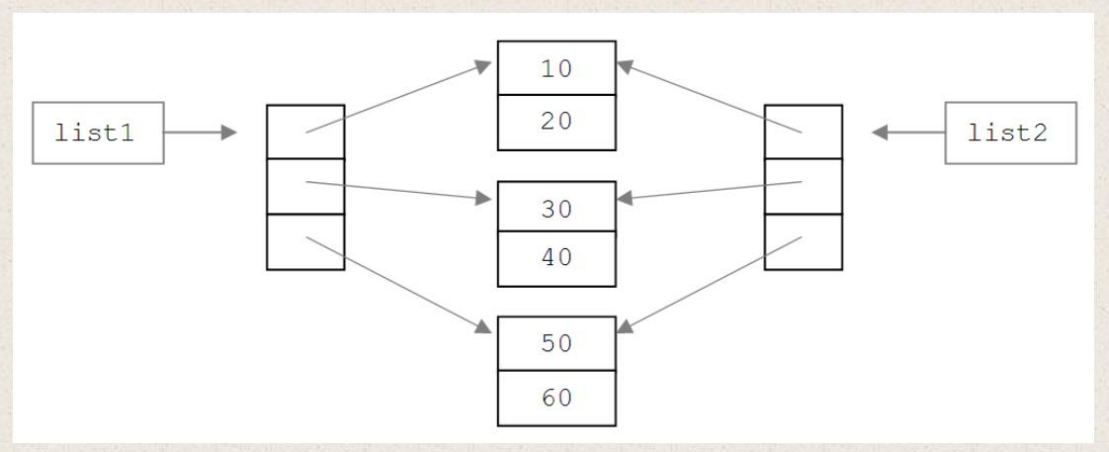


Fig 14.5 List assignment and copy in a different way

14.3 TURTLE GRAPHICS

Turtle Graphics is one of the most engaging and visual ways to learn programming concepts and understand how objects work in Python.

It provides a graphics environment in which a *"turtle"* moves on the screen under program control, leaving a trail as it goes — much like a pen drawing lines on paper.T he turtle can

move forward, turn, change color, and even draw shapes, all by calling its methods. This system allows beginners to visualize program execution and directly see how object-oriented commands affect an object's state.

Concept of the Turtle Object

In Python, the turtle module provides a built-in Turtle class.

When we create a new turtle using:

```
import turtle
t = turtle.Turtle()
```

we are instantiating an object from the Turtle class.

This t object has attributes (such as position, direction, color, and pen state) and methods (like forward(), left(), and circle()).

Each turtle object operates independently, allowing you to create multiple turtles on the same screen.

Advantages of Using Turtle Graphics

1. Provides an intuitive, visual approach to understanding programming logic.
2. Encourages experimentation and creativity.
3. Demonstrates object behavior (state, methods, and encapsulation).
4. Useful for teaching loops, conditionals, and functions through graphical tasks.
5. Allows multiple objects (turtles) to illustrate interactions and concurrency.

Turtle Graphics in Python:

- Uses objects and methods to represent motion and drawing.
- Makes abstract programming concepts visual and interactive.
- Provides a practical introduction to object-oriented design through creativity and play.
- It bridges the gap between logic and visualization, making it an ideal educational tool for new programmers.

14.3.1 Creating a Turtle Graphics Window

Turtle graphics is a fun way to introduce programming and OOP through graphics. It uses a "turtle" that moves around a screen, drawing lines as it goes.

```
import turtle
turtle.setup(800, 600)
window = turtle.Screen()
window.title("My Turtle Window")
```

A turtle screen of 800×600 pixels is created, titled *"My Turtle Window"*.

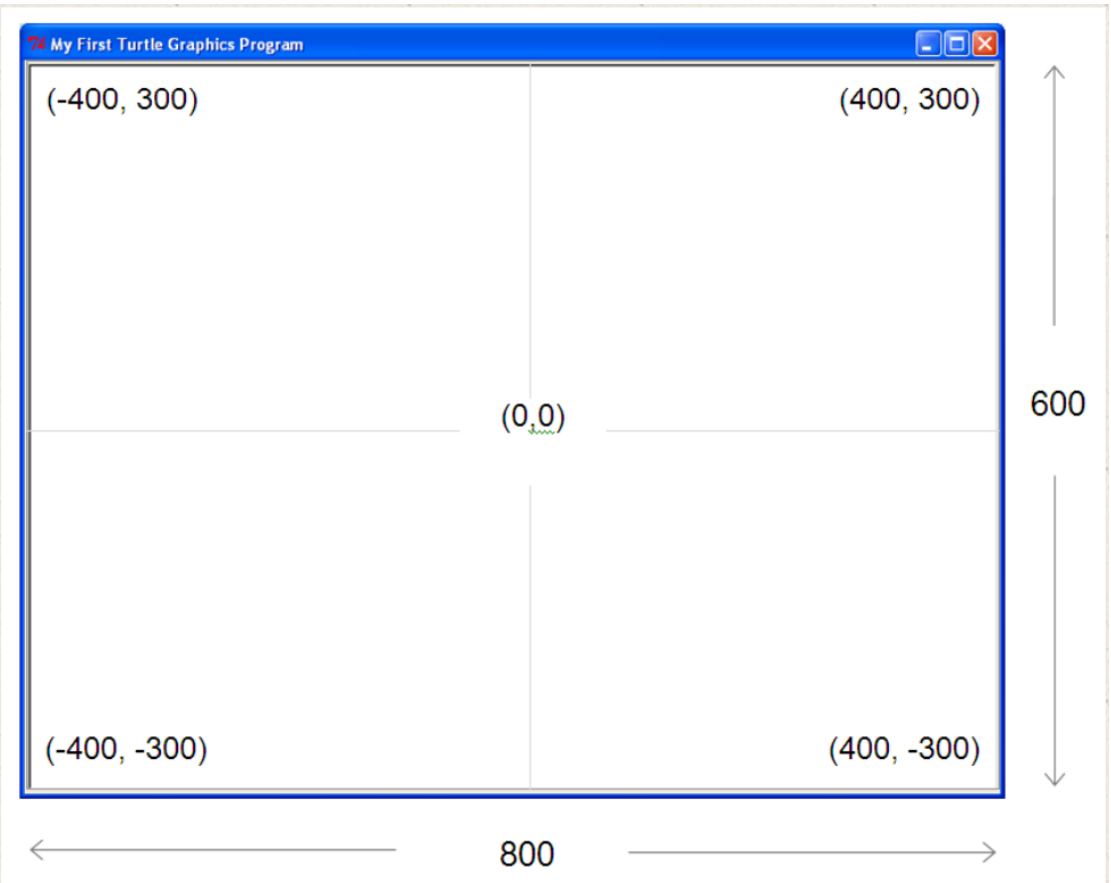


Fig 14.6 My Turtle Window

14.3.2 Turtle Position and Movement

In the turtle graphics system, every turtle object has a position and a heading (direction) that determine where it is on the screen and which way it is facing. The position is represented by x and y coordinates within the graphics window, where the center of the window is coordinate (0, 0). The turtle moves relative to its current position using methods such as forward(distance) and backward(distance), which move it along its heading, and left(angle) and right(angle), which rotate the turtle by the specified number of degrees. The movement is continuous, and if the turtle's pen is down (the default state), it draws a visible line as it moves.

Absolute positioning can also be achieved using the goto(x, y) method, which moves the turtle directly to a specific location on the screen. The methods setx(x) and sety(y) move the turtle horizontally or vertically without changing its other coordinate. The home() method returns the turtle to the center (0, 0) with its heading facing east. By combining movement and rotation commands within loops, complex geometric figures such as polygons and spirals can be easily drawn.

Thus, turtle movement illustrates fundamental object behavior — the object (turtle) maintains an internal state (position and heading) and responds to method calls that modify that state, making the concept of object interaction both visible and intuitive.

A turtle's position is defined by (x, y) coordinates.

t = turtle.getturtle()

t.setposition(100, 100)

The turtle moves to position (100,100), drawing a line if its pen is down.

Relative movement is done using methods such as:

t.forward(100)

t.left(90)

This allows shapes such as squares to be drawn.**Example – Drawing a Square**

```
for i in range(4):
    t.forward(100)
    t.left(90)
```

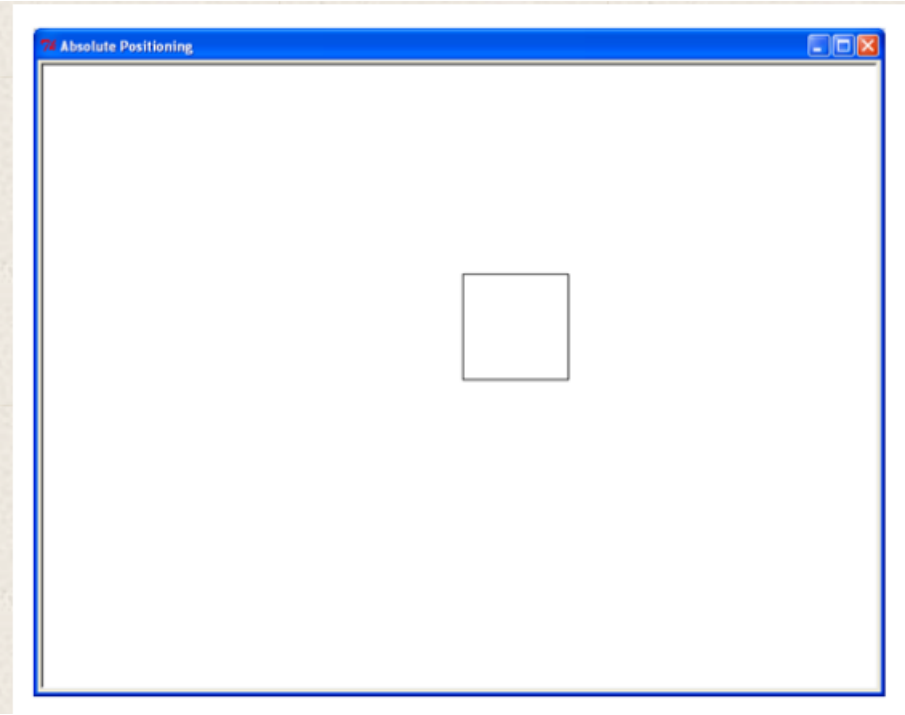


Fig 14.7 Example – Drawing a Square

14.3.3 Pen Attributes and Colors

Every turtle in the Python turtle module carries a pen, which controls how and when lines are drawn as the turtle moves. The pen's properties — such as color, thickness, and drawing state — can be customized through various methods that make turtle drawings more expressive and visually distinct. By default, the pen is down, meaning the turtle draws a line whenever it moves. The method penup() lifts the pen, allowing the turtle to move without drawing, while pendown() lowers it again to resume drawing.

The color of the pen can be changed using pencolor(), which accepts either a color name (e.g., "red", "blue") or an RGB color value (e.g., (0.5, 0.2, 0.8)). To modify the thickness of the line, the pensize() or width() method is used. For example, t.pencolor("green") and t.pensize(4) set

the pen to draw thick green lines. The fill color used to shade shapes can be controlled with fillcolor() and activated using begin_fill() and end_fill().

These attributes enable the creation of colorful and detailed designs, making programs both interactive and visually engaging. Managing pen attributes reinforces the concept of object state in object-oriented programming — the turtle object "remembers" its current pen color, width, and state, and every drawing action reflects these properties.

- Pen up / down:
  penup() and pendown() toggle drawing.
- Line width:
  pensize(5) sets the line width in pixels.
- Color:
  pencolor('blue') or pencolor(255, 0, 0) (if using RGB mode).

  t.pencolor('green')

  t.pensize(4)

  t.forward(120)

Example – Drawing a Colored Triangle with Pen Attributes

The following example demonstrates how pen color, fill color, and line width can be controlled to create an attractive filled triangle using the turtle graphics module.

```
import turtle
t = turtle.Turtle()
t.pensize(4)            # Set line thickness
t.pencolor("blue")      # Set outline color
t.fillcolor("yellow")   # Set fill color
t.begin_fill()          # Start filling the shape
for i in range(3):      # Draw an equilateral triangle
   t.forward(150)
   t.left(120)
t.end_fill()            # Complete the fill
t.hideturtle()
turtle.done()
```

Output Description:

A blue-bordered triangle filled with yellow color is drawn at the center of the screen. The thick border is a result of setting the pen size to 4 pixels.

This simple example illustrates how pen attributes affect both the appearance and quality of graphical output, while reinforcing the object-oriented nature of the turtle — every visual change is a result of sending commands (messages) to the turtle object to modify its internal drawing state.

14.3.4 Shapes, Sizes, and Speed

- Shape: 'arrow', 'turtle', 'circle', 'square', 'triangle', 'classic'
- Resize: t.resizemode('user') and t.turtlesize(3,3) enlarge the turtle.
- Speed: t.speed(6) controls animation; t.hideturtle() speeds drawing.

Creating Custom Shapes

points = ((5,5), (10,0), (5,-5), (0,0))

turtle.register_shape('mypolygon', points)

t.shape('mypolygon')

14.3.5 Multiple Turtles and Animation

You can create multiple turtles using:

t1 = turtle.Turtle()

t2 = turtle.Turtle()

Each turtle can move independently, creating animations such as bouncing balls or horse races.

14.4 CASE STUDY – HORSE RACE SIMULATION

This case study illustrates how objects, modules, and randomness combine to simulate a real-world system.

The Problem

Simulate a horse race where each horse (turtle) moves forward a random distance until one reaches the finish line.

Program Modules Used

- turtle – for graphics visualization
- random – for random movement
- time – to control simulation speed

Algorithm Overview

1. Create a turtle window.
2. Register horse images.
3. Position 10 horses at the starting line.
4. Move each horse forward by a random amount.
5. Detect when a horse crosses the finish line.

6. Display the winner.

Program Snippet

```
        import turtle, random, time
def createHorse(x, y, color):
    h = turtle.Turtle()
    h.shape('turtle')
    h.color(color)
    h.penup()
    h.setposition(x, y)
    h.pendown()
    return h


def startRace(horses):
    finish = 300
    while True:
        for h in horses:
            h.forward(random.randint(1, 5))
            if h.xcor() >= finish:
                print(h.pencolor(), "wins!")
                return


screen = turtle.Screen()
colors = ['red', 'blue', 'green', 'orange', 'purple']
horses = [createHorse(-300, i * 50, colors[i]) for i in range(5)]
startRace(horses)
screen.exitonclick()
```

Output:

Turtles race across the screen, and the color of the winning turtle is printed in the console.

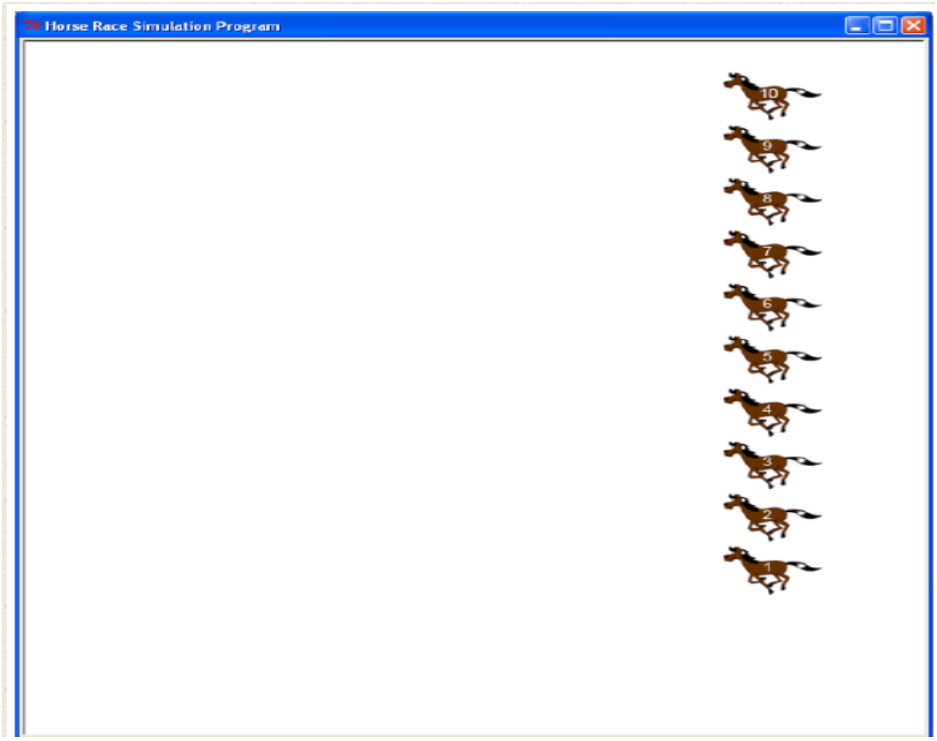


Fig 14.7 Example – Horse Race Simulation

14.5 MODULAR DESIGN

14.5.1 Modules and Top-Down Design

Top-Down Design breaks a large problem into smaller, manageable modules.

Each module handles a single task, making programs easier to write, debug, and reuse.

Example Breakdown – Horse Race Program

Module	Functionality
graphics	Create screen and horses
race	Move horses and determine winner
main	Combine everything and run

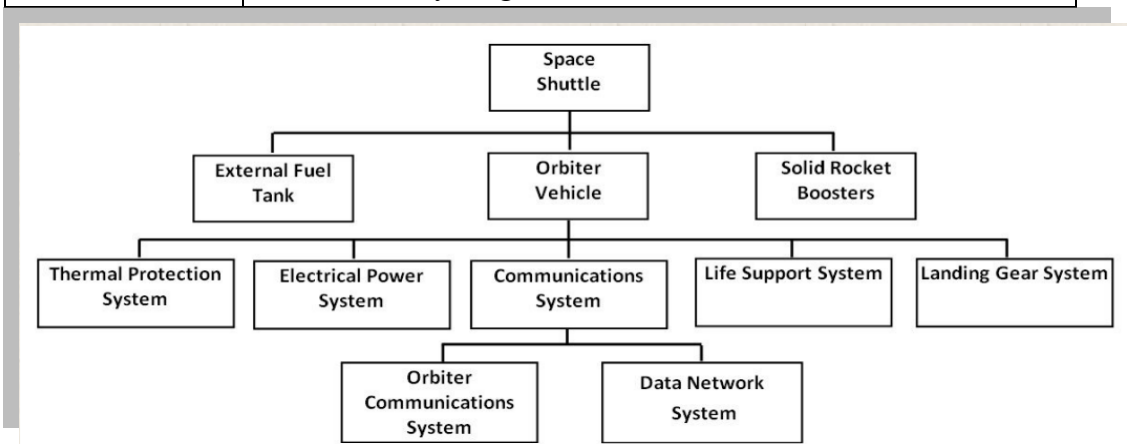


Fig 14.8 Modular Design of the NASA Space Shuttle

Pytbon Code:

```
# main.py
from power import ElectricalSystem
from control import CommunicationSystem

def main():
    print("Vehicle Control Simulation Starting...")
    power = ElectricalSystem()
    comms = CommunicationSystem()
    power.activate()
    comms.initialize()
    print("System Operational.")

if __name__ == "__main__":
    main()

# power.py
class ElectricalSystem:
    def activate(self):
        print("Electrical System Activated.")

# control.py
class CommunicationSystem:
    def initialize(self):
        print("Communication System Initialized.")
```

Output:

```
Vehicle Control Simulation Starting...
Electrical System Activated.
Communication System Initialized.
System Operational.
```

software systems should also be modularly structured—each module representing a manageable part of the overall design.

