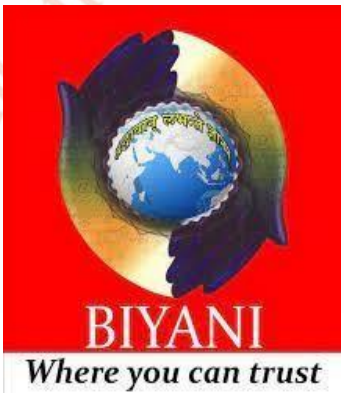
Biyani's Think Tank

Concept based notes

Python Programming

(BCA Semester V)

Smriti Verma
Asst. Professor
Deptt. of Information Technology
Biyani Girls College, Jaipur



Published by :
Think Tanks
Biyani Group of Colleges

Sector-3, Vidhyadhar Nagar,
Jaipur-302 023 (Rajasthan)
Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007
E-mail : acad@biyanicolleges.org
Website :www.gurukpo.com; www.biyanicolleges.org

First Edition: 2025

Leaser Type Setted by :

Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the "Teach Yourself" style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, Chairman & Dr. Sanjay Biyani, Director (Acad.) Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this Endeavour. They played an active role in coordinating the various stages of this Endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Author

Syllabus
BCA Semester V
Python Programming

Unit – I: Python Concepts: Origin, Comparison, Comments, Variables and Assignment, Identifiers, Basic Style Guidelines, Standard Types, Internal Types, Operators, Built-in Functions, Numbers and Strings. Sequences: Strings, Sequences, String-Operators & functions, Special Features of Strings, Memory Management, programs & examples.

Conditionals and Loops: if statement, else Statement, elif Statement, while Statement, for Statement, break Statement, continue Statement, pass Statement, else Statement

Unit – II: Object and Classes: Classes in Python, Principles of Object Orientation, Creating Classes, Instance Methods, Class variables, Inheritance, Polymorphism, Type Identification, Python libraries(Strings, Data structures & algorithms).

Lists and Sets: Built-in Functions, List type built in Methods, Tuples, Tuple Operators, Special Features of Tuples, Set: Introduction, Accessing, Built-in Methods (Add, Update, Clear, Copy, Discard, Remove), Operations (Union, Intersection, Difference).

Unit-III: Dictionaries : Introduction to Dictionaries, Built-in Functions, Built-in Methods, Dictionary Keys, Sorting and Looping, Nested Dictionaries.

Files: File Objects, File Built-in Function, File Built-in Methods, File Built-in Attributes, Standard Files, Command-line Arguments, File System, File Execution, Persistent Storage Modules.

Regular Expression: Regular Expression: Introduction/Motivation, Special Symbols and Characters for REs, REs and Python.

Unit – IV: Excetiptons: Concepts of Exceptions, Exceptions in Python, Detecting and Handling Exceptions, Exceptions as Strings, Raising Exceptions, Assertions, Standard Exceptions.

Database Interaction : SQL Database Connection using Python, Creating and Searching Tables, Reading and storing config information on database, Programming using database connections, Python Multithreading: Understanding threads, Forking threads, synchronizing the threads, Programming using multithreading.

Table of Contents

Unit I: Python Concepts

Sequences

Conditionals and Loops

Unit II: Objects and Classes

Lists and Sets

2.9 Built-in Functions for Lists
2.10 List Type Built-in Methods
2.11 Tuples and Their Features
2.12 Tuple Operators
2.13 Set: Introduction and Accessing Elements
2.14 Set Methods (Add, Update, Clear, Copy, Discard, Remove)
2.15 Set Operations (Union, Intersection, Difference)

Unit III: Dictionaries, Files, and Regular Expressions

Dictionaries

3.1 Introduction to Dictionaries
3.2 Built-in Functions for Dictionaries
3.3 Dictionary Methods
3.4 Dictionary Keys and Access
3.5 Sorting and Looping in Dictionaries
3.6 Nested Dictionaries

Files

3.7 File Objects and File Modes
3.8 File Built-in Functions
3.9 File Methods and Attributes
3.10 Standard Files
3.11 Command-line Arguments
3.12 File System and Execution
3.13 Persistent Storage Modules

Regular Expressions

3.14 Introduction and Motivation
3.15 Special Symbols and Characters in REs
3.16 Using Regular Expressions in Python

Unit IV: Exceptions, Database Interaction, and Multithreading

Exceptions

4.1 Concept of Exceptions
4.2 Handling Exceptions in Python
4.3 Exception Objects and Strings
4.4 Raising Exceptions

4.5 Assertions
4.6 Standard Exceptions

Database Interaction

4.7 SQL Database Connection using Python
4.8 Creating and Searching Tables
4.9 Reading & Storing Configuration Info
4.10 Programming with Database Connections

Multithreading

4.11 Understanding Threads in Python
4.12 Forking Threads
4.13 Synchronizing Threads
4.14 Programming Using Multithreading

Recommended Exercise

Sample Paper (Internal)

Chapter 1
Introduction To Python

1.1 Origin of Python

Q: What is the origin of Python, and how did it evolve?