This modular approach is fundamental to object-oriented programming, where each class and module models a *real-world component* with clearly defined attributes and behaviors.

Modular design allows large programs to be broken down into manageable size parts, in which each part (module) provides a clearly specified capability. It aids the software development process by providing an effective way of separating programming tasks among various individuals or teams. It allows modules to be individually developed and tested, and eventually integrated as a part of a complete system. Finally, modular design facilitates program modification since the code responsible for a given aspect of the software is localized in a small number of modules, and not distributed through various parts of the program.

14.5.2 Python Modules and Importing

Python allows reusing code via modules.A module is simply a Python file (.py) containing reusable functions or classes.

Creating a Module

```
# file: math_utils.py
def square(x):
    return x * x
```

Using the Module

```
import math_utils
print(math_utils.square(4))
```

Output

16

Selective Importing :

```
from math_utils import square
print(square(5))
```

14.6 SUMMARY

- Objects combine data and methods into a single entity.
- Variables store references to objects, not the objects themselves.
- Garbage collection reclaims unused memory automatically.
- The turtle module provides a visual introduction to object behavior.
- Modules and top-down design promote reusable, structured programming.

14.7 TECHNICAL TERMS

- Object
- Reference
- Garbage Collection
- Turtle Graphics
- Module

14.8 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain the concept of software objects in Python.
2. Describe how garbage collection works.
3. Write a program using turtle graphics to draw a star.
4. Explain the importance of modular programming in Python.

Short Notes

1. Turtle attributes and shapes
2. Object references and id() function
3. Difference between shallow and deep copy
4. Advantages of using modules

14.9 SUGGESTED READINGS

1. Dierbach, *Introduction to Computer Science Using Python*, Wiley, 2013.
2. Ljubomir Perković, *Introduction to Computing Using Python*, Wiley, 2012.
3. Alan Kay, *The Early History of Smalltalk*, ACM, 1993.
4. Python Software Foundation, *turtle — Turtle Graphics Documentation*, https://docs.python.org

Dr. Vasantha Rudramalla

LESSON- 15

RECURSION

AIMS AND OBJECTIVES

After completing this chapter, the learner will be able to:

- Understand the concept and working of recursion in problem-solving.
- Write recursive functions in Python to solve mathematical and algorithmic problems.
- Compare recursion and iteration in terms of logic and performance.
- Perform runtime analysis of recursive algorithms.
- Apply recursion to solve searching, mathematical, and divide-and-conquer problems.
- Recognize the role of functional programming concepts in recursive design.

STRUCTURE

15.1 INTRODUCTION TO RECURSION

Recursion is a programming technique in which a function calls itself directly or indirectly to solve a smaller version of the original problem.

In a recursive process, each call solves a simpler subproblem, and the recursion continues until a base case is reached — a condition where the problem can be solved directly without further recursive calls.

Formally, recursion divides a problem into:

1. Base case – A stopping condition that prevents infinite recursion.
2. Recursive case – The part where the function calls itself to solve a smaller problem.

Example – Simple Recursive Function

```
def countdown(n):
    if n == 0:
        print("Blast off!")
    else:
        print(n)
        countdown(n - 1)
```

Output:

```
5
4
3
2
1
Blast off!
```

Explanation:

- Each recursive call reduces the problem size by one.
- When n becomes 0, the **base case** is reached, and recursion stops.

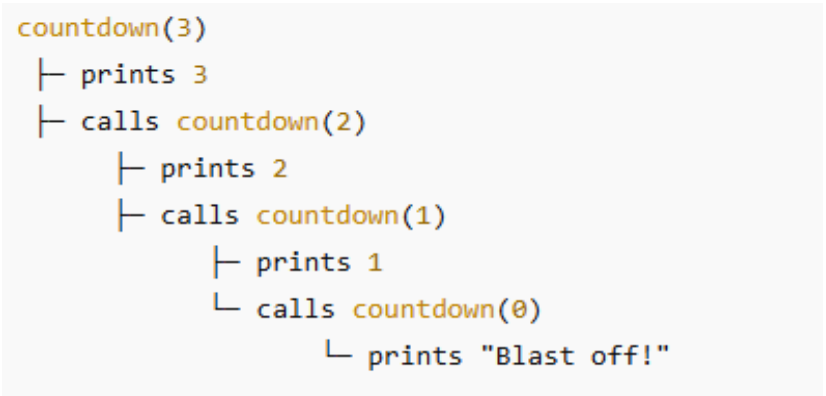


Fig 15.1 Conceptual Visualization – Recursion

Each recursive call is added to the **call stack**, and execution resumes backward once the base case is reached.

How does this code implement the function countdown() for input value $n > 0$? The insight used in the code is this: *Counting down from (positive number) n can be done by printing n first and then counting down from n* 1. This fragment of code is called *the recursive step*. With the two cases resolved, we obtain the recursive function:

```
def countdown(n):
        'counts down to 0'
        if n <= 0: # base case
                print('Blastoff!!!')
        else: # n > 0: recursive step
                print(n) # print n first and then
                countdown(n-1) # count down from n-1
```

A recursive function that terminates will always have:

1. One or more base cases, which provide the stopping condition for the recursion. In function countdown(), the base case is the condition $n \leq 0$, where n is the input.
2. One or more recursive calls, which must be on arguments that are "closer" to the base case than the function input. In function countdown(), the sole recursive call is made on n 1, which is "closer" to the base case than input n.

```
def cheers(n):
   """Prints 'Hip ' n times followed by 'Hurray!!!' using recursion."""
   if n <= 0:              # base case
      print("Hurray!!!")
   else:
      print("Hip ", end='')  # print prefix without newline
      cheers(n - 1)        # recursive call
```

How it works

- **Base case:** when n <= 0 the function prints the final word Hurray!!! and stops.
- **Recursive case:** when n > 0 it prints the prefix Hip (note the trailing space) and recursively calls cheers(n-1).

The printed prefixes accumulate (left-to-right) because each call prints one Hip before delegating the remainder.

Examples (interactive)

```
>>> cheers(0)
Hurray!!!
>>> cheers(1)
Hip Hurray!!!
>>> cheers(4)
Hip Hip Hip Hip Hurray!!!
```

Complexity

- **Time:** O(n) — one recursive call per Hip printed.
- **Space:** O(n) call-stack depth (recursion frames).
- If you prefer to **return** the string instead of printing, you can implement a version that builds and returns the string (useful for testing).
- If you expect negative inputs and want them handled differently, replace the if n <= 0: guard with if n == 0: and raise an error for n < 0.

Recursive Function Calls and the Program Stack:

Printing Digits Vertically Using Recursion

```
def vertical(n):
    """Prints the digits of n vertically."""
    if n < 10:                # base case: single-digit number
        print(n)
    else:                     # recursive case
        vertical(n // 10)     # print all but the last digit
        print(n % 10)         # print the last digit
```

Explanation

The function vertical(n) prints each digit of the integer n on a separate line, from **most significant digit** to **least significant digit**.

It uses recursion to repeatedly **reduce** the number by removing its last digit until only one digit remains — the **base case**.

- **Base Case:**
  If n is a single-digit number (n < 10), simply print it.
- **Recursive Case:**
  If n has two or more digits:

1. Call vertical(n // 10) — this discards the last digit and recursively prints the remaining digits.
2. After returning from recursion, print the last digit using print(n % 10).

Example Execution

>>> vertical(348)

Output:

3

4

8

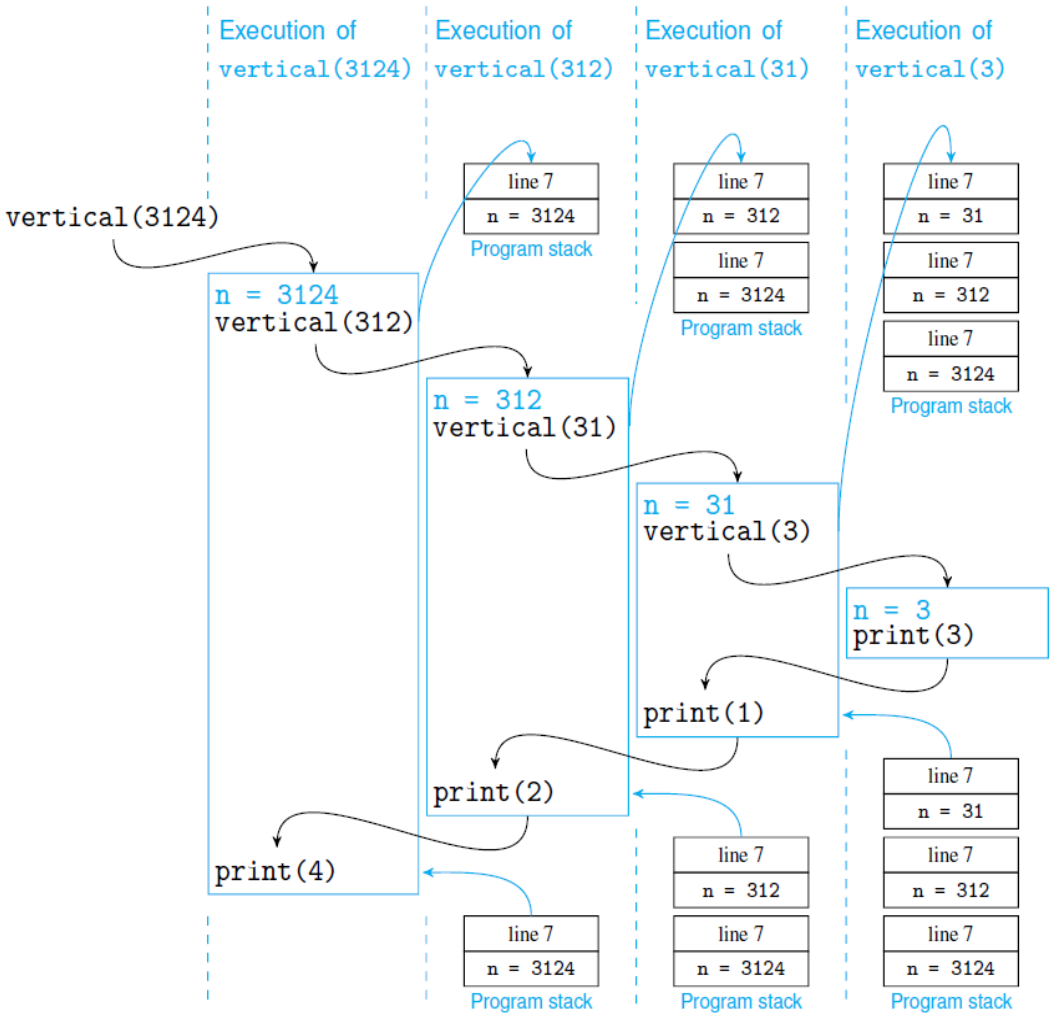


Figure 15.2 Recursive function execution.

15.2 EXAMPLES OF RECURSION

Recursion is best understood through simple, familiar problems that can naturally be defined in terms of smaller versions of themselves. Classic examples include the computation of a **factorial**, the **sum of natural numbers**, the **Fibonacci sequence**, and **countdown functions**.

Each of these problems follows a common recursive structure: a **base case** that directly provides an answer and a **recursive case** that reduces the problem toward that base case. For example, the factorial function can be expressed as n! = n × (n−1)!, where the base case is 0! = 1. Similarly, the Fibonacci series is defined as F(n) = F(n−1) + F(n−2), where the sequence builds upon previously computed results. Recursive functions like countdown(n) or sum_n(n) repeatedly call themselves with a smaller input until the simplest instance of the problem is reached.

These examples illustrate the **self-referential nature** of recursion—each function call handles part of the work and delegates the rest to a smaller, identical subproblem. Through this process, recursion converts complex problems into simpler ones, demonstrating how powerful and elegant recursive thinking can be when applied to mathematical and algorithmic problem-solving.

15.2.1 Factorial Function

The **factorial** of a non-negative integer n is defined as:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

and by definition, 0! = 1.

Recursive Implementation

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Example:

```
print(factorial(5))
```

Output:

```
120
```

Explanation:

Each recursive call computes n * factorial(n-1) until the base case n==0 is reached.

15.2.2 Sum of Natural Numbers

Recursive definition:

$$sum(n) = n + sum(n-1), sum(0) = 0$$

```
def sum_n(n):
    if n == 0:
        return 0
```

```
    else:
        return n + sum_n(n - 1)
```

Example:

sum_n(5) = 5 + 4 + 3 + 2 + 1 = 15

15.2.3 Fibonacci Sequence

The **Fibonacci sequence** is defined recursively as:

$$F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1$$

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Output:

fibonacci(6) → 8

Increasing and Decreasing Sequence

The following function recursively prints a sequence of numbers from 1 up to n, and then back down to 1.

```
def pattern(n):
    """Prints a recursive number pattern 1..n..1"""
    if n == 0:
        return
    print(n, end=' ')      # First part: descending
    pattern(n - 1)         # Recursive call
    print(n, end=' ')      # Second part: ascending
```

Example Execution

```
>>> pattern(4)
```

Output:

4 3 2 1 1 2 3 4

```
pattern(4)
 ├ prints 4
 ├ calls pattern(3)
      ├ prints 3
      ├ calls pattern(2)
           ├ prints 2
           ├ calls pattern(1)
                ├ prints 1
                ├ calls pattern(0)
                ├ returns, prints 1
           ├ returns, prints 2
      ├ returns, prints 3
 ├ returns, prints 4
```

Figure 15.3 Symmetric Recursive Structure

Fractals:

In our next example of recursion, we will also print a pattern, but this time it will be a graphical pattern drawn by a Turtle graphics object. For every nonnegative integer *n*, the printed pattern will be a curve called the *Koch curve Kn*. For example, Figure 10.4 shows Koch curve *K*5.

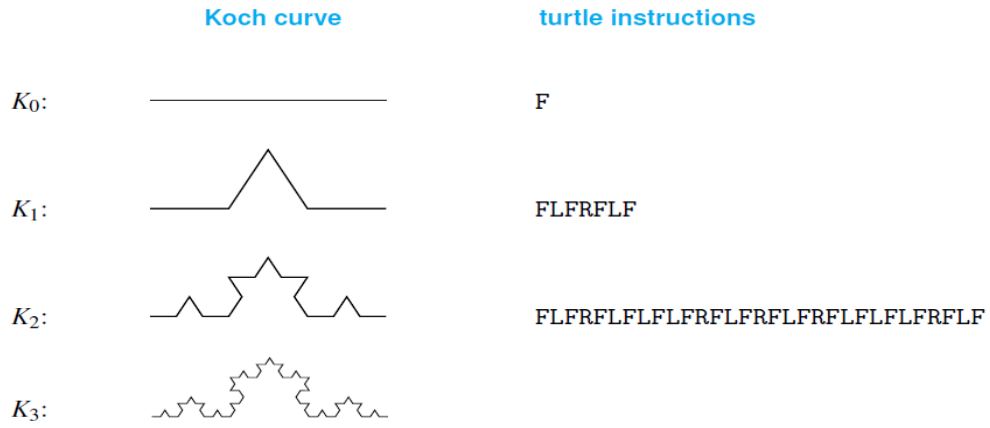


Figure 15.4 Koch curves with drawing instructions

Generating and Drawing the Koch Curve

```
from turtle import Screen, Turtle
def koch(n):
    """Returns turtle directions for drawing the nth Koch curve."""
    if n == 0:                  # Base case
        return 'F'              # 'F' means move forward
    tmp = koch(n - 1)           # Recursive case: build Koch(n-1)
```

```
    return tmp + 'L' + tmp + 'R' + tmp + 'L' + tmp


def drawKoch(n):
    """Draws the nth Koch curve using turtle graphics."""
    s = Screen()              # Create drawing window
    t = Turtle()              # Create turtle
    t.speed(0)                # Set fastest drawing speed
    t.penup()
    t.goto(-150, 0)           # Position turtle for drawing
    t.pendown()
    directions = koch(n)      # Obtain recursive directions
    for move in directions:   # Interpret each command
        if move == 'F':
            t.forward(300 / (3 ** n))  # Move forward, scaled to recursion level
        elif move == 'L':
            t.left(60)            # Turn left 60 degrees
        elif move == 'R':
            t.right(120)          # Turn right 120 degrees
    s.mainloop()                  # Keep window open
# Example: draw Koch curve of level 3
drawKoch(3)
```

15.3 RUN TIME ANALYSIS OF RECURSIVE FUNCTIONS

Recursive functions can be analyzed in terms of **time complexity** and **space complexity**.

- **Factorial function**: O(n) – One recursive call per level.
- **Fibonacci function**: O(2^n) – Exponential growth due to repeated subproblems.
- **Binary search**: O(log n) – Divides problem size by 2 each step.

Each recursive call adds a new **activation record** to the **call stack**, consuming additional memory.

Thus, recursion provides elegant solutions but can become inefficient without optimization (e.g., **memoization**).

Koch curves Run Time Analysis

- Each level of recursion produces **4^n segments**.

- Hence, **Time Complexity = $O(4^n)$**
- **Space Complexity = O(n)** (for recursion depth).

15.4 RECURSIVE SEARCHING

Recursion is widely used in **search algorithms** such as linear and binary search.

15.4.1 Linear Search (Recursive)

Linear Search is the simplest searching algorithm that sequentially checks each element of a list until the target value is found or the end of the list is reached.

In its **recursive form**, the function checks one element per recursive call, reducing the problem size by one at each step — just like iterative looping but using the **call stack** instead of explicit loop control.
The recursive linear search function works as follows:

1. **Base Case:**

   If the list is empty or the search index has reached the end, the element is not found — return -1.

2. **Recursive Case:**

   Compare the target element with the current list element.

   - If they match, return the current index.
   - Otherwise, make a recursive call on the rest of the list (or increment the index).

```
def linear_search(lst, key, index=0):
    if index == len(lst):
        return -1
    elif lst[index] == key:
        return index
    else:
        return linear_search(lst, key, index + 1)
```

Example:

linear_search([5, 3, 8, 6], 8) → 2

Complexity: O(n)

15.4.2 Binary Search (Recursive)

Binary Search is an efficient algorithm for finding a target value within a **sorted list**.

Unlike linear search, which checks each element sequentially, binary search **divides the search**

space in half with each step.

It compares the target element to the **middle element** of the list:

- If the target equals the middle element, the search is successful.
- If the target is smaller, the search continues recursively in the **left half**.
- If the target is larger, it continues recursively in the **right half**.

This "divide-and-conquer" approach drastically reduces the number of comparisons, making binary search one of the most efficient search algorithms.

Recursive Definition

Binary search naturally lends itself to a **recursive solution**, since each recursive call works on a smaller (half-sized) portion of the list.

1. **Base Case:**

   If the list portion to search is empty (low > high), return -1 (element not found).

2. **Recursive Case:**

   - Compute the **middle index**:

$$mid = \frac{low + high}{2}$$

   - Compare the target with list[mid].
     - If equal → return mid.
     - If smaller → recursively search the **left half**.
     - If larger → recursively search the **right half**.

Binary search applies only to **sorted lists**.

```
def binary_search(lst, key, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    if lst[mid] == key:
        return mid
    elif key < lst[mid]:
        return binary_search(lst, key, low, mid - 1)
    else:
        return binary_search(lst, key, mid + 1, high)
```

Example:

binary_search([1,3,5,7,9,11], 7, 0, 5) → 3

Complexity: O(log n)

15.5 ITERATION VS RECURSION

Aspect	Iteration	Recursion
Definition	Repeats statements using loops (for, while).	Function calls itself with smaller subproblems.
Control Mechanism	Loop control variable	Function call stack
Base/End Condition	Loop termination condition	Base case
Memory Use	Constant	Increases with recursion depth
Speed	Faster (less overhead)	Slower (function calls add overhead)
Elegance	Less abstract, sometimes verbose	Elegant and mathematically natural
Example	Loops	Factorial, Fibonacci

Example comparison for factorial:

```
# Iterative
def fact_iter(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
# Recursive
def fact_rec(n):
    if n == 0:
        return 1
    else:
        return n * fact_rec(n-1)
```

Both yield the same output, but recursion provides clearer logical structure for divide-and-conquer problems.

Feature	Linear Search	Binary Search
Data requirement	Works on unsorted lists	Requires sorted list
Approach	Sequential	Divide and conquer
Complexity (Time)	O(n)	O(log n)
Complexity (Space)	O(1) iterative / O(n) recursive	O(log n) recursive
Example Use	Small or unsorted datasets	Large sorted datasets

Recursive binary search demonstrates the power of recursion to simplify complex logic — instead of using multiple loop conditions, it expresses the solution as repeated self-calls on progressively smaller problems.It is a cornerstone example of how recursion can combine mathematical elegance with computational efficiency.

15.6 RECURSIVE PROBLEM SOLVING

Recursive problem solving is a method of approaching complex problems by breaking them down into **smaller, similar subproblems** that can be solved using the same technique. In this approach, a function or algorithm calls itself with a smaller input until it reaches a **base case** — a simple condition that can be solved directly without further recursion. Once the base case is reached, the function's intermediate results are combined as the recursion "unwinds," producing the final solution.

Recursion mirrors the **divide-and-conquer** strategy: divide the problem into manageable parts, solve each recursively, and combine the results. This technique is especially powerful for problems defined in terms of smaller versions of themselves — such as computing factorials, generating Fibonacci numbers, traversing tree structures, solving the Towers of Hanoi puzzle, and performing binary search.

Recursive problem solving encourages **top-down thinking** — focusing first on defining the overall structure of the solution, then letting recursion handle the details of smaller computations automatically. It offers elegant, mathematically consistent solutions and is widely used in **algorithms, data structures, and graphics**. However, it must always include a well-defined **base case** to prevent infinite recursion and excessive memory usage.

15.6.1 Towers of Hanoi

The classic **Towers of Hanoi** puzzle demonstrates recursion elegantly.
The problem involves moving n disks from one peg to another, following these rules:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. Use a third peg as an auxiliary.

```
def hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
    else:
        hanoi(n-1, source, target, auxiliary)
        print(f"Move disk {n} from {source} to {target}")
        hanoi(n-1, auxiliary, source, target)
```

Example:

```
hanoi(3, 'A', 'B', 'C')
```

Output:

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
```

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

Complexity: $O(2^n - 1)$

15.6.2 Greatest Common Divisor (GCD)

The Euclidean algorithm is naturally recursive:

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Example:

gcd(48, 18) → 6

Complexity: O(log n)

15.7 FUNCTIONAL LANGUAGE APPROACH

Recursion forms the foundation of **functional programming**, where problems are solved by defining **functions in terms of themselves** rather than changing state or using loops.
In Python, recursion aligns with a **declarative approach** — describing *what* to do, not *how* to do it.

Key features:

- Functions are **pure** (no side effects).
- Emphasis on **mathematical definition**.
- No use of mutable variables.
- Enables **higher-order functions** such as map(), filter(), and reduce().

Example:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)
```

Here, recursion replaces iteration naturally, making the function concise and closer to its mathematical definition.

Concept of Map–Reduce

The **Map–Reduce** model divides computation into two primary phases:

1. **Map Phase:**

   A function is **applied independently** to each element in a collection (list, tuple, etc.).

   The result is a new list containing the function's output for each input element.

   This corresponds to **mapping** a function over data — similar to recursion over lists where each recursive call processes one element.

2. **Reduce Phase:**

   The results of the map phase are **combined** into a single cumulative value using a **reducer function**.

   This process recursively collapses multiple results into one — for example, summing a list of numbers or concatenating strings.

Sequential Map–Reduce Example (Word Counting)

The **Map–Reduce framework** processes data in two main phases — **map** and **reduce** — following the functional programming and recursive problem-solving approach.

In this example:

```
>>> words = ['two', 'three', 'one', 'three', 'three', 'five', 'one', 'five']
>>> smr = SeqMapReduce(occurrence, occurrenceCount)
>>> smr.process(words)
[('one', 2), ('five', 2), ('two', 1), ('three', 3)]
```

1. **Map Phase:**

   Each word is mapped into a key–value pair — (word, 1) — representing one occurrence.

   Example: ('three', 1)

2. **Group Phase:**

   Intermediate pairs are grouped by key:

   {'one': [1,1], 'three': [1,1,1], 'five': [1,1], 'two': [1]}

3. **Reduce Phase:**

   The reducer function sums each list of values to count occurrences.

   Final result:

   [('one', 2), ('five', 2), ('two', 1), ('three', 3)]

This example demonstrates the **recursive nature of Map–Reduce** — the map phase applies a function to each element independently (like recursive traversal), and the reduce phase combines results cumulatively (like recursive aggregation).

Thus, Map–Reduce models **functional recursion** — dividing, processing, and recombining data efficiently.

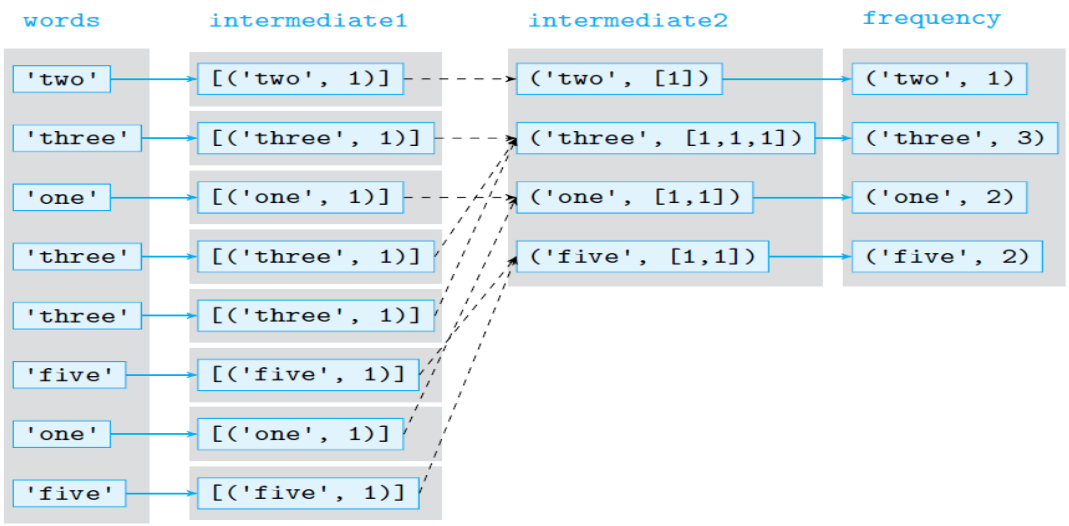


Figure 15.5 MapReduce for word frequency

```
class SeqMapReduce(object):
    """A sequential Map–Reduce implementation."""
        def __init__(self, mapper, reducer):
        """Functions mapper and reducer are problem-specific."""
        self.mapper = mapper
        self.reducer = reducer
    def process(self, data):
        """Runs Map–Reduce on data using the provided mapper and reducer."""
        intermediate1 = [self.mapper(x) for x in data]      # Map Phase
        intermediate2 = partition(intermediate1)            # Group Phase
        return [self.reducer(x) for x in intermediate2]     # Reduce Phase
```

15.8 SUMMARY

- Recursion solves problems by **dividing them into smaller, similar subproblems**.
- Every recursive function must include a **base case** to stop the recursion.
- Recursive solutions can be more elegant but less efficient than iterative ones.
- Recursive algorithms are used in **searching**, **sorting**, and **divide-and-conquer** methods.
- Functional programming embraces recursion as a natural expression of computation.

15.9 TECHNICAL TERMS

- Term
- Recursion
- Base Case
- Call Stack
- Tail Recursion
- Memoization

15.10 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Define recursion. Explain how it differs from iteration with examples.
2. Discuss runtime analysis of recursive algorithms.
3. Explain how recursion is used in searching algorithms.
4. Write and explain the recursive solution to the Towers of Hanoi problem.

Short Notes

1. Base case and recursive case
2. Functional programming and recursion
3. Binary search using recursion
4. Tail recursion

15.11 SUGGESTED READINGS

1. Ljubomir Perković, *Introduction to Computing Using Python*, Wiley, 2012.
2. Dierbach, *Introduction to Computer Science Using Python*, Wiley, 2013.
3. Allen Downey, *Think Python: How to Think Like a Computer Scientist*, O'Reilly, 2015.
4. Python Software Foundation, *Recursion and Functional Programming*, https://docs.python.org.

Dr. Vasantha Rudramalla

LESSON- 16
NAMESPACES

AIMS AND OBJECTIVES

The aim of this lesson is to explain how Python manages names, variables, and their visibility through namespaces, and how encapsulation supports modular design, error handling, and code reuse.

After completing this lesson, the learner will be able to:

- Understand the concept and purpose of namespaces and scope.
- Distinguish between local and global variables.
- Explain how encapsulation promotes modularity and information hiding.
- Describe how functions, modules, and classes define their own namespaces.
- Understand exceptional control flow and handle errors using try–except blocks.
- Apply namespace and exception management techniques in structured Python programs.

STRUCTURE

16.1 INTRODUCTION

Every Python program consists of a collection of names (variables, functions, lasses, etc.) that refer to objects in memory.The association between a name and its corresponding object is stored in a structure called a namespace.

Namespaces are essential for:

- Organizing variables and preventing naming conflicts,
- Enabling modular programming and encapsulation, and
- Managing variable visibility (local vs global scope).
- This chapter introduces the concept of namespaces, showing how they relate to functions, modules, exceptions, and classes, and how they enforce information hiding and code organization.

16.2 ENCAPSULATION IN FUNCTIONS

16.2.1 Code Reuse

Encapsulation allows a programmer to divide a large program into smaller, independent parts.By encapsulating logic inside functions, code becomes reusable, manageable, and less error-prone.

For instance:

```
def area_circle(radius):
    """Returns the area of a circle."""
    return 3.14159 * radius ** 2

print(area_circle(5))
```

This function can be reused anywhere, without redefining its logic — an example of code reuse through encapsulation.
In the example below, two functions are defined:

- jump() — performs a **single well-defined action**: move the turtle without drawing.
- emoticon() — uses jump() and other turtle commands to draw a **complete smiley face**.

This structure illustrates how **encapsulation** enables clarity, reusability, and abstraction in Python programs.

Program: Drawing a Smiley Face Using Encapsulation

```
def jump(t, x, y):
    """Makes turtle t jump to coordinates (x, y) without drawing."""
    t.penup()
    t.goto(x, y)
    t.pendown()
def emoticon(t, x, y):
    """Directs turtle t to draw a smiley face with chin at (x, y)."""
    t.pensize(3)
    t.setheading(0)

    # Draw head
    jump(t, x, y)
    t.circle(100)

    # Draw right eye
    jump(t, x + 35, y + 120)
    t.dot(25)

    # Draw left eye
    jump(t, x - 35, y + 120)
    t.dot(25)

    # Draw smile
    jump(t, x - 60.62, y + 65)
    t.setheading(-60)
    t.circle(70, 120)
```

Explanation of Encapsulation

1. **Encapsulated Helper Function (jump)**
    - Handles only one responsibility: repositioning the turtle without leaving a trace.
    - By defining it once, this logic can be reused anywhere, rather than repeating pen-up and pen-down commands in multiple places.

- This is an example of **procedural abstraction** — hiding the "how" behind a function name that describes the "what."

2. **Main Function (emoticon)**
   - Focuses on the higher-level concept of drawing a smiley face.
   - It **uses jump()** without needing to know its internal implementation.
   - This demonstrates **encapsulation and modular design** — one function calls another to achieve its goal.

Program Output

When executed with:

```
import turtle
t = turtle.Turtle()
emoticon(t, 0, 0)
turtle.done()
```

The output is a **smiley face** drawn on the screen, with circular head, two eyes, and a curved smile.

16.2.2 Modularity (Procedural Decomposition)

Modularity means breaking a large problem into smaller, more manageable procedures. Each function handles a single responsibility, which collectively contributes to solving the overall problem.

For example:

```
def input_data():
    return int(input("Enter value: "))

def process_data(x):
    return x * 2

def display_result(result):
    print("Result:", result)
```

Each function has a clear boundary and can be modified independently — an essential property of modular design.

The function jump() is independent of the function emoticon() and can be tested and debugged independently. Once function jump() has been developed, the function emoticon() is easier to implement.

16.2.3 Encapsulation (Information Hiding)

Encapsulation also provides information hiding: details inside a function are hidden from the rest of the program.

This prevents unintended interference with internal variables.

```
def compute_sum():
    total = 0
```

```
    for i in range(5):
        total += i
    return total

print(compute_sum())
```

Here, the variable total exists only within the function and is not visible outside. This is achieved through local namespaces.

The developer of the function emoticon() does not need to know how function jump() works, just that it lifts turtle t and drops it at coordinates (*x, y*). This simplifies the process of developing function emoticon(). Another benefit of encapsulation is that if the implementation of function jump() changes (and is made more efficient, for example), the function emoticon() would not have to change.

16.2.4 Local Variables and Namespaces

When a function executes, Python creates a local namespace to store its variables. Each function call gets its own local namespace, which disappears after the function returns.

```
def example():
    x = 10  # local variable
    print(x)

example()
print(x)  # NameError: x is not defined
```

This ensures that variables inside a function are encapsulated and do not affect other parts of the program.

16.2.5 Namespaces Associated with Function Calls

When a function is called:

1. A new local namespace is created.
2. Function parameters and local variables are stored in it.
3. Python searches for variable names in the following order (LEGB Rule):
    - L: Local (inside the current function)
    - E: Enclosing (in nested functions)
    - G: Global (module-level)
    - B: Built-in (Python system functions)

```
x = 5
def outer():
    y = 10
    def inner():
        z = 15
        print(x, y, z)
    inner()
outer()
```

Output:

```
5 10 15
```

Here:

- z is local to inner().
- y is enclosing (outer function).
- x is global.

16.2.6 Namespaces and the Program Stack

Each function call adds a stack frame to the program stack, containing its local namespace. When the function ends, that frame is removed.

```
Program Stack
 ├── main() frame
 │     ├── global x = 5
 ├── outer() frame
 │     ├── local y = 10
 ├── inner() frame
       ├── local z = 15
```

Fig 16.1 Python manages these namespaces automatically using the call stack.

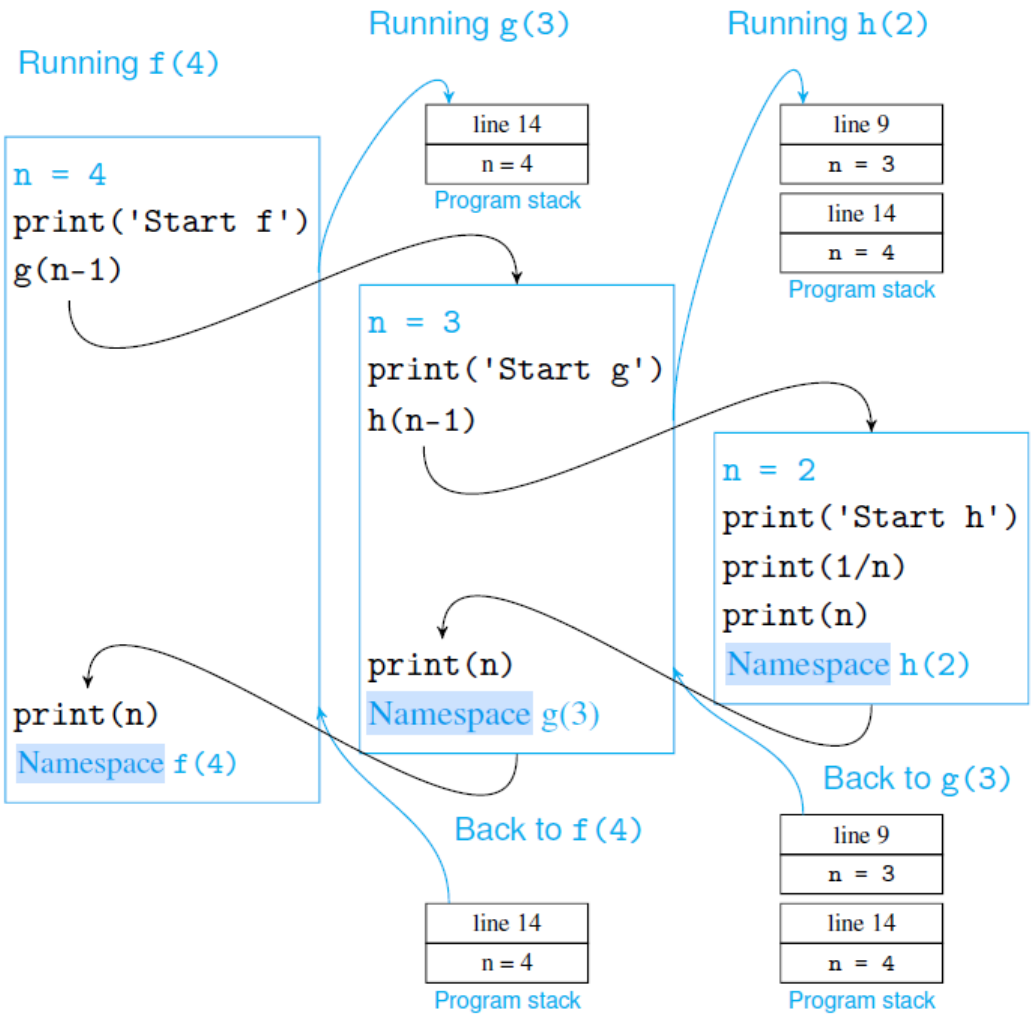


Figure 16.2 Execution of f(4).