A:
Python is a **high-level, interpreted programming language** that was created by **Guido van Rossum** and first released in **1991**. The language was designed with an emphasis on **code readability**, allowing programmers to express concepts in fewer lines of code compared to other programming languages like C and Java.

Python's design philosophy stresses the importance of simplicity and flexibility. Van Rossum was influenced by languages like **ABC**, **Modula-3**, and **C**, but Python's key feature became its emphasis on being **easy to learn and use**, making it an ideal language for both beginners and experienced developers. Python has grown in popularity due to its versatility, being used in fields such as **web development**, **data science**, **automation**, and **artificial intelligence**.

1.2 Comparison with Other Languages

Q: How does Python compare with other programming languages, such as Java, C++, and JavaScript?

A:
Python is often compared to languages like **Java**, **C++**, and **JavaScript**, and each of these languages has its strengths and weaknesses. Below is a comparative overview:

- **Python vs. Java**:
  - Python is **interpreted**, while Java is **compiled** into bytecode. This makes Python more flexible and easier to run across different systems without compilation, but Java's bytecode can lead to more optimized performance in large applications.
  - Python's syntax is much more concise and easier to read, while Java is more verbose and requires the definition of a class and a `main()` method for execution.
  - Python is generally **slower** than Java due to its dynamic typing and interpreted nature, but it is better suited for rapid prototyping and ease of use.
- **Python vs. C++**:
  - C++ is a **low-level language** with manual memory management, while Python abstracts away memory management using its built-in **garbage collector**, making Python easier to work with.
  - Python has **dynamic typing**, whereas C++ uses **static typing**, making Python more flexible but at the cost of some performance.

  - C++ is preferred in performance-critical applications, such as game development or systems programming, while Python is used in a wide variety of applications, from data science to web development.
- **Python vs. JavaScript**:
  - JavaScript is primarily used for **client-side web development**, while Python is used for **server-side** development (though frameworks like **Node.js** have made JavaScript more widely used server-side).
  - JavaScript is more focused on handling **asynchronous programming** with event-driven architecture, whereas Python's emphasis is on simplicity and readability.
  - Python's ecosystem for data analysis and machine learning (e.g., **Pandas**, **NumPy**, **TensorFlow**) is stronger than JavaScript's.

1.3 Comments in Python

Q: How are comments used in Python, and why are they important?

A:
Comments in Python are used to provide **explanations** or **annotations** within the code that are not executed by the Python interpreter. Comments improve code readability and help developers understand the purpose and function of code, especially when the codebase becomes large or when multiple developers are involved.

In Python, there are two types of comments:

- **Single-line comments**: They begin with a hash symbol (`#`) and are used for brief explanations or notes.

```
# This is a single-line comment
x = 5  # Assign 5 to x
```

- **Multi-line comments**: Python does not have a dedicated syntax for multi-line comments, but you can use a multi-line string (triple quotes) as a comment.

```
'''
This is a multi-line comment
that spans more than one line.
'''
```

Comments are critical for **documentation** and ensuring that others (or even yourself in the future) can understand and maintain the code. Python also provides **docstrings**, which are multi-line comments used to document functions, classes, and modules.

1.4 Variables and Assignment

Q: What are variables in Python, and how does assignment work in Python?

A:
A **variable** in Python is a symbolic name that refers to a value. Variables are used to store data that can be accessed and manipulated throughout the program. Python is a **dynamically typed language**, meaning you do not need to declare the data type of a variable before using it.

Assignment in Python is done using the **=** operator. When you assign a value to a variable, Python automatically determines the data type based on the value assigned.

Example:

```
x = 10  # Integer
y = "Hello"  # String
z = 3.14  # Float
```

Variables in Python are not bound to a specific type, so you can reassign a variable to a different type:

```
x = 10  # Integer
x = "Now I am a string"  # Reassigned to a string
```

1.5 Identifiers and Naming Conventions

Q: What are identifiers in Python, and what are the rules for naming them?

A:
An **identifier** in Python is a name used to identify variables, functions, classes, modules, or other objects. Python has several rules and conventions for naming identifiers:

- An identifier must begin with a **letter** (a-z, A-Z) or an **underscore** (_).
- The rest of the identifier can consist of **letters**, **digits** (0-9), and underscores.
- Identifiers cannot be **keywords** (reserved words in Python like `if`, `else`, `for`, etc.).
- Python is **case-sensitive**, meaning `variable`, `Variable`, and `VARIABLE` are all different identifiers.

Some naming conventions are also recommended for readability:

- **Lowercase** for variables and functions: `my_variable`, `calculate_area()`
- **CamelCase** for class names: `MyClass`, `EmployeeDetails`
- Use **underscores** to separate words in identifiers (snake_case).

1.6 Basic Style Guidelines

Q: What are the basic style guidelines for writing Python code?