The following code demonstrates how functions create their own namespaces and how Python manages function calls using the program stack.

Program Code

```
def h(n):
   print('Start h')
   print(1 / n)
   print(n)

def g(n):
   print('Start g')
   h(n - 1)
   print(n)

def f(n):
   print('Start f')
   g(n - 1)
   print(n)

f(3)
```

Program Output

```
Start f
Start g
Start h
0.5
2
2
3
```

- Every function call creates a new namespace (local environment).
- Python's program stack keeps track of nested calls.
- When a function ends, its namespace is destroyed and the stack unwinds.
- Functions can use the same variable names independently — they don't interfere with one another.
- The order of return is the reverse of the order of calls — last called, first returned (LIFO).
- **Namespaces** isolate each function's variables.
- **The program stack** manages execution order and variable lifetime.
- Each function's execution context is **independent** and **temporary**.
- The combination of both enables **modular, predictable, and error-free** function behavior.

16.3 GLOBAL VERSUS LOCAL NAMESPACES

Thus, names assigned inside functions belong to the local namespace, while names assigned outside functions belong to the global namespace. Python uses this separation to avoid naming conflicts and to support encapsulation and modularity in programs.

Scope Type	Where Defined	Visibility	Lifetime
Local	Inside a function	Within that function only	Created on function call, destroyed on return
Global	At the top level (module/shell)	Entire module	Exists for the duration of the program

16.3.1 Global Variables

A **global variable** is defined at the top level of a program or module.
It belongs to the **global namespace** and can be accessed by all functions within that module.

```
count = 0  # global variable
def increment():
    global count
    count += 1
increment()
print(count)  # Output: 1
```

Using global inside a function allows modification of global variables.

16.3.2 Local Scope

Variables defined inside a function exist only while that function is executing.

```
def func():
    local_var = "inside"
    print(local_var)
func()
# print(local_var) → Error: not defined
```

Local variables improve safety by preventing unexpected interference.

16.3.3 Global Scope

Global variables persist throughout the program.
However, excessive use of globals can cause **namespace pollution** and **unintended interactions** between functions.
Best practice: Use **function parameters and return values** rather than globals whenever possible.

16.3.4 Changing Global Variables Inside a Function

If a global variable must be changed inside a function, it must be explicitly declared as global.

Example:

```
total = 5
def add():
    global total
    total += 10

add()
print(total)  # Output: 15
```

Without the global keyword, Python treats total as a **local variable**, leading to an UnboundLocalError.

Example:

```
def f(b):
    global a       # all references to 'a' inside f() refer to the global variable 'a'
    a = 6          # modifies the global variable 'a'
    return a * b   # uses the global 'a'

a = 0              # this 'a' has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))
```

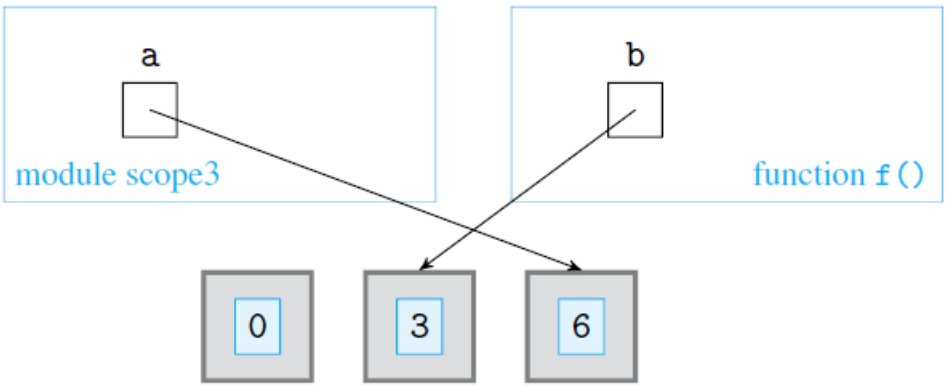


Figure 16.3 usage of Keyword global**.**

16.4 EXCEPTIONAL CONTROL FLOW

In Python, when an error occurs during program execution, an exception object is created. The term exception comes from the fact that an exceptional event has occurred — one that causes the program's normal flow of execution to be interrupted.

Normally, a program follows a predictable control flow, proceeding step by step according to the logic defined in its functions and loops. However, when an error arises — such as dividing by zero, accessing an invalid index, or opening a missing file — Python creates an exception object to represent this error condition.

Once the exception object is created, the regular (normal) control flow is suspended, and the program enters a separate path known as the exceptional control flow. This control flow is *not part of the usual sequence of operations* and typically isn't represented in the program's flowchart because it occurs only when an unexpected event happens.

If the exception is not handled by the programmer using a try–except statement, Python's default exceptional control flow takes over:

- The program stops execution immediately.
- The error message and stack trace associated with the exception object are printed to the screen.
  This default behavior helps identify where and why the error occurred, but it also terminates the program abruptly.

16.4.1 Introduction

Sometimes, unexpected events occur during program execution — such as dividing by zero, opening a missing file, or invalid user input.
These events cause **exceptions** that alter the program's normal control flow.

In Figure 16.4 , we illustrate what happens when we make the function call f(2) from the shell.The execution runs normally all the way to function call h(0). During the execution of h(0), the value of n is 0. Therefore, an error state occurs when the expression 1/n is evaluated. The interpreter raises a ZeroDivisionError exception and creates a ZeroDivisionError exception object that contains information about the error.

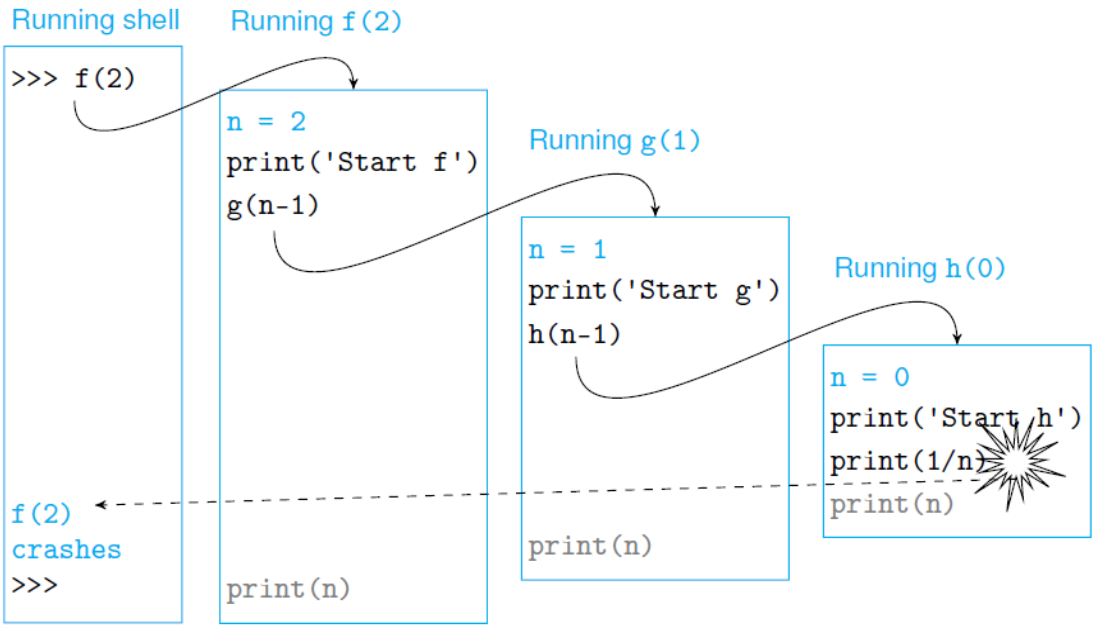


Figure 16.4 Execution of f(2)

16.4.2 Catching and Handling Exceptions

Python allows developers to **handle exceptions gracefully** using try and except blocks:

Example:

```
try:
   num = int(input("Enter a number: "))
   print(10 / num)
except ZeroDivisionError:
   print("Cannot divide by zero.")
except ValueError:
   print("Invalid input.")
```

Example:

```
try:
   # try block --- executed first; if an exception occurs here,
   # execution immediately jumps to the corresponding except block
   strAge = input('Enter your age: ')
   intAge = int(strAge)
   print('You are {} years old.'.format(intAge))

except:
   # except block --- executed only if an exception
   # is raised while executing the try block
   print('Enter your age using digits 0-9!')
```

Block	Purpose	Executed When
try	Contains normal code that may raise an exception	Always executed first
except	Handles errors that occur in try	Only executed if an exception is raised
finally	(optional) Runs cleanup code	Always executed, even if an exception occurs

- The try–except structure prevents program crashes by **catching runtime errors**.
- If no exception occurs, the except block is skipped.
- If an exception occurs, Python **switches to exceptional control flow** and executes the except block.
- This technique makes programs **robust and user-friendly**, especially when handling invalid inputs or file I/O operations.

16.4.3 The Default Exception Handler

If no except block is defined, Python's **default exception handler** terminates the program and prints a traceback.

```
Enter your age: fifteen
Traceback (most recent call last):
```

File "/Users/me/age1.py", line 2, in <module>
intAge = int(strAge)
ValueError: invalid literal for int() with base 10: 'fifteen'

16.4.4 Catching Exceptions of a Given Type

Multiple exception types can be caught individually or grouped:

```
try:
   ...
except (ValueError, TypeError) as e:
   print("Error:", e)
```

Example:

```
try:
   # try block — executed first
   strAge = input('Enter your age: ')
   intAge = int(strAge)
   print('You are {} years old.'.format(intAge))

except ValueError:
   # except block — executed only if a ValueError occurs
   print('Enter your age using digits 0-9!')
```

Example 1 — Valid Input

Enter your age: 22
You are 22 years old.
No exception occurs; the except block is skipped.

Example 2 — Invalid Input

Enter your age: twenty
Enter your age using digits 0-9!
A ValueError is raised; Python switches to **exceptional control flow** and executes the except ValueError block.

- Use except <ExceptionType>: to handle a **specific kind** of error.
- It makes the program **safer** and **more maintainable**.
- Avoid a bare except: unless absolutely necessary.
- Multiple except blocks can be chained to handle **different exceptions separately**.

16.4.5 Multiple Exception Handlers

Each except block handles one type of error, allowing selective responses to different exceptions.

```
try:
   # try block — may raise different kinds of exceptions
   num1 = int(input('Enter the numerator: '))
   num2 = int(input('Enter the denominator: '))
```

```
    result = num1 / num2
    print('Result =', result)

except ValueError:
    # raised if input cannot be converted to an integer
    print('Enter both numbers using digits 0-9!')

except ZeroDivisionError:
    # raised if denominator is zero
    print('Cannot divide by zero! Please enter a nonzero denominator.')

except:
    # handles any other unexpected exceptions
    print('An unexpected error occurred.')
```

Example 1 — Valid Input

```
Enter the numerator: 10
Enter the denominator: 2
Result = 5.0
```

Normal execution; no exception occurs.

Example 2 — Invalid Input

```
Enter the numerator: ten
Enter the denominator: 2
Enter both numbers using digits 0-9!
```

A ValueError occurs while converting "ten" to integer.

Example 3 — Division by Zero

```
Enter the numerator: 10
Enter the denominator: 0
Cannot divide by zero! Please enter a nonzero denominator.
```

A ZeroDivisionError occurs during the division operation.

Example 4 — Other Unexpected Exception

```
Enter the numerator:
An unexpected error occurred.
```

A generic except block catches an unexpected error (e.g., empty input or EOF).

16.4.6 Controlling Exceptional Control Flow

Exceptions can be raised intentionally using the raise statement:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Denominator cannot be zero.")
    return a / b
```

This enforces **controlled error management** and promotes **robust software design**.

16.5 MODULES AS NAMESPACES

module to describe a file containing Python code. When the module is executed (imported), then the module is (also) a namespace. This namespace has a name, which is the name of the module. In this namespace will live the names that are defined in the global scope of the module: the names of functions, values, and classes defined in the module. These names are all referred to as the module's attributes.

16.5.1 Modules and Attributes

A **module** is a Python file that serves as a **separate namespace** containing definitions of variables, functions, and classes.

```
# file: math_ops.py
def add(x, y): return x + y
def sub(x, y): return x - y
```

When imported, Python creates a **module object** with attributes that correspond to these definitions:

```
import math_ops
print(math_ops.add(5, 3))
```

Once a module is imported, the Python built-in function dir() can be used to view all the module's attributes:

```
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'hypot', 'isinf',
'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

16.5.2 What Happens During Import

When import is executed:

1. Python searches for the module in the **module search path**.
2. If found, it creates a **new namespace** for that module.
3. The module's code executes, defining its internal variables.

16.5.3 Module Search Path

The search path includes:

- The current directory,
- Standard library directories,
- Custom paths defined in sys.path.

You can view it using:
import sys
print(sys.path)

16.5.4 Top-Level Module

The module where execution begins is treated as the **top-level module** (with name '__main__').
Its global namespace becomes the **program's primary namespace**.

16.5.5 Different Ways to Import

Python provides several import mechanisms:

- import math
- from math import sqrt
- from math import sin as sine

Each approach controls **how names are imported** into the local namespace, affecting visibility and potential conflicts.

16.6 CLASSES AS NAMESPACES

A namespace is associated with every class. Python uses namespaces in a clever way to implement classes and class methods.

16.6.1 A Class Is a Namespace

Each **class** in Python defines its own namespace, storing attributes (variables) and methods (functions).

```
class Student:
    school = "ANU"      # class variable
    def __init__(self, name):
        self.name = name  # instance variable
```

school belongs to the **class namespace**, while name belongs to the **object's namespace**.

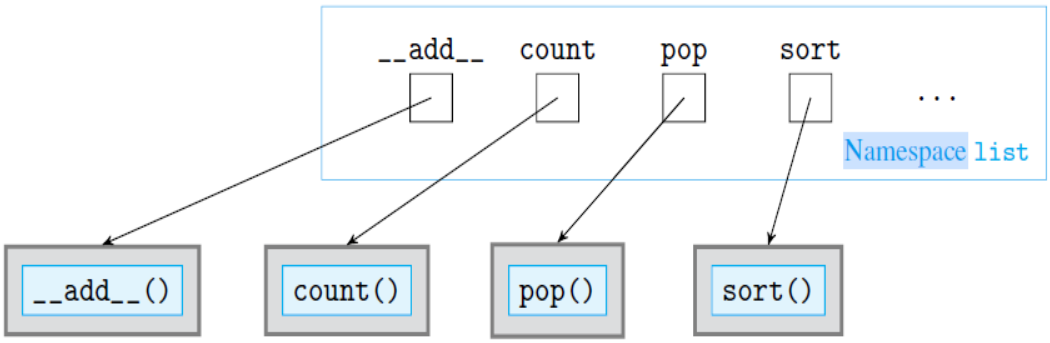


Figure 16.5 The namespace list and its attributes.

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
...,
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

16.6.2 Class Methods Are Functions in Class Namespace

```
print(Student.school)
s1 = Student("Lavanya")
print(s1.name)
```

Each object accesses class variables through the shared class namespace, ensuring consistent structure across instances.

16.7 SUMMARY

- **Namespaces** organize names and prevent conflicts.
- **Local and global scopes** define variable visibility.
- **Encapsulation** hides details and promotes modular design.
- **Exceptions** control abnormal program flow.
- **Modules and classes** provide hierarchical namespaces for large programs.

16.8 TECHNICAL TERMS

- **Namespace:** Mapping from names to objects.
- **Scope:** The region where a variable name is visible.
- **Encapsulation:** Hiding internal data from external access.
- **Global variable:** Defined at the module level.
- **Local variable:** Defined inside a function.
- **Exception:** An event that alters control flow.
- **Module:** A file that defines its own namespace.

16.9 SELF-ASSESSMENT QUESTIONS

1. Define namespace.
2. What is encapsulation and how does it improve modularity?
3. Explain the difference between local and global variables.
4. What is the LEGB rule for variable lookup?
5. Write an example demonstrating try and except.
6. What happens during the import process?
7. Explain how a class serves as a namespace.
8. Discuss the importance of exception handling in robust programming.

16.10 SUGGESTED READINGS

- Ljubomir Perković, *Introduction to Computing Using Python: An Application Development Focus*, Wiley (2012).
- Mark Lutz, *Learning Python*, O'Reilly Media.
- Allen B. Downey, *Think Python: How to Think Like a Computer Scientist*, O'Reilly.

Dr. Vasantha Rudramalla

LESSON- 17

GRAPHICAL USER INTERFACES (GUI)

AIMS AND OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the concept and components of Graphical User Interfaces (GUIs).
- Develop basic GUI applications using the tkinter module in Python.
- Use common widgets such as labels, buttons, and entry fields effectively.
- Apply event-driven programming principles to handle user interactions.
- Design modular and reusable GUI programs using object-oriented programming (OOP) concepts.
- Implement a complete GUI-based application such as a calculator using tkinter.

STRUCTURE

17.1 INTRODUCTION

Most programs so far have used text-based input and output, where users type commands and read printed responses.
However, modern applications use graphical user interfaces (GUIs) — windows with buttons, menus, labels, and other interactive components.
A GUI allows users to interact with programs visually, making software more intuitive and user-friendly.
Python provides a standard GUI toolkit called tkinter, which supports event-driven programming — the foundation of modern user interfaces.

17.2 BASICS OF TKINTER GUI DEVELOPMENT

A graphical user interface (GUI) consists of basic visual building blocks such as buttons,labels, text entry forms, menus, check boxes, and scroll bars, among others, all packed inside a standard window. Building blocks are commonly referred to as widgets. To develop GUIs, a developer will require a module that makes such widgets available. We will use the module tkinter that is included in the Standard Library. In this section, we explain the basics of GUI development using tkinter:

17.2.1 What is tkinter?

tkinter is Python's built-in module for GUI development. It acts as a bridge between Python and the **Tcl/Tk** GUI framework.
Tkinter allows developers to:

- Create windows, buttons, labels, and text boxes.
- Handle user input through events.
- Build interactive desktop applications easily.

To start using tkinter:

```
from tkinter import Tk, Label
root = Tk()                # Create main window
lbl = Label(root, text="Hello, GUI World!")
lbl.pack()                 # Place label in window
root.mainloop()            # Run event loop
```

Output:

A simple window appears displaying "Hello, GUI World!".

Figure 17.1 A text label. The Label widget created with the text argument will display a text label.

17.2.2 The Main Window and Event Loop

Every tkinter application has:

1. Root Window (Tk()) — the main window created at startup.
2. Widgets — interface elements such as Button, Label, Entry, etc.
3. Event Loop (mainloop()) — continuously listens for user actions (mouse clicks, key presses) and updates the interface.

The call to mainloop() keeps the program running until the user closes the window.

17.2.3 Common Widgets

Widget	Purpose	Example
Label	Displays text or images	Label(root, text="Welcome!")
Button	Performs an action when clicked	Button(root, text="Click Me")
Entry	Single-line text input	Entry(root)
Text	Multi-line text input	Text(root)
Frame	Container for grouping widgets	Frame(root)
Canvas	For drawing shapes and graphics	Canvas(root, width=200, height=100)

17.2.4 Geometry Management

Tkinter provides layout managers to control widget placement:

- pack() — stacks widgets vertically or horizontally.
- grid() — arranges widgets in rows and columns.
- place() — positions widgets by absolute coordinates.

```
from tkinter import *
root = Tk()
Label(root, text="Name").grid(row=0, column=0)
Entry(root).grid(row=0, column=1)
Button(root, text="Submit").grid(row=1, column=1)
root.mainloop()
```

17.2.5 Example — Simple Login Window

```
from tkinter import *
root = Tk()
root.title("Login")
Label(root, text="Username").grid(row=0)
Label(root, text="Password").grid(row=1)
Entry(root).grid(row=0, column=1)
Entry(root, show='*').grid(row=1, column=1)
Button(root, text='Login').grid(row=2, column=1)
root.mainloop()
```

Explanation:
This simple interface uses the grid() geometry manager for organized alignment.

17.3 EVENT-BASED TKINTER WIDGETS

Widgets have an interactive behavior that needs to be programmed using a style of programming called *event-driven programming*. In addition to GUI development, event-driven programming is also used in the development of computer games and distributed client/server applications, among others.

17.3.1 Event-Driven Programming

Traditional programs follow a **sequential flow**, but GUI programs are **event-driven**. Events occur when the user interacts with the GUI — such as clicking a button or pressing a key. Tkinter uses **callback functions** to respond to these events.

Figure 17.2 GUI with one Button widget

```
>>>
Day: 07 Jul 2011
Time: 23:42:47 PM

from tkinter import Tk, Button
from time import strftime, localtime

def clicked():
   'prints day and time info'
   time = strftime('Day: %d %b %Y\nTime: %H:%M:%S %p\n', localtime())
   print(time)

root = Tk()

# create button labeled 'Click it' and event handler clicked()
button = Button(root,
            text='Click it',       # text displayed on the button
            command=clicked)       # function called when button is clicked
button.pack()
root.mainloop()
```

This example shows that in **event-driven programming**, control flow depends on **user actions** (events) rather than a predefined sequence of commands.

The **command parameter** acts as a **callback function**, executed only when the user interacts with the GUI (in this case, clicks the button).

17.3.2 Binding Events

You can "bind" an event (e.g., mouse click) to a function using the command parameter or the bind() method.
Example:

```
from tkinter import *
def greet():
   print("Hello, User!")
root = Tk()
btn = Button(root, text="Greet", command=greet)
btn.pack()
root.mainloop()
```

When the user clicks the button, the greet() function executes.

17.3.3 Keyboard and Mouse Events

You can bind specific events to widgets:

```
def key_pressed(event):
   print("You pressed:", event.char)

root.bind("<Key>", key_pressed)
```

Event patterns include:

- <Button-1> → Left mouse click
- <Button-3> → Right mouse click
- <Key> → Any key press
- <Return> → Enter key
- <Motion> → Mouse movement

Example — Handling Mouse Events in tkinter

Mouse events allow a program to respond to user interactions such as **clicks**, **double-clicks**, and **right-clicks**.

In tkinter, these actions are handled by **event binding**, where a specific mouse action is associated with an event-handling function.

Program: Demonstrating Mouse Click Events

```
import tkinter as tk

def on_click(event):
   """Called when a mouse button is pressed."""
   x, y = event.x, event.y
   btn = event.num  # button number: 1=left, 2=middle, 3=right
```

```
    lbl.config(text=f"Clicked: Button {btn} at ({x}, {y})")
    print(f"CLICK: Button {btn} at coordinates ({x}, {y})")

def on_double_click(event):
    """Called when the left button is double-clicked."""
    lbl.config(text=f"Double-click at ({event.x}, {event.y})")
    print(f"DOUBLE CLICK at ({event.x}, {event.y})")

def on_right_click(event):
    """Called when the right mouse button is pressed."""
    lbl.config(text=f"Right-click at ({event.x}, {event.y})")
    print(f"RIGHT CLICK at ({event.x}, {event.y})")

# Create main window
root = tk.Tk()
root.title("Mouse Events Demo")
root.geometry("420x160")

# Create label for displaying event information
lbl = tk.Label(root, text="Click anywhere inside the window", font=("Arial", 12))
lbl.pack(pady=15, fill="x")

# Bind mouse actions to event handlers
root.bind("<Button-1>", on_click)          # Left-click
root.bind("<Double-Button-1>", on_double_click)  # Double-click
root.bind("<Button-3>", on_right_click)     # Right-click

root.mainloop()
```

Explanation

1. **Event Binding:**
    - The bind() function associates a mouse event with a handler (callback function).
    - Syntax:
    - widget.bind("<EventPattern>", callback_function)
    - Example events:
        - <Button-1> — Left mouse click
        - <Double-Button-1> — Double left-click
        - <Button-3> — Right mouse click
        -
2. **Event Object:**
   Each callback receives an event object that contains details such as:
    - event.x, event.y → Position of the cursor within the window.
    - event.num → Mouse button number.

- event.widget → Widget where the event occurred.

3. **Label Update:**
   - The label lbl dynamically updates to show which mouse button was pressed and the coordinates of the click.
   - The same message is printed in the console for verification.

4. **Window Setup:**
   - The Tk() function creates the main window.
   - Label() displays messages to the user.
   - geometry() defines the window size.
   - mainloop() starts the event loop, keeping the window active.

Output

When the program runs, a window appears with the text:

Click anywhere inside the window

User Action	Label Output (in GUI)	Console Output
Left-click at (120, 45)	Clicked: Button 1 at (120, 45)	CLICK: Button 1 at coordinates (120, 45)
Double-click at (150, 60)	Double-click at (150, 60)	DOUBLE CLICK at (150, 60)
Right-click at (100, 30)	Right-click at (100, 30)	RIGHT CLICK at (100, 30)

This example demonstrates how tkinter enables **event-driven programming** for **mouse actions**. By using bind() and handling the event object, programs can respond interactively to user input — a fundamental concept in building responsive GUIs.

17.3.4 Example — Interactive Counter

```
from tkinter import *
count = 0
def increase():
    global count
    count += 1
    label.config(text=f"Count: {count}")
root = Tk()
label = Label(root, text="Count: 0")
label.pack()
Button(root, text="Add 1", command=increase).pack()
root.mainloop()
```

Output:

Each button click updates the counter value dynamically

17.4 DESIGNING GUIS

Designing a Graphical User Interface (GUI) involves creating a visual environment through which users can interact intuitively with a program. A well-designed GUI enhances user experience by providing clear navigation, consistent layout, and immediate feedback to user actions. In Python, GUI design using tkinter focuses on arranging widgets logically, managing user input, and ensuring responsiveness through event-driven behavior. Effective GUI design balances functionality, aesthetics, and usability, ensuring that the interface not only looks appealing but also supports the underlying logic of the application in a structured and efficient manner.

17.4.1 GUI Design Principles

A good GUI should be:

- **Intuitive:** Easy to use without instructions.
- **Consistent:** Uses uniform fonts, colors, and layout.
- **Responsive:** Reacts quickly to user actions.
- **Error-Tolerant:** Handles invalid input gracefully.

Widget Canvas:
The Canvas widget is a fun widget that can display drawings consisting of lines and geometrical objects. You can think of it as a primitive version of turtle graphics. (In fact, turtle graphics is essentially a tkinter GUI.) We illustrate the Canvas widget by building a very simple pen drawing application.The application consists of an initially empty canvas. The user can draw curves inside the canvas using the mouse. Pressing the left mouse button starts the drawing of the curve.Mouse motion while pressing the button moves the pen and draws the curve.

Example:

Figure 17.3 Pen drawing app.

Code:

```
from tkinter import Tk, Canvas

# event handlers
def begin(event):
```

```
    global oldx, oldy
    oldx, oldy = event.x, event.y    # record current mouse position

def draw(event):
    global oldx, oldy
    canvas.create_line(oldx, oldy, event.x, event.y)  # draw a line segment
    oldx, oldy = event.x, event.y                     # update coordinates

root = Tk()
oldx, oldy = 0, 0  # initialize mouse coordinates

# create a canvas
canvas = Canvas(root, height=100, width=150, bg='white')

# bind mouse events
canvas.bind("<Button-1>", begin)          # mouse click event
canvas.bind("<B1-Motion>", draw)          # mouse drag event

canvas.pack()
root.mainloop()
```

When the user clicks and drags the mouse over the canvas:

- A continuous line is drawn following the mouse movement.
- Releasing the mouse button stops drawing.

This forms the basis for paint applications, signature capture tools, and interactive graphics programs.

The **Canvas widget** in tkinter provides a powerful area for **graphics, shapes, and interactive drawings**.It supports several built-in methods for drawing and manipulating shapes such as lines, rectangles, and ovals.Each shape drawn on the canvas is assigned a unique **item ID**, which can later be used to move, modify, or delete that shape.

Some Canvas methods:

Method	**Description**
create_line(x1, y1, x2, y2, …)	Creates one or more line segments connecting the specified coordinate points (x1, y1), (x2, y2), etc. Returns the ID of the created line item.
create_rectangle(x1, y1, x2, y2)	Draws a rectangle with opposite vertices at (x1, y1) and (x2, y2). Returns the ID of the constructed rectangle.
create_oval(x1, y1, x2, y2)	Creates an oval (or circle) inscribed within a rectangle defined by the corner points (x1, y1) and (x2, y2). Returns the ID of the constructed oval.
delete(ID)	Deletes the item identified by its ID from the canvas. If called without arguments (delete('all')), it clears the entire canvas.
move(item, dx, dy)	Moves a canvas item horizontally by dx units and vertically by dy units relative to its current position.

17.4.2 Steps in GUI Design

1. **Define Requirements:** Identify inputs, outputs, and interactions.
2. **Sketch Layout:** Determine arrangement of widgets.
3. **Implement Layout:** Use frames and geometry managers.
4. **Connect Functionality:** Bind callbacks and logic.
5. **Test Usability:** Verify that navigation and interactions are intuitive.

17.4.3 Example — Temperature Converter

```
from tkinter import *
def convert():
    c = float(celsius.get())
    f = (c * 9/5) + 32
    result_label.config(text=f"{f:.2f} °F")
root = Tk()
root.title("Celsius to Fahrenheit")
Label(root, text="Celsius:").grid(row=0, column=0)
celsius = Entry(root)
celsius.grid(row=0, column=1)
Button(root, text="Convert", command=convert).grid(row=1, column=0, columnspan=2)
result_label = Label(root, text="Result: ")
result_label.grid(row=2, column=0, columnspan=2)
root.mainloop()
```

17.5 OOP FOR GUIS

the OOP approach to designing GUIs. This approach will make our GUI applications far easier to reuse.

Example — Displaying Date and Time Using a Message Box
This program demonstrates how to handle a **button click event** and display the current date and time using a **popup message box** from the tkinter.messagebox module.

Program Code

```
from tkinter import Tk, Button
from tkinter.messagebox import showinfo
from time import strftime, localtime
def clicked():
    """Displays the current day and time information."""
    time = strftime('Day: %d %b %Y\nTime: %H:%M:%S %p\n', localtime())
    showinfo(message=time)  # display output in a popup message box

# Create main window
root = Tk()
```

```
root.title("Date and Time Display")

# Create button labeled 'Click it' and assign the event handler
button = Button(root,
                text='Click it',       # text on top of button
                font=('Arial', 14),
                command=clicked)       # event handler function
button.pack(pady=20)

root.mainloop()
```

Output

When the program runs, a window appears with a single button labeled **"Click it"**.

When the user clicks the button:

A message box pops up showing the current date and time, for example:

Day: 29 Oct 2025

Time: 03:24:45 PM

This example illustrates how tkinter integrates GUI widgets (like buttons) with event-driven behavior. Using showinfo(), you can make the interface more interactive and user-friendly by displaying information directly in a popup window rather than the console.

17.5.1 Class-Based GUI Design

Using **object-oriented programming (OOP)**, we can design modular, reusable GUI components.

Example:

```
from tkinter import *
class CounterApp:
    def __init__(self, root):
        self.count = 0
        self.label = Label(root, text="Count: 0")
        self.label.pack()
        Button(root, text="Increase", command=self.increment).pack()
    def increment(self):
        self.count += 1
        self.label.config(text=f"Count: {self.count}")
root = Tk()
app = CounterApp(root)
root.mainloop()
```

Here, the GUI's **state** (count) and **behavior** (increment) are encapsulated within a class.

Output

When the program runs, a window appears with:

- A label showing the text:
  Count: 0

- A button labeled **Increase**.

User Interaction:

- Clicking the **Increase** button repeatedly updates the label:
- Count: 1
- Count: 2
- Count: 3
- ...

This example shows how **OOP concepts integrate seamlessly with tkinter GUI design**. By encapsulating interface elements and logic inside a class:

- The GUI becomes **organized and scalable**.
- Each object maintains its **own state**, allowing multiple independent GUIs if needed.

Such an approach is widely used in larger GUI projects where multiple windows, widgets, or components interact cohesively.

17.5.2 Advantages of OOP in GUI Design

- **Encapsulation:** GUI logic is grouped in one class.
- **Reusability:** Components can be reused in other programs.
- **Maintainability:** Easier to debug and update.
- **Scalability:** Ideal for complex interfaces.

17.6 CASE STUDY — DEVELOPING A CALCULATOR

This case study demonstrates how to combine tkinter widgets, event handling, and OOP principles to create a simple **GUI Calculator**.

17.6.1 Program Design

Features:

- Numeric buttons (0–9)
- Operators (+, –, ×, ÷)
- Clear (C) and Equal (=) functions

17.6.2 Implementation

```
from tkinter import *

class Calculator:
    def __init__(self, root):
        self.expression = ""
        self.input_text = StringVar()

        input_frame = Frame(root)
        input_frame.pack()

        input_field = Entry(input_frame, textvariable=self.input_text, width=25, font=('Arial', 18))
```

```
        input_field.grid(row=0, column=0, columnspan=4)
        input_field.pack(ipady=8)

        btns_frame = Frame(root)
        btns_frame.pack()

        buttons = [
            ['7', '8', '9', '/'],
            ['4', '5', '6', '*'],
            ['1', '2', '3', '-'],
            ['0', 'C', '=', '+']
        ]

        for row in buttons:
            for btn in row:
                Button(btns_frame, text=btn, width=5, height=2, font=('Arial', 14),
                       command=lambda b=btn: self.on_click(b)).pack(side='left')
            Frame(btns_frame, height=2).pack()

    def on_click(self, key):
        if key == 'C':
            self.expression = ""
            self.input_text.set("")
        elif key == '=':
            try:
                result = str(eval(self.expression))
                self.input_text.set(result)
                self.expression = result
            except:
                self.input_text.set("Error")
        else:
            self.expression += key
            self.input_text.set(self.expression)

root = Tk()
root.title("Simple Calculator")
Calculator(root)
root.mainloop()
```

17.6.3 Output

A fully functional calculator GUI that performs basic arithmetic operations with real-time display updates.

When the program runs, a window appears with:

- A **display box** at the top showing the current input or result.
- Four rows of buttons for digits and arithmetic operations (+, –, ×, /).
- Buttons for **Clear (C)** and **Equals (=)** operations.

Example Interaction:

User Input	Display Output
7 + 8 * 2 =	23
100 / 4 =	25.0
9 + (Error) → "Error"	Error message displayed

This case study illustrates the practical use of tkinter for:

- **Interface design**
- **Event handling**
- **OOP-based GUI architecture**
- **Real-time user feedback**

The Calculator GUI demonstrates how Python can create **interactive, user-friendly, and maintainable applications** with minimal code.

17.7 SUMMARY

- GUIs allow **visual interaction** with programs.
- **tkinter** provides an easy, cross-platform toolkit for GUI design.
- GUIs follow an **event-driven model** — responding to user inputs.
- Widgets are arranged using **geometry managers** like pack(), grid(), and place().
- **OOP** simplifies GUI design by encapsulating logic and interface.
- Practical GUI applications can be built using modular, reusable code.

17.8 TECHNICAL TERMS

- Term
- GUI
- Event
- Widget
- Callback
- Mainloop
- Geometry Manager

17.9 SELF-ASSESSMENT QUESTIONS

1. Define a GUI. How is it different from command-line interaction?
2. Explain the role of mainloop() in tkinter.
3. What are widgets? List five commonly used tkinter widgets.
4. Differentiate between pack(), grid(), and place() layout methods.
5. Describe event-driven programming with an example.
6. Explain how classes are used to design object-oriented GUIs.
7. Write a Python tkinter program to design a simple “Login” window.
8. Describe the steps involved in GUI design.
9. Explain how the command parameter is used in buttons.
10. Discuss how a calculator GUI can be implemented using tkinter.

17.10 SUGGESTED READINGS

- Ljubomir Perković, *Introduction to Computing Using Python: An Application Development Focus*, Wiley (2012).
- Mark Lutz, *Programming Python*, O'Reilly Media.
- Alan D. Moore, *Python GUI Programming with Tkinter*, Packt Publishing.
- John Zelle, *Python Programming: An Introduction to Computer Science*, Franklin, Beedle & Associates.

Mrs. Appikatla Pushpa Latha

LESSON- 18

THE WORLD WIDE WEB (WWW)

AIMS AND OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the architecture and functioning of the World Wide Web (WWW).
- Explain the roles of web servers, clients, and HTTP communication protocols.
- Identify the components of a URL (Uniform Resource Locator) and its purpose in locating web resources.
- Describe the structure and syntax of HTML (HyperText Markup Language).
- Use Python's urllib and html.parser modules to retrieve and analyze web content.
- Apply regular expressions (regex) for pattern matching and data extraction from web pages.
- Develop simple web automation tools such as data extractors and crawlers.
- Understand the design and implementation of a recursive web crawler for navigating and analyzing web pages.

STRUCTURE

18.1 INTRODUCTION

The World Wide Web (WWW) is a system of interlinked hypertext documents that can be accessed via the Internet using web browsers. It enables users to navigate information using hyperlinks, retrieve content from remote servers, and communicate using standardized protocols.

Python provides modules and libraries that allow programs to interact with web resources — downloading pages, parsing HTML, following links, and even automating browsing tasks. This lesson covers the underlying structure of the Web and explores how Python can interface with it programmatically**.**

18.2 THE WORLD WIDE WEB

The World Wide Web (WWW or, simply, the web) is a distributed system of documents linked through hyperlinks and hosted on web servers across the Internet.