A:
Python follows the **PEP 8** style guide, which provides conventions for writing clean, readable, and consistent Python code. Some key guidelines include:

- **Indentation**: Use **4 spaces per indentation level**. Avoid using tabs.
- **Line length**: Limit all lines to a maximum of **79 characters** to ensure that code is readable in various editors and on different screens.
- **Blank lines**: Use blank lines to separate functions, classes, and blocks of code within a function. Two blank lines should be used to separate top-level functions and classes.
- **Naming conventions**: Follow standard conventions like using **snake_case** for functions and variables, and **CamelCase** for classes.
- **Imports**: Imports should be on separate lines and appear at the top of the file, with standard library imports first, followed by third-party imports, and then application-specific imports.
- **Comments**: Include **docstrings** for all public modules, functions, classes, and methods. Use inline comments sparingly and only when necessary.

1.7 Standard Data Types

Q: What are the standard data types in Python?

A:
Python has several **built-in data types** that are essential for storing and manipulating data:

1. **Numeric types**:
   - **int**: Integer values (e.g., `42`)
   - **float**: Floating-point numbers (e.g., `3.14`)
   - **complex**: Complex numbers (e.g., `3 + 4j`)
2. **Text Type**:
   - **str**: Strings, which are sequences of characters (e.g., `"Hello, world!"`)
3. **Sequence Types**:
   - **list**: An ordered, mutable collection (e.g., `[1, 2, 3]`)
   - **tuple**: An ordered, immutable collection (e.g., `(1, 2, 3)`)
   - **range**: An immutable sequence of numbers, often used in loops (e.g., `range(10)`)
4. **Mapping Type**:
   - **dict**: A collection of key-value pairs (e.g., `{"name": "Alice", "age": 30}`)
5. **Set Types**:
   - **set**: An unordered collection of unique elements (e.g., `{1, 2, 3}`)
   - **frozenset**: An immutable set
6. **Boolean Type**:
   - **bool**: Represents truth values, either `True` or `False`
7. **Binary Types**:
   - **bytes**: Immutable sequences of bytes (e.g., `b"hello"`)
   - **bytearray**: Mutable sequences of bytes
   - **memoryview**: A view object that exposes an array's buffer interface

1.8 Internal Types

Q: What are internal types in Python, and how do they work?

A:
Internal types refer to the **built-in data types** in Python that are implemented under the hood. These types include **lists**, **tuples**, **strings**, **dictionaries**, and others. Python's **internal memory model** allows dynamic allocation and garbage collection for objects. This is why Python is considered a high-level language, abstracting away memory management details like pointers and manual allocation that low-level languages like C or C++ require.

Internally, these types are implemented as **C objects** in CPython (the standard Python implementation). Understanding how these internal types work can be important when working with large datasets or optimizing performance.

1.9 Operators in Python

Q: What are the different types of operators in Python?

A:
Python supports a wide range of **operators** that allow you to perform different operations on variables and values. These include:

1. **Arithmetic Operators**: For performing basic mathematical operations:
   - `+`, `-`, `*`, `/`, `//`, `%`, `**`
2. **Comparison Operators**: Used to compare values:
   - `==`, `!=`, `>`, `<`, `>=`, `<=`
3. **Logical Operators**: Used to perform logical operations:
   - `and`, `or`, `not`
4. **Assignment Operators**: Used to assign values to variables:
   - `=`, `+=`, `-=`, `*=`, `/=`, etc.
5. **Bitwise Operators**: For working with binary representations of integers:
   - `&`, `|`, `^`, `<<`, `>>`, `~`
6. **Membership Operators**: Used to check if a value is present in a sequence:
   - `in`, `not in`
7. **Identity Operators**: Used to check if two variables point to the same object:
   - `is`, `is not`

1.10 Built-in Functions

Q: What are some common built-in functions in Python?

A:
Python provides a wide array of **built-in functions** that can be used without importing any modules. Some of the most commonly used ones include:

- `print()`: Outputs data to the console.
- `len()`: Returns the length of an object (e.g., list, string).
- `type()`: Returns the type of an object.
- `int()`, `float()`, `str()`: Type conversion functions.
- `range()`: Generates a sequence of numbers.
- `sum()`: Returns the sum of an iterable.
- `max()`, `min()`: Returns the maximum or minimum value from an iterable.

These functions are essential in Python and help simplify many programming tasks.

1.11 Numbers and Strings

Q: How does Python handle numbers and strings, and what operations can you perform on them?

A:
In Python:

- **Numbers**: Python supports different numeric types: `int` (for whole numbers), `float` (for floating-point numbers), and `complex` (for complex numbers). Operations like addition, subtraction, multiplication, and division can be performed using arithmetic operators. Python also allows implicit type conversion, for example, adding an integer and a float will result in a float.
- **Strings**: Strings in Python are sequences of characters enclosed in single (`'`) or double (`"`) quotes. Python supports string concatenation, slicing, and various string methods like `upper()`, `lower()`, `replace()`, and `split()`. Strings are **immutable**, meaning once created, their contents cannot be changed.

Example:

```
# Numbers
a = 10
b = 5.5
sum_result = a + b  # Result is a float

# Strings
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name  # Concatenation
```

1.12 Introduction to Sequences

Q: What are sequences in Python, and what types of sequences exist?