18.2.1 Web Servers and Web Clients

A program that requests a resource from a web server is called a *web client*. The web server receives the request and sends the requested resource (if if exists) back to the client.Your favorite browser (whether it is Chrome, Firefox, Internet Explorer, or Safari) is a web client. A browser has capabilities in addition to being able to request and receive web resources. It also processes the resource and displays it, whether the resource is a web page, text document, image, video, or other multimedia. Most important, a web browser displays the hyperlinks contained in a web page and allows the user to navigate between web pages by just clicking on the hyperlinks.
The Web functions through a client-server model:

- A web server stores and delivers web pages upon request.
- A web client (browser or Python script) requests resources from the server.

When a client requests a page (e.g., index.html), the server sends it using the HTTP protocol, and the client renders it for display.

Example Flow:

1. User enters a URL in a browser.
2. Browser sends an HTTP request to the web server.
3. Server responds with the HTML content.
4. Browser interprets and displays the page.

18.2.2 “Plumbing” of the WWW

The core “plumbing” of the Web involves:

- TCP/IP: Provides the communication framework between computers.
- HTTP: Defines how requests and responses are exchanged.
- HTML: Describes the structure and content of web pages.
- DNS (Domain Name System): Maps human-readable addresses (like www.python.org) to IP addresses.

These components work seamlessly to enable resource sharing and information exchange on a global scale.

18.2.3 Naming Scheme: Uniform Resource Locator (URL)

A URL (Uniform Resource Locator) identifies and locates a resource on the Web.

General Format:

scheme://host:port/path?query#fragment

Example:

https://www.example.com:443/articles/python.html?topic=networking#links

Component	Meaning
https	Protocol used (HTTP Secure).
www.example.com	Server name.
443	Port number (optional).
/articles/python.html	Path to the resource.
?topic=networking	Query parameters.
#links	Reference to a section within the page.

18.2.4 Protocol: HyperText Transfer Protocol (HTTP)

HTTP defines how a client and server exchange information.

Common HTTP request methods:

- GET – Retrieve data (most common).
- POST – Send data to a server (e.g., form submission).
- HEAD – Retrieve only header information.
- PUT / DELETE – Modify or remove data (less common in browsers).

18.2.5 HyperText Markup Language (HTML)

HTML is the markup language used to define the structure and content of web pages.

Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <h1>Welcome to the Web</h1>
    <p>This is a simple web page.</p>
    <a href="https://www.python.org">Visit Python</a>
  </body>
</html>
```

18.2.6 HTML Elements

HTML documents are composed of tags that define elements:

Tag	Purpose
…	Heading text
…	Paragraph
<a href="…">	Hyperlink
<img src="…">	Image
…	Section grouping

Each element can have attributes that provide additional information, e.g., href, src, or style.

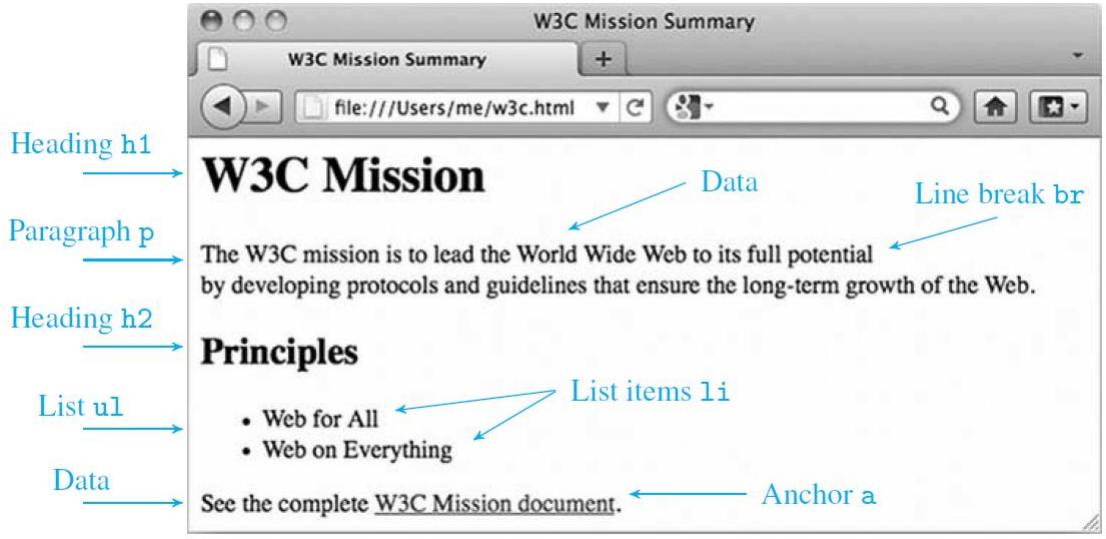


Figure 18.1 Web page w3c.html.

In general, an HTML element consists of three components:

1. A pair of tags: the start tag and the end tag
2. Optional attributes within the start tag
3. Other elements or data between the start and end tag

In HTML source file w3c.html, there is an example of an element (title) contained inside another element (head):

<head><title>W3C Mission Summary</title></head>

18.2.7 Tree Structure of an HTML Document

An HTML document can be represented as a tree structure, where each element (node) can have children.

<html>
└── <body>
    ├── <h1>
    └── <p>

This hierarchical organization allows structured parsing and processing of web content.

Figure 18.2 shows all the elements in file w3c.html. The figure makes explicit what element is contained in another and the resulting tree structure of the document. This tree structure and the HTML elements together determine the layout of the web page.

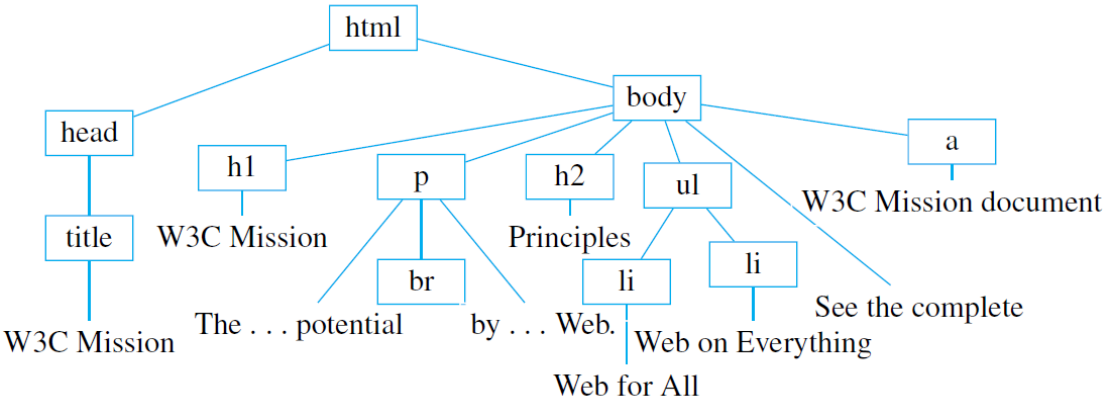


Figure 18.2 Structure of w3c.html.

18.2.8 Anchor Element and Links

The anchor tag <a> creates a hyperlink that connects documents:

```
<a href="https://www.python.org">Python Website</a>
```

Absolute links contain full URLs, while relative links point to pages relative to the current document's location.

18.3 PYTHON WWW API

Python provides standard library modules to access and process web resources.

18.3.1 Module urllib.request

The urllib.request module handles opening and reading URLs.

Example — Retrieving a Web Page:

```
from urllib.request import urlopen

url = 'https://www.python.org'
```

```
page = urlopen(url)
html = page.read().decode('utf-8')

print(html[:500])  # print first 500 characters
```

Explanation:

- urlopen() sends an HTTP request.
- read() returns raw bytes.
- .decode('utf-8') converts bytes into text.

18.3.2 Module html.parser

Python's html.parser provides a class-based mechanism to analyze HTML structure.

Example:

```
from html.parser import HTMLParser
class MyParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
    def handle_endtag(self, tag):
        print("End tag:", tag)
    def handle_data(self, data):
        print("Data:", data)
parser = MyParser()
parser.feed("<html><body><h1>Hello</h1></body></html>")
```

Output:

```
Start tag: html
Start tag: body
Start tag: h1
Data: Hello
End tag: h1
End tag: body
End tag: html
```

18.3.3 Overriding HTML Parser Handlers

You can extend HTMLParser to collect specific information such as all hyperlinks.

```
class LinkParser(HTMLParser):
```

```
        def __init__(self):
            super().__init__()
            self.links = []
        def handle_starttag(self, tag, attrs):
            if tag == 'a':
    for (attr, value) in attrs:
        if attr == 'href':
            self.links.append(value)

parser = LinkParser()
parser.feed('<a href="https://example.com">Example</a>')
print(parser.links)
```

Output:

```
['https://example.com']
```

18.3.4 Module urllib.parse

urllib.parse provides functions to handle and decompose URLs.

Example:

```
from urllib.parse import urlparse
url = 'https://www.python.org:443/doc/index.html?lang=en#section2'
components = urlparse(url)
print(components.scheme)
print(components.netloc)
print(components.path)
print(components.query)
print(components.fragment)
```

Output:

```
https
www.python.org:443
/doc/index.html
lang=en
section2
```

18.4 CASE STUDY: WEB CRAWLER

A web crawler is a program that automatically downloads web pages and extracts hyperlinks for further exploration.

The figure **18.4** illustrates the interconnected structure of five HTML pages — **one.html**, **two.html**, **three.html**, **four.html**, and **five.html** — and the frequency of specific words found within each page.

Each rectangle represents a single web page, and the labels inside indicate **keywords** (e.g., *Beijing*, *Paris*, *Chicago*) along with their **frequency counts**. The arrows represent **hyperlinks** connecting one page to another, showing how a **web crawler** might traverse the web structure.
For example:

- one.html contains the words *Beijing* (×3), *Paris* (×5), and *Chicago* (×5), and links to two.html and three.html.
- three.html mentions *Chicago* (×3) and *Beijing* (×6), linking onward to four.html.
- five.html includes *Nairobi* (×7) and *Bogota* (×2), linking to four.html.

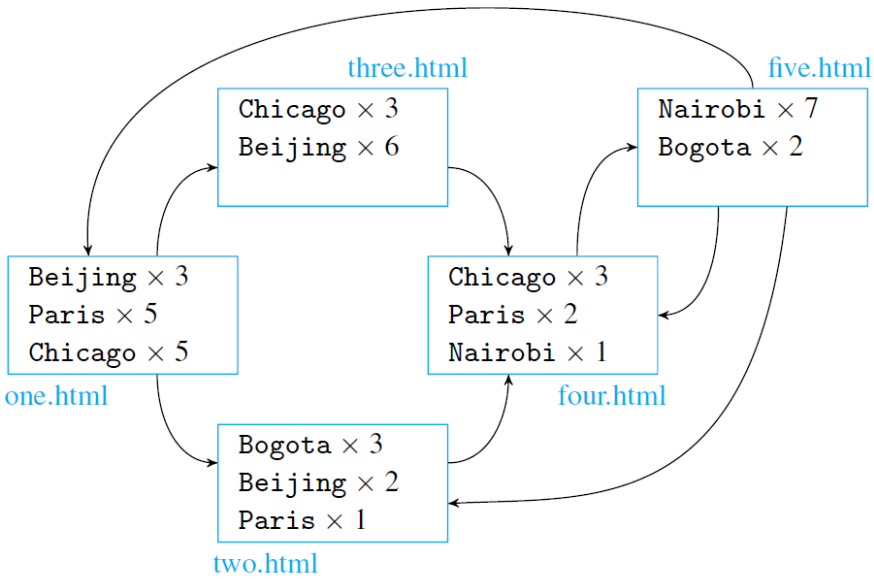


Figure 18.4 Five linked web pages.

This web structure demonstrates how a **recursive web crawler** analyzes not only **link relationships** but also **page content** by counting keyword occurrences and following references between documents. Such analysis forms the foundation for **search engine indexing** and **page ranking algorithms**, which assess the importance and relevance of web pages based on their content and connectivity.

18.4.1 Recursive Crawler (Version 0.1)

This version improves upon the basic crawler by adding recursion and duplicate page handling.

It uses a set named visited to track which web pages have already been processed, preventing redundant crawling and infinite loops due to circular links.

```
visited = set()  # initialize visited to an empty set


def crawl2(url):
    '''A recursive web crawler that calls analyze()
    on every visited web page'''

    # add url to the set of visited pages
    global visited
    visited.add(url)

    # analyze() returns a list of hyperlink URLs in web page 'url'
    links = analyze(url)

    # recursively continue crawl from every link in 'links'
    for link in links:
        # follow link only if not visited
        if link not in visited:
            try:
                crawl2(link)
            except:
                pass
```

Explanation

1. Set Initialization (visited):
   A Python set is used to keep track of URLs that have already been visited.
   Since sets automatically ignore duplicates, this ensures that each web page is analyzed only once.

2. Recursive Design:
   - Each time the function crawl2() is called with a new URL, that page is analyzed (e.g., its HTML content is parsed to extract links).
   - Then the crawler iterates over all hyperlinks found on that page.
   - For each unvisited link, it calls itself recursively — continuing the crawl process deeper into the link structure.

3. Avoiding Infinite Loops:
   Without the visited check, the crawler could get stuck following circular links (e.g., A → B → A).

The visited set prevents this by ensuring that previously seen URLs are skipped.

4. Error Handling (try–except):
   - The try block attempts to crawl a linked page.
   - The except block catches any exceptions (such as connection errors or invalid URLs) and silently ignores them.
   - This keeps the crawler from stopping unexpectedly when encountering problematic pages.

5. Global Declaration:
   - The global visited declaration is optional but serves as a clear reminder that the variable belongs to the global scope, shared across recursive calls.

Output Behavior

When the crawler starts with an initial page, say:
crawl2("https://example.com/one.html")

It performs the following actions:

1. Adds one.html to the visited set.
2. Extracts all links from one.html (e.g., two.html, three.html).
3. For each new link, recursively calls crawl2(link).
4. Continues until all reachable pages are analyzed.

Integration with analyze()

A typical implementation of the analyze() function, used in this crawler, might look like:

```
from urllib.request import urlopen
from html.parser import HTMLParser


class LinkCollector(HTMLParser):
    def __init__(self):
        super().__init__()
        self.links = []
    def handle_starttag(self, tag, attrs):
        if tag == 'a':
            for (attr, val) in attrs:
                if attr == 'href' and val.endswith('.html'):
                    self.links.append(val)


def analyze(url):
```

```
print(f"Analyzing: {url}")
try:
    content = urlopen(url).read().decode('utf-8')
    collector = LinkCollector()
    collector.feed(content)
    return collector.links
except:
    return []
```

Summary

This recursive crawler version introduces three major improvements:

1. Recursion to automatically traverse linked pages.
2. A visited set to manage page tracking and prevent loops.
3. Exception handling to ensure robustness when facing inaccessible or malformed pages.

Together, these enhancements make the crawler a practical, extensible foundation for real-world web scraping, indexing, or search engine prototypes.

18.4.2 Recursive Crawler (Version 0.2)

The goal of Version 0.2 is to:

1. Follow every hyperlink discovered in the currently analyzed page.
2. Avoid revisiting pages that have already been processed.
3. Continue crawling until all reachable pages within a given web domain are visited.

```
visited = set()   # initialize visited to an empty set


def crawl2(url):
    '''A recursive web crawler that calls analyze()
    on every visited web page'''

    global visited          # indicates the use of the global variable
    visited.add(url)        # mark current page as visited

    # analyze() returns a list of hyperlink URLs found in web page 'url'
    links = analyze(url)

    # recursively continue crawl from every link in 'links'
    for link in links:
```

```
        # follow link only if not yet visited
        if link not in visited:
            try:
                crawl2(link)   # recursive call
            except Exception as e:
                # safely ignore network or parsing errors
                print("Skipping:", link, "Reason:", e)
                pass
```

analyze()

The crawler depends on a separate helper function that retrieves and parses each web page:

```
from urllib.request import urlopen
from html.parser import HTMLParser

class LinkCollector(HTMLParser):
    """Collects all hyperlinks from an HTML page."""
    def __init__(self):
        super().__init__()
        self.links = []
    def handle_starttag(self, tag, attrs):
        if tag == 'a':
            for (attr, value) in attrs:
                if attr == 'href' and value.endswith('.html'):
                    self.links.append(value)

def analyze(url):
    """Returns a list of hyperlinks extracted from a given URL."""
    print("Analyzing:", url)
    try:
        content = urlopen(url).read().decode('utf-8')
        collector = LinkCollector()
        collector.feed(content)
        return collector.links
    except:
```

```
    return []
```

Execution Flow

Suppose the program begins with:

crawl2("https://example.com/one.html")

Step-by-Step Process:

1. one.html is analyzed and added to visited.
2. analyze("one.html") returns a list of links → ['two.html', 'three.html'].
3. crawl2('two.html') is called; two.html is analyzed, producing ['four.html'].
4. The recursion continues until all reachable pages (two.html, three.html, four.html, five.html, etc.) have been processed.
5. Each page is visited only once, even if multiple pages link back to it.

Sample Output

```
Analyzing: https://example.com/one.html
Analyzing: https://example.com/two.html
Analyzing: https://example.com/four.html
Analyzing: https://example.com/five.html
Analyzing: https://example.com/three.html
```

Advantages of Recursion in Crawling

- Simplifies the program structure by letting each call handle its own subset of links.
- Easily scalable for small to medium websites.
- Encourages modularity when combined with separate parsing and analysis functions.

Version 0.2 of the crawler demonstrates how **recursion and state management** can be used to explore a network of web pages efficiently.

By integrating the analyze() function, a visited set, and exception handling, this program forms a foundation for more advanced tools such as search-engine spiders and data-collection bots.

18.4.3 Web Page Content Analysis

The web page analysis consists of computing (1) the frequency of every word in the web page content (i.e., in the text data) and (2) the list of links contained in the web page. We have already computed the list of links.

```
def analyze(url):
    '''prints the frequency of every word in web page url and
    prints and returns the list of http links, in absolute
    format, in it'''
    print('Visiting', url)  # for testing
    # obtain links in the web page
    content = urlopen(url).read().decode()
```

```
    collector = Collector(url)
    collector.feed(content)
    urls = collector.getLinks()  # get list of links

    # compute word frequencies
    content = collector.getData()  # get text data as a string
    freq = frequency(content)
    # print the frequency of every text data word in web page
    print('\n{:50} {:10} {:5}'.format('URL', 'word', 'count'))
    for word in freq:
        print('{:50} {:10} {:5}'.format(url, word, freq[word]))
    # print the http links found in web page
    print('\n{:50} {:10}'.format('URL', 'link'))
    for link in urls:
        print('{:50} {:10}'.format(url, link))
    return urls
```

Supporting Components

The analyze() function depends on two important helper classes/functions:

1. The Collector Class

The **Collector** is a subclass of HTMLParser.

It is designed to:

- Collect all hyperlinks (<a href="..."> tags).
- Collect visible text data for word frequency analysis.

```
from html.parser import HTMLParser
from urllib.parse import urljoin

class Collector(HTMLParser):
    'Collects text and links from a web page'

    def __init__(self, url):
        HTMLParser.__init__(self)
        self.url = url
        self.links = []
        self.data = []
```

```
    def handle_starttag(self, tag, attrs):
        if tag == 'a':
            for (attr, value) in attrs:
                if attr == 'href':
                    absolute = urljoin(self.url, value)
                    if absolute.startswith('http'):
                        self.links.append(absolute)

    def handle_data(self, data):
        self.data.append(data)

    def getLinks(self):
        return self.links

    def getData(self):
        return ' '.join(self.data)
```

2. The frequency() Function

The frequency() function computes the number of times each word occurs in the input text string.

It uses a **dictionary** to store the results, with each word as a key and its count as a value.

```
def frequency(text):
    'returns a dictionary with frequency of each word in text'
    freq = {}
    words = text.split()
    for w in words:
        w = w.lower()
        freq[w] = freq.get(w, 0) + 1
    return freq
```

Putting It All Together

Here is how these functions integrate into the **recursive crawler**:

```
visited = set()
def crawl2(url):
    '''Recursive crawler that analyzes and visits web pages'''
    global visited
```

```
    visited.add(url)
    links = analyze(url)
    for link in links:
        if link not in visited:
            try:
                crawl2(link)
            except:
                pass
```

Each time crawl2(url) visits a page, it calls analyze(url) to print the word frequencies and list of discovered links. Then it recursively crawls those links that have not yet been visited.

Sample Output (Textbook Example)

Visiting https://example.com/one.html

URL	word	count
https://example.com/one.html	beijing	3
https://example.com/one.html	paris	5
https://example.com/one.html	chicago	5

URL	link
https://example.com/one.html	https://example.com/two.html
https://example.com/one.html	https://example.com/three.html
https://example.com/one.html	https://example.com/four.html

Advantages of This Design

- Reusability: The analyze() function can be used independently to examine a single web page.
- Transparency: The printed tables clearly show how the crawler progresses and what it extracts.
- Extensibility: The frequency() function can easily be modified to filter stop words, compute relative frequencies, or export data.
- Scalability: The approach can be integrated into larger search-engine style crawlers with minimal changes.

18.5 SUMMARY

- The World Wide Web operates through client-server communication using the HTTP protocol.
- URLs identify resources, while HTML structures content.
- Python provides modules like urllib.request, urllib.parse, and html.parser for web automation and content retrieval.
- Regular expressions enable text pattern matching for data extraction.
- A web crawler automates navigation and analysis of web pages.

18.6 TECHNICAL TERMS

- HTTP
- URL
- HTML
- Parser
- Crawler

18.7 SELF-ASSESSMENT QUESTIONS

1. What are the main components of the World Wide Web?
2. Explain the structure and components of a URL.
3. Write a Python program to download and display HTML from a website.
4. What is the purpose of HTMLParser?
5. Define regular expressions and list common regex functions.
6. Explain how Python's urllib module handles HTTP requests.
7. Describe how to extract hyperlinks from HTML using Python.
8. Design a simple web crawler that follows links up to two levels deep.

18.8 SUGGESTED READINGS

- Ljubomir Perković, *Introduction to Computing Using Python: An Application Development Focus*, Wiley (2012).
- Mark Lutz, *Programming Python*, O'Reilly Media.
- David Beazley, *Python Essential Reference*, Addison-Wesley.
- Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly Media.

Mrs. Appikatla Pushpa Latha

LESSON- 19
STRING PATTERN MATCHING

AIMS AND OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the concept of pattern matching and its importance in text mining and data extraction.
- Explain the syntax and structure of regular expressions (regex).
- Identify and apply common regex operators and metacharacters.
- Use the Python re module for pattern-based searching, matching, replacing, and extracting data.
- Develop programs that extract information such as emails, URLs, dates, and numbers from text files and web content.
- Understand how regex enables data cleaning, validation, and web data mining.

STRUCTURE

19.1 INTRODUCTION TO STRING PATTERN MATCHING

To mine the text content of a web page or other text document, we need tools that help us define text patterns and then search for strings in the text that match these text patterns.When analyzing or mining text data (such as web pages, logs, or emails), it is often necessary to search for patterns instead of fixed words.

For example, you might need to:

- Find all email addresses in a document,
- Extract all URLs from a web page, or
- Identify all dates in a text file.
- To perform such tasks efficiently, we use regular expressions, or regex, which are patterns describing sets of strings.

19.2 NEED FOR TEXT MINING

Text mining is a process of automatically discovering patterns, relationships, or structures in textual data.

Using regular expressions, one can:

- Detect **specific structures** (e.g., phone numbers, postal codes).
- Filter **HTML content** and extract hyperlinks.
- Validate **user inputs** in web forms (e.g., email formats).
- Analyze **log files** or large corpora of text.
- Regular expressions are foundational tools in **data preprocessing** for Natural Language Processing (NLP) and **information retrieval**.

19.3 REGULAR EXPRESSIONS

Regular expressions provide a compact and flexible way to match text patterns. They use special symbols (called metacharacters) to describe text structures. Regular expressions (regex) are patterns that describe sets of strings and are widely used for searching and text manipulation.

19.3.1 Basic Concepts and Examples

The simplest regular expression is one that doesn't use any regular expression operators. For example, the regular expression best matches only one string, the string 'best':

Regular Expression	**Matching String(s)**
best	best

'be.t' matches best, but also 'belt', 'beet', 'be3t', and 'be!t', among others:

Regular Expression	**Matching String(s)**
be.t	best, belt, beet, bezt, be3t, be!t, be t, . . .

Pattern: b e . t
↓ ↓ ↓ ↓
String: b e s t → ✅ Match

b e l t → ✅ Match
b e e t → ✅ Match
b e ! t → ✅ Match
b e t → ✅ Match

Explanation:

- The dot (.) acts as a **wildcard**, matching **any single character**.
- Only strings that start with b and e, and end with t of length 4, will match.

For example, the operator * in regular expression be*t matches *0 or more* repetitions of the previous character (e). It therefore matches bt and also bet, beet, and so on:

Regular Expression	**Matching String(s)**
be*t	bt, bet, beet, beeet, beeeet, . . .
be+t	bet, beet, beeet, beeeet, . . .
bee?t	bet, beet

Example: be*t

Pattern: b e* t
↓ ↓ ↓
String: b t → ✅ (zero 'e')
b e t → ✅
b e e t → ✅
b e e e t → ✅

Flow of Matching:

Start → 'b' → zero or more 'e's → 't' → Match ✅

Explanation:

- The * operator repeats the previous character zero or more times.
- This pattern is **greedy**, meaning it will match as many 'e's as possible before moving to 't'.

Example: be+t

The pattern be+t requires **at least one 'e'** before 't'.

Pattern: b e+ t
↓ ↓ ↓
String: b t → ❌ (no 'e')
b e t → ✅
b e e t → ✅
b e e e t → ✅

Flow of Matching:

Start → 'b' → one or more 'e's → 't' → Match ✅

Example: bee?t

The pattern bee?t matches strings where 'b' is followed by one 'e' and **an optional 'e'** before 't'.

Pattern: b e e? t

String: b e t → ✅ (0 extra 'e')

b e e t → ✅ (1 extra 'e')

b e e e t → ❌ (too many 'e')

Explanation:

- The ? operator means the previous character may occur **zero or one time**.

For example, regular expression hello|Hello matches strings 'hello' and 'Hello':

Regular Expression	**Matching String(s)**
hello|Hello	hello, Hello.
a+|b+	a, b, aa, bb, aaa, bbb, aaaa, bbbb, . . .
ab+|ba+	ab, abb, abbb, . . . , and ba, baa, baaa, . . .

Example: Alternation — hello|Hello

Alternation (|) allows for **either-or** matching.

Pattern: hello | Hello

↓ ↓

String: hello → ✅

String: Hello → ✅

String: hELLO → ❌

Flow Diagram:

Start

├── "hello" → Match ✅

└── "Hello" → Match ✅

Example: Grouping — (ab)+

Grouping ( ) allows repetition of multiple characters as a unit.

Pattern: (ab)+

String: ab → ✅

String: abab → ✅

String: ababab → ✅

String: a → ❌

String: abb → ❌

Flow Diagram:

Start → 'a' → 'b' → repeat group (ab) → Match ✅

Example: Character Class [A-Za-z0-9_]

This matches any **alphanumeric character or underscore**.

Pattern: [A-Za-z0-9_]+

String: Hello123_ → ✅

String: Hi! → ❌ ('!' not in class)

Flow Diagram:

Start → Accept any A–Z, a–z, 0–9, or '_' → repeat (+) → Match ✅

Regular Expression	Matching Strings
best	best
be.t	best, belt, beet, be3t, be_t, be t
be*t	bt, bet, beet, beeet, beeeet
be+t	bet, beet, beeet, beeeet
bee?t	bet, beet
`hello	Hello`
`a+	b+`
`ab+	ba+`

Example — Find all links in a web page:

```
import re
html = '<a href="https://python.org">Python</a> <a href="/docs">Docs</a>'
links = re.findall(r'href="(.*?)"', html)
print(links)
```

Output:

['https://python.org', '/docs']

Example: Email Extraction Regex

Pattern used:

[\w\.-]+@[\w\.-]+

Components Breakdown:

Component	Meaning
[\w\.-]+	one or more word, dot, or hyphen characters
@	literal at-symbol
[\w\.-]+	domain name part

Example Text:

Contact: user1@mail.com, info@company.org

Matches:
user1@mail.com
info@company.org
Flow Diagram:
Start
→ [word/dot/hyphen]+
→ '@'
→ [word/dot/hyphen]+
→ Match ✅

19.3.2 Common Regex Operators

Operator	Interpretation
.	Matches any character except a new line character.
*	Matches 0 or more repetitions of the regular expression immediately preceding it. So in regular expression `ab*`, operator `*` matches 0 or more repetitions of `b`, not `ab`.
+	Matches 1 or more repetitions of the regular expression immediately preceding it.
?	Matches 0 or 1 repetitions of the regular expression immediately preceding it.
[]	Matches any character in the set of characters listed within the square brackets; a range of characters can be specified using the first and last character in the range and putting `'-'` in between.
^	If `S` is a set or range of characters, then `[^S]` matches any character *not* in `S`.
|	If A and B are regular expressions, A|B matches any string that is matched by A or B.

Fig 19.1 Some regular expression operator.

19.3.3 Character Classes and Quantifiers

Character classes help define sets of characters.

Common shorthand character classes include:

Symbol	Description
\d	Digit (0–9)
\D	Non-digit
\w	Word character (letters, digits, underscore)
\W	Non-word character
\s	Whitespace
\S	Non-whitespace

Example:

```
import re
pattern = r'\d+'
text = "My age is 25 and my pin code is 530003."
numbers = re.findall(pattern, text)
print(numbers)
```

Output:

```
['25', '530003']
```

19.4 PYTHON MODULE RE

Python's re module implements functions for regex pattern matching. It provides tools to search, extract, replace, and validate strings based on regular expressions.

19.4.1 Regex Matching Functions

Function	Description
re.match(pattern, string)	Checks if the pattern matches from the beginning of the string.
re.search(pattern, string)	Searches the entire string for the first occurrence of the pattern.
re.findall(pattern, string)	Returns a list of all non-overlapping matches.
re.sub(pattern, repl, string)	Substitutes all occurrences of a pattern with another string.
re.split(pattern, string)	Splits a string based on the pattern.

19.4.2 Examples and Applications

Example 1 — Find all links in a web page

```
import re
html = '<a href="https://python.org">Python</a> <a href="/docs">Docs</a>'
links = re.findall(r'href="(.*?)"', html)
print(links)
```

Output:

```
['https://python.org', '/docs']
```

Example 2 — Extract email addresses

```
text = "Contact: user1@mail.com, info@company.org"
emails = re.findall(r'[\w\.-]+@[\w\.-]+', text)
print(emails)
```

Output:

```
['user1@mail.com', 'info@company.org']
```

Example 3 — Replace all digits

```
import re
text = "Order numbers: 123, 456, 789"
```

```
new_text = re.sub(r'\d', 'X', text)
print(new_text)
```

Output:
Order numbers: XXX, XXX, XXX

Example 4 — Search for a pattern

```
sentence = "Python programming is powerful."
if re.search(r'program', sentence):
   print("Pattern found!")
```

Output:
Pattern found!

19.4.3 Using Regex for Data Extraction

Regular expressions can extract structured data from unstructured text.
Common regex functions:

Function	Description
re.match()	Matches from the beginning of a string.
re.search()	Finds the first occurrence of a pattern.
re.findall()	Returns all non-overlapping matches.
re.sub()	Substitutes one string for another.

Example — Extract Email Addresses:

```
text = "Contact: user1@mail.com, info@company.org"
emails = re.findall(r'[\w\.-]+@[\w\.-]+', text)
print(emails)
```

Output:

```
['user1@mail.com', 'info@company.org']
```

Example — Extract Dates

```
import re
text = "Meetings on 12-05-2024 and 25/10/2025."
dates = re.findall(r'\d{2}[-/]\d{2}[-/]\d{4}', text)
print(dates)
```

Output:

```
['12-05-2024', '25/10/2025']
```

Example — Validate Phone Numbers

```
numbers = ["+91-9876543210", "12345", "91-876543210"]
for num in numbers:
   if re.match(r'^\+?\d{1,2}-\d{10}$', num):
      print(num, "is valid.")
```

Output:

+91-9876543210 is valid.

19.5 CASE STUDY — EXTRACTING LINKS AND EMAILS

The following program combines regular expressions with file processing and text extraction.

```
import re
html_content = '''
<html><body>
<p>Contact: user1@mail.com</p>
<a href="https://example.com/home">Home</a>
<a href="https://example.com/about">About</a>
</body></html>
'''

emails = re.findall(r'[\w\.-]+@[\w\.-]+', html_content)
links = re.findall(r'href="(.*?)"', html_content)

print("Emails found:", emails)
print("Links found:", links)
```

Output:

Emails found: ['user1@mail.com']
Links found: ['https://example.com/home', 'https://example.com/about']

19.6 APPLICATIONS OF REGULAR EXPRESSIONS

Domain	Application
Web scraping	Extracting hyperlinks, image URLs, and metadata
Data cleaning	Removing unwanted symbols, HTML tags, or whitespace
Validation	Checking emails, phone numbers, IP addresses
Natural Language Processing	Tokenizing text, filtering stop words
Log analysis	Detecting error messages or IP addresses
Security	Finding SQL injection or suspicious input patterns

1. Web Scraping

Application: Extracting hyperlinks, image URLs, and metadata

Explanation:

Web scraping involves collecting data from websites. Since web pages are mostly in HTML, Regex can be used to extract specific patterns of text such as URLs, links, or metadata.

For example:

- Extracting hyperlinks:

- href="(https?://[^"]+)"

→ This pattern matches all hyperlinks beginning with http or https.

- Extracting image URLs:
- <img[^>]+src="([^">]+)"

→ Captures all image source (src) attributes from <img> tags.

- Extracting metadata (e.g., title, description):
- <meta\s+name="description"\s+content="([^"]+)"

→ Extracts content of meta description tags.

🔷 **Use case example:**

In web scraping scripts using Python (e.g., with requests and re), Regex helps filter out only the needed text from raw HTML before further processing or storing it in databases.

2. Data Cleaning

Application: Removing unwanted symbols, HTML tags, or whitespace

Explanation:
Data collected from the web or files often contains extra characters, symbols, or HTML code that must be cleaned before analysis. Regex helps identify and remove such unwanted patterns quickly.
Examples:

- Removing HTML tags:
- <[^>]+>

→ Removes everything between < and > (i.e., HTML tags).

- Removing special characters:
- [^a-zA-Z0-9\s]

→ Keeps only alphabets, digits, and spaces.

- Trimming extra whitespace:
- \s+

→ Matches multiple spaces or tabs; can be replaced with a single space.

🔷 **Use case example:**

Cleaning text data before feeding it to an NLP model or database ensures consistency and accuracy.

3. Validation

Application: Checking emails, phone numbers, IP addresses

Explanation:

Regex is widely used for input validation—to ensure data entered by users follows the correct format.

Examples:

- Email validation:
- ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$

→ Matches most standard email formats.

- Phone number validation (India):
- ^(\+91[\-\s]?)?[6-9]\d{9}$

→ Matches Indian mobile numbers with or without country code.

- IP address validation:
- ^((25[0-5]|2[0-4]\d|[0-1]?\d{1,2})\.){3}(25[0-5]|2[0-4]\d|[0-1]?\d{1,2})$

→ Ensures valid IPv4 format.

🔷 **Use case example:**

Used in web forms or backend systems to reject invalid entries before saving them to databases.

4. Natural Language Processing (NLP)

Application: Tokenizing text, filtering stop words

Explanation:

Regex helps process and analyze textual data efficiently in NLP.

Examples:

- Tokenization (splitting text into words):
- \w+

→ Extracts words from text while ignoring punctuation.

- **Filtering stop words:**
- You can use Regex to remove common words like “the”, “is”, “at”, etc., using patterns such as:
    - \b(the|is|in|at|which|on)\b

→ Removes listed stop words.

- Identifying specific word patterns:
- For instance, finding all hashtags or mentions in tweets:
- #\w+ or @\w+

🔷 **Use case example:**

In preprocessing pipelines of NLP tasks such as sentiment analysis or text classification.

5. Log Analysis

Application: Detecting error messages or IP addresses

Explanation:

System and application logs contain large volumes of text data. Regex enables automatic pattern matching to detect key information.

Examples:

- Extracting IP addresses:
- \b\d{1,3}(\.\d{1,3}){3}\b

→ Finds all IPv4 addresses.

- Finding error messages:
- ERROR|FATAL|EXCEPTION

→ Detects critical log entries.

- Extracting timestamps:
- \d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}

→ Captures datetime formats (e.g., 2025-10-29 20:15:30).

🔷 **Use case example:**

Used in server monitoring or debugging tools to locate issues from log files quickly.

6. Security

Application: Finding SQL injection or suspicious input patterns

Explanation:

Regex can detect potentially harmful user inputs or code injection attempts in web applications.

Examples:

- Detecting SQL injection attempts:
- (?:')|(?:--)|(/*(?:.|[\n\r])*?*/)|(\b(select|update|delete|insert|drop|exec)\b)

→ Identifies suspicious SQL keywords or comment patterns.

- Identifying cross-site scripting (XSS) attempts:
- <script.*?>.*?</script>

→ Matches embedded JavaScript code in inputs.

- Filtering special characters:
- [<>'"%;()&+]

→ Detects characters that can be part of malicious payloads.

🔷 **Use case example:**

Used in web application firewalls (WAFs), input sanitization, and log-based intrusion detection systems.

19.7 SUMMARY

- String pattern matching enables flexible search and extraction from text.
- Regular expressions (regex) are symbolic patterns that describe sets of strings.
- The re module provides Python tools for searching, matching, and replacing patterns.
- Common operators like ., *, +, ?, and | allow powerful pattern combinations.
- Regex is vital for web mining, data validation, and text processing.

19.8 TECHNICAL TERMS

- Regex (Regular Expression)
- Metacharacter
- Character Class
- Quantifier
- Capture Group
- Escaping

19.9 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Define regular expressions. Explain their importance in text mining.
2. Discuss various regex operators with suitable examples.
3. Describe how the re module supports pattern matching and substitution.
4. Explain the difference between re.match(), re.search(), and re.findall().

Short Notes

1. Write a note on character classes.
2. How are grouping and alternation used in regex?
3. Explain the use of re.sub() with an example.

19.10 SUGGESTED READINGS

1. Ljubomir Perković, *Introduction to Computing Using Python: An Application Development Focus*, Wiley, 2012.
2. Mark Lutz, *Learning Python*, 5th Edition, O'Reilly, 2013.
3. Al Sweigart, *Automate the Boring Stuff with Python*, No Starch Press, 2020.
4. Python Official Documentation — https://docs.python.org/3/library/re.html

Mrs. Appikatla Pushpa Latha

LESSON- 20

DATABASE PROGRAMMING IN PYTHON

AIMS AND OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the concept of databases and their importance in software applications.
- Explain the structure and operations of relational databases using SQL.
- Identify the role of tables, keys, and relationships in organizing data.
- Use SQL commands such as SELECT, INSERT, UPDATE, and DELETE to manipulate data.
- Apply aggregate functions and grouping to summarize data.
- Develop Python programs that interact with databases using the sqlite3 module.
- Create, query, and modify databases directly from Python scripts.
- Manage database transactions, handle exceptions, and ensure data consistency.
- Integrate SQL queries into Python-based data analysis and application workflows.
- Appreciate the role of databases in real-world applications such as data storage, reporting, and analytics.

STRUCTURE

20.1 DATABASES AND SQL

20.1.1 Introduction to Databases and SQL

A database is an organized collection of data designed to store, manage, and retrieve information efficiently. Modern software systems rely heavily on databases to maintain user records, transaction histories, inventory details, and analytical data.

A Database Management System (DBMS) provides the interface to define, create, manipulate, and maintain databases. Among different types of DBMSs, the Relational Database Management System (RDBMS) is the most widely used. It stores data in the form of tables, which can be related through keys.

The Structured Query Language (SQL) is the standard language for interacting with RDBMSs. SQL allows users to:

- Create and modify database structures.
- Insert, update, delete, and retrieve data.
- Control access and manage transactions.

20.1.2 Database Tables

A table is the fundamental structure in a relational database. It consists of:

- Rows (records) – represent individual entities.
- Columns (fields) – represent attributes of those entities.

Example: STUDENT Table

RollNo	Name	Branch	Marks
101	Anjali	CSE	87
102	Ramesh	ECE	75
103	Kavya	CSE	92

Key Concepts:

- Primary Key: uniquely identifies each row (e.g., RollNo)
- Foreign Key: links one table to another (e.g., RollNo in COURSE table)
- Constraints: ensure data integrity (e.g., NOT NULL, UNIQUE)

20.1.3 Structured Query Language (SQL) Overview

SQL consists of multiple categories of commands:

Category	Description	Examples
DDL (Data Definition Language)	Defines and modifies database schema	CREATE, ALTER, DROP
DML (Data Manipulation Language)	Manages data in tables	INSERT, UPDATE, DELETE
DQL (Data Query Language)	Retrieves data	SELECT
DCL (Data Control Language)	Grants/revokes permissions	GRANT, REVOKE
TCL (Transaction Control Language)	Manages transactions	COMMIT, ROLLBACK

20.1.4 Statement SELECT

The SELECT statement retrieves data from tables.
Syntax:

```
SELECT column_list
FROM table_name
[WHERE condition]
[ORDER BY column_name [ASC|DESC]];
```

Example:

```
SELECT Name, Marks
FROM Student
WHERE Branch = 'CSE'
ORDER BY Marks DESC;
```

Result:

Name	Marks
Kavya	**92**
Anjali	**87**

20.1.5 Clause WHERE

Filters records based on a condition.
Examples:

```
SELECT * FROM Student WHERE Marks > 80;
SELECT * FROM Student WHERE Branch = 'CSE' AND Marks > 85;
SELECT * FROM Student WHERE Name LIKE 'A%';
```

Operators Used:

- Comparison: =, >, <, >=, <=, <>
- Logical: AND, OR, NOT
- Pattern Matching: LIKE, IN, BETWEEN
- NULL Checking: IS NULL, IS NOT NULL

20.1.6 Built-in SQL Functions

SQL includes aggregate and scalar functions.

Type	Function	Example
Aggregate	COUNT()	SELECT COUNT(*) FROM Student;
	AVG()	SELECT AVG(Marks) FROM Student;
	MAX(), MIN(), SUM()	
Scalar	UPPER(Name)	Converts text to uppercase
	LENGTH(Name)	Returns string length
	ROUND(Marks, 2)	Rounds numbers

20.1.7 Clause GROUP BY and HAVING

Used for grouping and filtering aggregate results.

```
SELECT Branch, AVG(Marks) AS AvgMarks
FROM Student
GROUP BY Branch
HAVING AVG(Marks) > 80;
```

Output:

Branch	AvgMarks
CSE	**89.5**

20.1.8 SQL Queries Involving Multiple Tables

JOIN operations combine data from multiple tables.

Example:

```
SELECT Student.Name, Course.CourseName
FROM Student
JOIN Course
ON Student.RollNo = Course.RollNo;
```

JOIN Types:

- INNER JOIN → Only matching rows
- LEFT JOIN → All from left table + matches
- RIGHT JOIN → All from right table + matches
- FULL OUTER JOIN → All rows from both sides (where supported)

20.1.9 Statement CREATE TABLE

Defines the structure of a table.

```
CREATE TABLE Employee (
   EmpID INTEGER PRIMARY KEY,
   Name TEXT NOT NULL,
   Department TEXT,
   Salary REAL CHECK (Salary > 0)
);
```

20.1.10 Statements INSERT, UPDATE, and DELETE

INSERT:

```
INSERT INTO Employee (EmpID, Name, Department, Salary)
VALUES (101, 'Meena', 'HR', 45000);
```

UPDATE:

```
UPDATE Employee
SET Salary = Salary + 5000
WHERE Department = 'IT';
```

DELETE:

```
DELETE FROM Employee WHERE EmpID = 101;
```

20.2 DATABASE PROGRAMMING IN PYTHON

20.2.1 Introduction

Python supports database operations through the DB-API (PEP 249) interface. It provides a consistent way to connect to, query, and manage relational databases.

20.2.2 Database Engines and SQLite

SQLite is a built-in, lightweight, file-based relational database included with Python. It is ideal for learning and small applications.

```
import sqlite3
conn = sqlite3.connect('university.db')
```

20.2.3 Creating a Database using sqlite3

```
import sqlite3
conn = sqlite3.connect('student.db')
cur = conn.cursor()
cur.execute('''CREATE TABLE IF NOT EXISTS Student (
            RollNo INTEGER PRIMARY KEY,
            Name TEXT,
            Branch TEXT,
            Marks INTEGER)''')

conn.commit()
conn.close()
```

20.2.4 Executing SQL Queries from Python

```
cur.execute("INSERT INTO Student VALUES (101, 'Anjali', 'CSE', 87)")
cur.execute("INSERT INTO Student VALUES (102, 'Ramesh', 'ECE', 75)")
conn.commit()
```

20.2.5 Fetching and Displaying Data

```
cur.execute("SELECT * FROM Student")
for row in cur.fetchall():
   print(row)
```

Output:

```
(101, 'Anjali', 'CSE', 87)
(102, 'Ramesh', 'ECE', 75)
```

20.2.6 Parameterized Queries

Prevents SQL injection:

```
cur.execute("SELECT * FROM Student WHERE Name=?", ('Anjali',))
```

20.2.7 Committing Transactions and Closing Connections

```
conn.commit()
```

```
conn.close()
```

20.2.8 Exception Handling

```
try:
   conn = sqlite3.connect('student.db')
   cur = conn.cursor()
   cur.execute("INSERT INTO Student VALUES (104, 'Meena', 'IT', 88)")
   conn.commit()
except sqlite3.Error as e:
   print("Error:", e)
finally:
   conn.close()
```

20.2.9 Using with Statement

```
with sqlite3.connect('student.db') as conn:
   cur = conn.cursor()
   cur.execute("SELECT * FROM Student")
   print(cur.fetchall())
```

20.2.10 Integrating SQL with Python Data Structures

Using Pandas for analysis:

```
import pandas as pd
conn = sqlite3.connect('student.db')
df = pd.read_sql_query("SELECT * FROM Student", conn)
print(df)
```

20.2.11 Example CRUD Application

(Full working Python example — Create, Read, Update, Delete)

```
import sqlite3

def create_table():
   with sqlite3.connect('college.db') as conn:
      conn.execute('''CREATE TABLE IF NOT EXISTS Student
                  (RollNo INTEGER PRIMARY KEY, Name TEXT, Branch TEXT, Marks
INTEGER)''')

def insert_student(r, n, b, m):
   with sqlite3.connect('college.db') as conn:
      conn.execute("INSERT INTO Student VALUES (?, ?, ?, ?)", (r, n, b, m))

def view_students():
   with sqlite3.connect('college.db') as conn:
      for row in conn.execute("SELECT * FROM Student"):
         print(row)
```

```
create_table()
insert_student(1, 'Anjali', 'CSE', 89)
insert_student(2, 'Ramesh', 'ECE', 78)
view_students()
```

20.3 FUNCTIONAL LANGUAGE APPROACH

20.3.1 List Comprehension and Functional Constructs

Functional programming emphasizes writing programs using expressions rather than statements.

Python supports several functional programming concepts such as list comprehensions, map(), filter(), reduce(), and lambda functions.

List Comprehensions

A list comprehension provides a concise way to create lists.

Syntax:

```
[expression for item in iterable if condition]
```

Example 1:

```
squares = [x*x for x in range(1, 6)]
print(squares)
```

Output:

```
[1, 4, 9, 16, 25]
```

Example 2: Filtering

```
even_numbers = [x for x in range(10) if x % 2 == 0]
print(even_numbers)
```

Output:

```
[0, 2, 4, 6, 8]
```

List comprehensions are faster and more readable than traditional loops.

Functional Constructs in Python

1. **lambda Function:**
   An anonymous one-line function.
   ```
   square = lambda x: x * x
   print(square(5))
   ```

2. **map() Function:**
   Applies a function to all items in an iterable.
   ```
   numbers = [1, 2, 3, 4]
   result = list(map(lambda x: x**2, numbers))
   print(result)
   Output: [1, 4, 9, 16]
   ```

3. **filter() Function:**
   Selects elements that satisfy a condition.
   evens = list(filter(lambda x: x % 2 == 0, numbers))
   print(evens)
   Output: [2, 4]

4. **reduce() Function (from functools):**
   Combines items cumulatively.
   from functools import reduce
   total = reduce(lambda x, y: x + y, numbers)
   print(total)
   Output: 10

20.3.2 MapReduce Problem Solving Framework

MapReduce is a programming paradigm for processing large datasets in parallel across multiple systems. It divides computation into two major phases:

1. **Map Phase:**
   Each record (key–value pair) is processed independently to produce intermediate results.
2. **Reduce Phase:**
   The intermediate results are aggregated or combined to produce final output.

This model is the foundation of frameworks like Hadoop and Spark.

Example: Word Count Using MapReduce

```
from functools import reduce

text = "Python supports functional programming using map and reduce"
words = text.split()

# Map step: Create pairs
mapped = [(w, 1) for w in words]

# Reduce step: Count occurrences
def reducer(acc, item):
    word, count = item
    acc[word] = acc.get(word, 0) + count
    return acc

word_count = reduce(reducer, mapped, {})
print(word_count)
```

Output:

```
{'Python': 1, 'supports': 1, 'functional': 1, 'programming': 1,
 'using': 1, 'map': 1, 'and': 1, 'reduce': 1}
```

20.3.3 MapReduce in the Abstract

In general, MapReduce operates on a collection of records:

Phase	Input	Function	Output
Map	(k_1, v_1)	map()	list of (k_2, v_2)
Shuffle	(k_2, v_2)	group by key	$(k_2, [v_{21}, v_{22}, \ldots])$
Reduce	$(k_2, \text{list}(v_2))$	reduce()	(k_3, v_3)

Conceptual Example:

```
data = [("A", 3), ("B", 5), ("A", 2), ("B", 7)]
# Map phase - already done
# Shuffle phase
shuffled = {}
for k, v in data:
    shuffled.setdefault(k, []).append(v)

# Reduce phase
reduced = {k: sum(vs) for k, vs in shuffled.items()}
print(reduced)
```

Output:

```
{'A': 5, 'B': 12}
```

20.3.4 Inverted Index Example

An inverted index maps words to the documents in which they appear — a key concept in search engines.

Example:

```
documents = {
    "doc1": "python supports functional programming",
    "doc2": "functional programming enables mapreduce",
    "doc3": "python and mapreduce are powerful"
}

# Map Phase
mapped = []
for doc, text in documents.items():
    for word in text.split():
        mapped.append((word.lower(), doc))
```

```
# Shuffle Phase
index = {}
for word, doc in mapped:
    index.setdefault(word, set()).add(doc)


# Display Inverted Index
for word, docs in index.items():
    print(word, ":", docs)
```

Output:

python : {'doc1', 'doc3'}

functional : {'doc1', 'doc2'}

mapreduce : {'doc2', 'doc3'}

This is the foundation of information retrieval systems like Google Search.

20.4 PARALLEL COMPUTING

20.4.1 Introduction to Parallel Computing Concepts

Parallel computing refers to performing multiple computations simultaneously.
It enhances performance by utilizing multiple CPU cores or processors.

Python supports parallelism through:

- The threading module (lightweight concurrency)
- The multiprocessing module (true parallelism with separate processes)

Advantages:

- Faster computation for large datasets
- Better utilization of CPU resources
- Useful in AI, image processing, and simulations

20.4.2 Class Pool of the multiprocessing Module

Python's multiprocessing.Pool provides an easy interface for distributing work among multiple processes.

Example:

```
from multiprocessing import Pool

def square(n):
    return n * n

if __name__ == "__main__":
    with Pool(4) as p:
        result = p.map(square, [1, 2, 3, 4, 5])
```

```
    print(result)
```

Output:

[1, 4, 9, 16, 25]

Each process handles one element in parallel.

20.4.3 Parallel Speedup and Performance

Speedup (S):

$$S = \frac{T_{sequential}}{T_{parallel}}$$

where $T_{sequential}$ = time for one core,

and $T_{parallel}$ = time with multiple cores.

Efficiency (E):

$$E = \frac{S}{N}$$

where N = number of processors.

Example:

If a task takes 10 seconds sequentially and 3 seconds on 4 cores:

$$S = \frac{10}{3} = 3.33, E = \frac{3.33}{4} = 0.83$$

Thus, efficiency = 83%.

20.4.4 Parallel MapReduce Implementation

Parallel MapReduce distributes both mapping and reducing tasks across processors.

Example: Word Count (Parallel)

```
from multiprocessing import Pool
from functools import reduce

text = ["python supports mapreduce",
        "mapreduce enables parallel computing",
        "parallel mapreduce in python"]

def map_func(line):
    return [(w, 1) for w in line.split()]

def reduce_func(acc, item):
```

```
    word, count = item
    acc[word] = acc.get(word, 0) + count
    return acc

if __name__ == "__main__":
    with Pool(3) as p:
        mapped = p.map(map_func, text)

    # Flatten and reduce
    pairs = [pair for sublist in mapped for pair in sublist]
    word_count = reduce(reduce_func, pairs, {})
    print(word_count)
```

Output:

```
{'python': 2, 'supports': 1, 'mapreduce': 3,
 'enables': 1, 'parallel': 2, 'computing': 1, 'in': 1}
```

20.4.5 Comparing Parallel and Sequential MapReduce

Aspect	Sequential	Parallel
Execution	**One processor**	**Multiple processors**
Speed	**Slower**	**Faster**
Resource Utilization	**Single core**	**Multi-core**
Complexity	**Simple**	**Requires synchronization**
Use Case	**Small data**	**Big data / high computation**

20.5 SUMMARY

- Functional programming emphasizes expressions and immutability.
- List comprehensions and lambda functions make Python concise and powerful.
- MapReduce divides computation into mapping and reducing phases for large data processing.
- Parallel computing executes tasks concurrently using multiple processors.
- The multiprocessing module enables scalable, high-performance Python programs.

20.6 TECHNICAL TERMS

- Functional Programming
- List Comprehension
- Lambda Function
- MapReduce
- Pool
- Speedup
- Efficiency

20.7 SELF-ASSESSMENT QUESTIONS

Short Questions

1. Define **functional programming**.
2. Write any **two examples of list comprehensions** in Python.
3. Differentiate between the **map()** and **filter()** functions.
4. What is the purpose of the **reduce()** function?
5. Explain the two main phases of the **MapReduce** framework.
6. State any **two advantages** of parallel computing.
7. Write the **formula for Speedup and Efficiency** in parallel computing.
8. What is the role of the **multiprocessing.Pool** class in Python?
9. Mention **any two real-world applications** of parallel computing.
10. List any **two differences between sequential and parallel MapReduce** execution.

Essay Questions

1. Explain the concept of **functional programming** and describe how it is supported in Python with examples.
2. Discuss **list comprehensions** in Python and illustrate their use with appropriate examples.
3. Compare and contrast the working of **map()**, **filter()**, and **reduce()** functions with code illustrations.
4. Describe the **MapReduce problem-solving framework**. How does it simplify data processing in distributed environments?
5. Develop a **Python program** that counts the frequency of words in a given text using the MapReduce approach.
6. What is **parallel computing**? Explain its importance in high-performance and data-intensive applications.
7. Write a **Python program using multiprocessing.Pool** to compute the cube of numbers in parallel.
8. Define **Speedup** and **Efficiency** in parallel systems. Illustrate how they are used to measure performance improvement.
9. Compare **sequential and parallel MapReduce** executions in terms of processing, resource usage, and efficiency.
10. Discuss the **applications of parallel computing** in real-world domains such as machine learning, data mining, and simulations.

20.8 SUGGESTED READINGS

1. Mark Lutz, *Learning Python*, O'Reilly Media.
2. Allen B. Downey, *Think Python*, O'Reilly Media.
3. Ramez Elmasri & Shamkant Navathe, *Fundamentals of Database Systems*, Pearson.
4. Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Google Research Paper.
5. Python Official Documentation: https://docs.python.org/3/library/multiprocessing.html

Mrs. Appikatla Pushpa Latha