A:
In Python, a **sequence** is an ordered collection of elements. Sequences allow you to store and manage multiple elements in a structured way. Python provides several types of sequences, each with unique characteristics and functionalities. Sequences support operations like indexing, slicing, and iteration.

The main sequence types in Python are:

- **Lists**: A mutable sequence that can store elements of different types. Lists are defined using square brackets `[]`.
- `my_list = [1, 2, 3, 'a', 'b']`
- **Tuples**: An immutable sequence used to store ordered collections of elements. Tuples are defined using parentheses `()`.
- `my_tuple = (1, 2, 3, 'a', 'b')`
- **Strings**: A sequence of characters. Strings are immutable in Python and are defined using single (`'`) or double (`"`) quotes.
- `my_string = "Hello"`
- **Ranges**: A sequence of numbers commonly used for iteration in `for` loops. Ranges are immutable and created using the `range()` function.
- `my_range = range(1, 5)  # Generates numbers 1 to 4`

Sequences are an essential concept in Python because they provide an easy way to store, manipulate, and iterate over ordered data.

1.13 Strings and Their Operations

Q: How are strings treated as sequences in Python, and what operations can be performed on strings?

A:
In Python, **strings** are sequences of characters that are **immutable**, meaning that once a string is created, its contents cannot be changed. However, strings can be manipulated or transformed using a variety of operations.

Some of the basic string operations include:

1. **Indexing**: You can access individual characters in a string using **indexing**, where indexing starts at `0`.

```
my_string = "Hello"
print(my_string[0])  # Output: H
```

4. **Slicing**: Slicing allows you to access a substring from a string.

```
print(my_string[1:4])  # Output: ell
```

6. **Concatenation**: You can join two or more strings together using the `+` operator.

```
greeting = "Hello"
name = "Alice"
message = greeting + " " + name  # Output: "Hello Alice"
```

10. **Repetition**: The `*` operator can be used to repeat a string multiple times.

```
print("abc" * 3)  # Output: abcabcabc
```

12. **Length**: The `len()` function returns the length of the string.

```
print(len(my_string))  # Output: 5
```

14. **Iteration**: Strings can be iterated over using loops like `for`.

```
for char in my_string:
    print(char)
```

Strings in Python also support methods like `lower()`, `upper()`, `replace()`, `split()`, and `strip()` for text manipulation.

1.14 Sequence Operations

Q: What are common operations performed on sequences (lists, tuples, etc.) in Python?

A:
Sequences in Python (lists, tuples, strings) support a wide variety of operations that make it easy to manipulate, traverse, and modify their contents. Some common operations include:

1. **Indexing**: As mentioned earlier, elements in a sequence can be accessed using their index.

```
my_list = [10, 20, 30]
print(my_list[1])  # Output: 20
```

4. **Slicing**: You can extract a portion of the sequence using slicing, defined as `sequence[start:end]`. This operation does not modify the original sequence but instead returns a new sequence.

```
my_list = [10, 20, 30, 40, 50]
print(my_list[1:4])  # Output: [20, 30, 40]
```

7. **Concatenation**: Sequences can be combined using the `+` operator to create a new sequence.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2  # Output: [1, 2, 3, 4, 5, 6]
```

11. **Repetition**: Sequences can be repeated using the `*` operator.

```
my_list = [1, 2]
print(my_list * 3)  # Output: [1, 2, 1, 2, 1, 2]
```

14. **Membership**: You can check if an element exists in a sequence using the `in` keyword.

```
my_list = [1, 2, 3]
print(2 in my_list)  # Output: True
```

17. **Iteration**: You can iterate through a sequence using loops.

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

20. **Length**: Use `len()` to get the number of elements in a sequence.

```
print(len(my_list))  # Output: 3
```

These operations help make Python sequences highly flexible and useful for a wide range of applications.

1.15 String Operators and Functions

Q: What are some key operators and functions that can be used with strings in Python?

A:
Strings in Python come with several built-in operators and functions that allow for efficient text manipulation. These include:

1. **Operators**:
   - **Concatenation (+)**: Combines two strings into one.
   - `result = "Hello" + " " + "World"  # Output: "Hello World"`
   - **Repetition (*)**: Repeats a string multiple times.
   - `result = "Python" * 3  # Output: "PythonPythonPython"`
2. **Built-in Functions**:
   - **`len()`**: Returns the length of a string.
   - `print(len("Hello"))  # Output: 5`
   - **`str()`**: Converts other data types into a string.
   - `print(str(123))  # Output: "123"`
   - **`ord()`**: Converts a character into its corresponding ASCII value.
   - `print(ord('A'))  # Output: 65`
   - **`chr()`**: Converts an ASCII value into its corresponding character.
   - `print(chr(65))  # Output: "A"`
3. **String Methods**:
   - **`lower()`**: Converts all characters to lowercase.
   - `print("Hello".lower())  # Output: "hello"`
   - **`upper()`**: Converts all characters to uppercase.
   - `print("Hello".upper())  # Output: "HELLO"`
   - **`replace()`**: Replaces a substring with another substring.
   - `print("Hello World".replace("World", "Python"))  # Output: "Hello Python"`
   - **`strip()`**: Removes leading and trailing whitespace from the string.
   - `print("  Hello  ".strip())  # Output: "Hello"`

1.16 Special Features of Strings

Q: What are some special features or functionalities that Python strings support?

A:
Python provides several special features that make strings highly versatile and easy to work with. Some key features include:

1. **Immutability**: Strings in Python are **immutable**, meaning once a string is created, it cannot be modified. Any operation that modifies a string creates a new string.

```
2. my_string = "Hello"
3. new_string = my_string.replace("H", "J")  # Creates a new string, "Jello"
```

4. **String Formatting**: Python provides multiple ways to format strings, including:
   - **f-strings** (Python 3.6+): Directly embed expressions inside strings.

- `name = "Alice"`
- `greeting = f"Hello, {name}"  # Output: "Hello, Alice"`
- **`format()` method**: Format strings by inserting values into placeholders.
- `"Hello, {}".format("Alice")  # Output: "Hello, Alice"`

5. **Escape Sequences**: Python supports special escape sequences in strings:
   - `\n` for newline
   - `\t` for tab
   - `\\` for a backslash

```
6. print("Hello\nWorld")  # Output: "Hello" (new line) "World"
```

7. **Multiline Strings**: You can create multiline strings using triple quotes (`'''` or `"""`).

```
8. multiline_string = """This is
9. a multiline string."""
```

1.17 Memory Management in Strings

Q: How is memory managed in Python when dealing with strings?

A:
In Python, **memory management** for strings is handled automatically by the Python interpreter through its **garbage collector**. Strings are immutable, so when a string is modified, a new string is created, and the old string may eventually be garbage-collected.

Python also uses **string interning**, which helps save memory by storing only one copy of identical strings. For example:

```
a = "Hello"
b = "Hello"
print(a is b)  # Output: True
```

In this case, both `a` and `b` refer to the same memory location, as Python internally reuses the string `"Hello"`.

1.18 Programming Examples

Q: Provide a programming example demonstrating the use of strings and sequences in Python.

A:
Here is an example that demonstrates string operations, slicing, and sequence manipulation:

```
# String Operations
name = "Python"
greeting = "Hello, " + name  # Concatenation
print(greeting)  # Output: Hello, Python

# String Slicing
substring = name[1:4]
print(substring)  # Output: yth
```

```
# List as a sequence
numbers = [1, 2, 3, 4, 5]
numbers.append(6)  # Adding an element to the list
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]

# Looping through a string
for char in name:
    print(char, end=" ")  # Output: P y t h o n
```

This example demonstrates how to perform string concatenation, slicing, and list manipulation. Python's support for sequences like lists and strings makes it a powerful and flexible language for a wide range of applications.

1.19 The if Statement

Q: What is the purpose of the `if` statement in Python, and how is it used?

A:
The **`if` statement** is one of the core conditional control structures in Python. It allows you to make decisions in your code by executing a block of code only when a specific condition is true.

Syntax:

```
if condition:
    # Code to execute if condition is True
```

- The `condition` is an expression that evaluates to either `True` or `False`.
- If the condition is `True`, the block of code indented beneath the `if` statement is executed.
- If the condition is `False`, the program moves to the next control structure or statement.

Example:

```
age = 18
if age >= 18:
    print("You are an adult.")
```

Output:

```
You are an adult.
```

In the above example, the condition `age >= 18` evaluates to `True`, so the code inside the `if` block gets executed.

1.20 The else Statement

Q: How does the `else` statement work in conjunction with the `if` statement in Python?

A:
The **`else` statement** is used to define a block of code that is executed when the condition in the corresponding `if` statement is **false**. It is always used after an `if` or `elif` block. While `if` handles the case when the condition is `True`, `else` handles the case when the condition is `False`.

Syntax:

```
if condition:
    # Code to execute if condition is True
else:
    # Code to execute if condition is False
```

Example:

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

Output:

```
You are a minor.
```

Here, since `age >= 18` is `False`, the code inside the `else` block executes.

1.21 The elif Statement

Q: What is the role of the `elif` statement in Python? How does it differ from `else`?

A:
The **`elif`** (short for "else if") statement provides additional conditions to check, allowing for more than two possibilities. It follows an `if` statement and allows you to check multiple conditions sequentially. If the first condition (`if`) is `False`, the program checks the condition in the `elif` statement, and so on.

You can have multiple `elif` statements in a chain, but only the first block with a `True` condition will be executed. If none of the conditions are `True`, the `else` block will execute (if provided).

Syntax:

```
if condition1:
    # Code for condition1
elif condition2:
    # Code for condition2
elif condition3:
    # Code for condition3
else:
    # Code if no condition is True
```

Example:

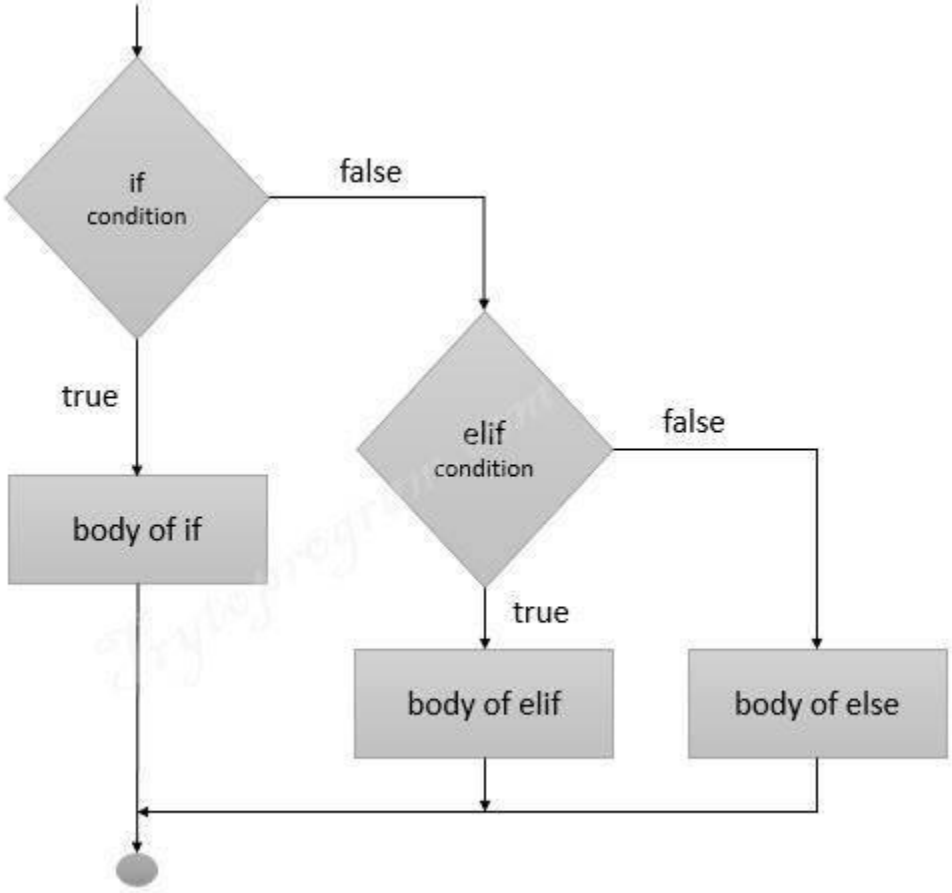
```
age = 20
if age < 18:
```

```
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior.")
```

Output:

```
You are an adult.
```

Here, the first condition `age < 18` is `False`, so the program checks the `elif` condition `age >= 18 and age < 65`, which is `True`. Therefore, the code inside the `elif` block is executed.



1.22 The while Loop

Q: What is the `while` loop, and when should it be used in Python?

A:
The **`while` loop** is used to execute a block of code **repeatedly** as long as a given **condition** remains true. The condition is checked before each iteration, and if it evaluates to `True`, the loop continues

to execute. If the condition evaluates to `False`, the loop terminates, and the program continues with the next statement after the loop.

The `while` loop is useful when the number of iterations is not known in advance and depends on the outcome of a condition.

Syntax:

```
while condition:
    # Code to execute as long as condition is True
```

Example:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Output:

```
1
2
3
4
5
```

In this example, the loop continues to execute as long as `count <= 5`. After each iteration, `count` is incremented by 1. Once `count` becomes 6, the condition `count <= 5` becomes `False`, and the loop stops.

1.23 The for Loop

Q: What is the `for` loop in Python, and how does it differ from the `while` loop?

A:
The **`for` loop** is used to iterate over a **sequence** (like a list, tuple, string, or range) and execute a block of code for each item in the sequence. Unlike the `while` loop, which depends on a condition to terminate, the `for` loop automatically terminates when it has iterated over all items in the sequence.

The `for` loop is typically used when the number of iterations is known or when you are iterating over a collection of data.

Syntax:

```
for item in sequence:
    # Code to execute for each item in sequence
```

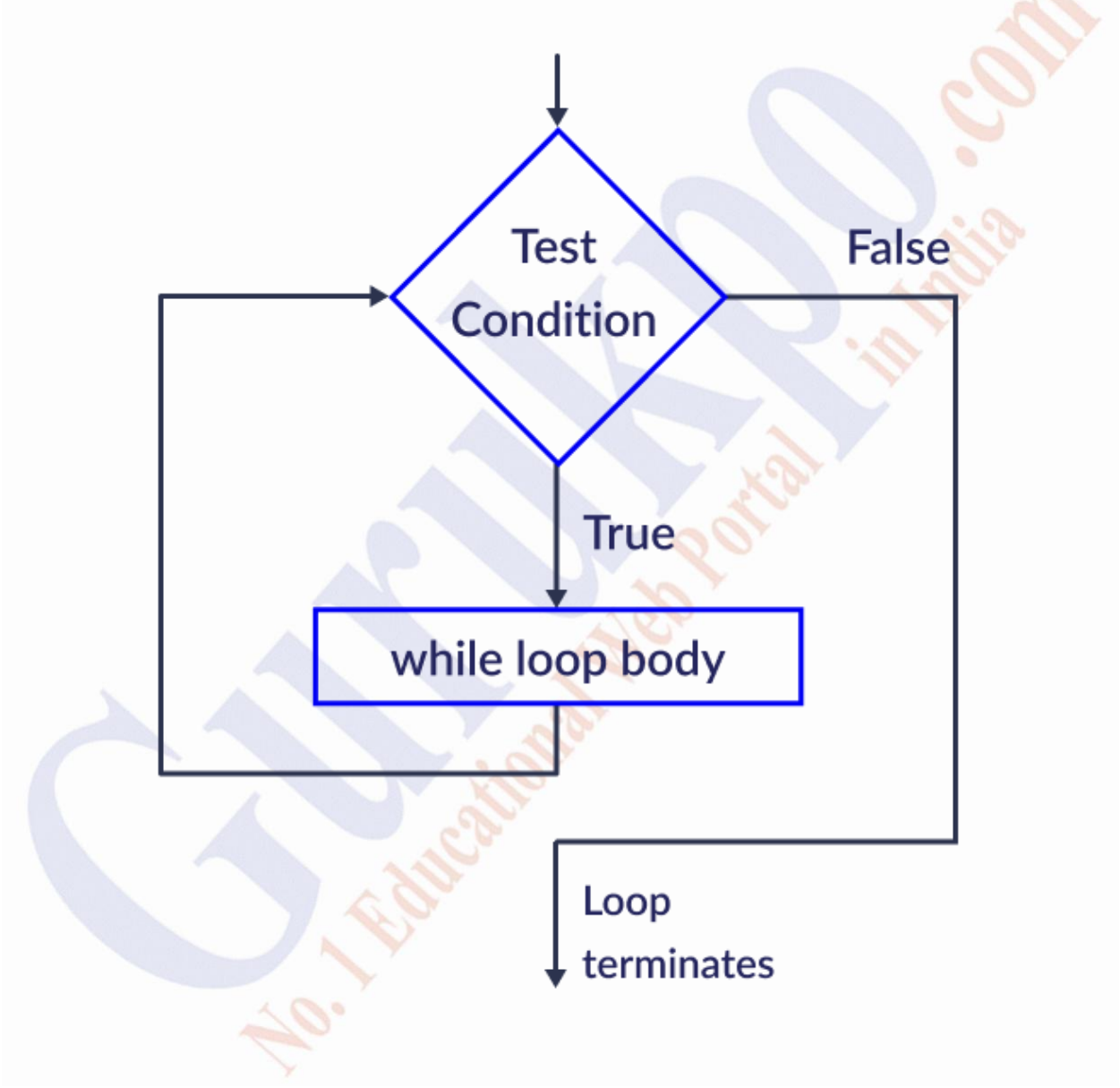
Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

Here, the `for` loop iterates over each item in the list `fruits`, and the code inside the loop is executed for each item.



1.24 break, continue, and pass Statements

Q: What are the functions of the `break`, `continue`, and `pass` statements in Python loops?

A:
These three statements are used to control the flow of execution inside loops:

1. **`break`**: The `break` statement is used to **exit** a loop early, regardless of whether the loop's condition is `True`. It is typically used when you want to terminate the loop based on a specific condition.

```
for i in range(10):
    if i == 5:
        break  # Exit the loop when i equals 5
    print(i)
```

6. **`continue`**: The `continue` statement is used to **skip** the current iteration of the loop and proceed with the next iteration. The rest of the code in the loop after the `continue` statement is skipped for the current iteration.

```
for i in range(10):
    if i == 5:
        continue  # Skip the iteration when i equals 5
     print(i)
```

11. **`pass`**: The `pass` statement is a **placeholder** that does nothing. It is used when a statement is syntactically required but no action is needed. It is often used as a placeholder for future code or within empty loops or functions.

```
 for i in range(10):
    if i == 5:
        pass  # Do nothing when i equals 5
    else:
        print(i)
```

1.25 else with Loop

Q: How is the `else` statement used with loops in Python?

A:
In Python, an **`else`** block can be used in conjunction with both `for` and `while` loops. The `else` block is executed when the loop has **completed all iterations** normally (i.e., the loop did not terminate via a `break` statement).

- The `else` block is **not executed** if the loop is prematurely terminated by a `break` statement.
- It is executed when the loop finishes all iterations successfully.

Syntax:

```
for item in sequence:
    # Code to execute for each item
else:
    # Code to execute if the loop completes without a break
```

Example:

```
for i in range(5):
    print(i)
else:
    print("Loop completed without a break.")
```

Output:

```
0
1
2
3
4
Loop completed without a break.
```

In this case, the loop completes without encountering a `break` statement, so the `else` block is executed.

If you use a `break` inside the loop, the `else` block will be skipped:

```
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print("Loop completed without a break.")
```

Output:

```
0
1
2
```

Here, the loop terminates early at `i == 3` because of the `break`, so the `else` block is skipped.

Chapter 2
Objects and Classes

2.1 Introduction to Classes in Python

Q: What are classes in Python, and how do they contribute to object-oriented programming?

A:
In Python, a **class** is a blueprint or template for creating objects. A class defines the attributes (data) and methods (functions) that the objects created from the class will have. Classes allow us to bundle data and functionality together, making it easier to organize and manage code. Classes are the cornerstone of **Object-Oriented Programming (OOP)**, which is a programming paradigm that revolves around the concept of objects and their interactions.

A class defines a set of properties and behaviors that are shared by all instances (objects) of that class. Objects are instances of classes, and each object can have its own specific values for the attributes defined in the class.

Example:

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Instance method
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age  # Instance variable

    def bark(self):
        print(f"{self.name} says Woof!")

# Create an instance of the Dog class
dog1 = Dog("Buddy", 3)
dog1.bark()  # Output: Buddy says Woof!
```

In this example, `Dog` is the class, and `dog1` is an object (instance) of that class.

2.2 Principles of Object-Oriented Programming (OOP)

Q: What are the fundamental principles of object-oriented programming, and how do they relate to Python?

A:
Object-Oriented Programming (OOP) is a programming paradigm that uses **objects** and **classes** to

organize and structure code. Python, being an object-oriented language, follows the four primary principles of OOP:

1. **Encapsulation**:
   Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data within a single unit or class. It helps in hiding the internal workings of an object and only exposes a well-defined interface.

   In Python, encapsulation can be achieved by defining attributes and methods within a class. You can control access to attributes using **getter** and **setter** methods or access modifiers like `private` and `protected`.

2. **Abstraction**:
   Abstraction involves hiding complex implementation details and only exposing essential features. It simplifies interactions with objects by providing a clear and simplified interface.

   In Python, you can use abstract base classes (ABCs) to enforce abstraction, making sure that derived classes implement specific methods.

3. **Inheritance**:
   Inheritance allows one class (child class) to inherit the attributes and methods from another class (parent class). This helps in reusing code and establishing relationships between different classes.

   Python supports single and multiple inheritance, allowing you to create a new class based on an existing class.

4. **Polymorphism**:
   Polymorphism refers to the ability of different classes to be treated as instances of the same class through inheritance. It allows different objects to respond to the same method call in different ways.

   In Python, polymorphism can be achieved through **method overriding**, where a child class provides its own implementation of a method that is already defined in the parent class.

2.3 Creating Classes

Q: How do you create a class in Python, and what are the key components of a class?

A:
To create a class in Python, you use the `class` keyword followed by the class name (which should follow the standard naming convention). The class body contains methods (functions) and attributes (variables) that define the class.

A class generally has the following key components:

- **Class attributes**: Variables shared by all instances of the class.
- **Instance attributes**: Variables unique to each instance of the class.
- **Methods**: Functions that define the behaviors of the objects.

Syntax:

```
class ClassName:
    # Constructor method to initialize attributes
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # Class method
    def method1(self):
        pass
```

Example:

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        print(f"{self.year} {self.brand} {self.model}")

car1 = Car("Toyota", "Camry", 2020)
car1.display_info()  # Output: 2020 Toyota Camry
```

2.4 Instance Methods

Q: What are instance methods in Python, and how do they differ from class methods?

A:
Instance methods are functions that are defined inside a class and operate on instances of that class. They take at least one argument, usually called `self`, which refers to the instance of the class on which the method is called. These methods can access and modify the attributes of the instance.

In contrast, **class methods** (defined with the `@classmethod` decorator) are functions that operate on the class itself, rather than on an instance. They take `cls` as their first parameter instead of `self`.

Example of an instance method:

```
class Dog:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Woof! I am {self.name}.")

dog1 = Dog("Buddy")
dog1.greet()  # Output: Woof! I am Buddy.
```

Here, `greet` is an instance method that uses `self` to access the `name` attribute of the `dog1` object.

2.5 Class Variables

Q: What are class variables in Python, and how are they different from instance variables?

A:
Class variables are variables that are shared among all instances of a class. They are defined inside the class but outside any methods. Every instance of the class has access to these variables, and they are generally used to store properties or data that should be shared across all instances.

Instance variables, on the other hand, are specific to each instance of the class. They are typically defined inside the `__init__` constructor and are unique to each object.

Example of class variables:

```
class Dog:
    species = "Canis familiaris"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable

dog1 = Dog("Buddy")
dog2 = Dog("Lucy")

print(dog1.species)  # Output: Canis familiaris
print(dog2.species)  # Output: Canis familiaris
```

In this case, `species` is a class variable shared by all instances of `Dog`.

2.6 Inheritance and Polymorphism

Q: What is inheritance, and how does polymorphism work in Python?

A:
Inheritance is a mechanism that allows one class (child class) to inherit attributes and methods from another class (parent class). This enables code reuse and the creation of a class hierarchy. The child class can override or extend the behavior of the parent class.

Polymorphism allows different classes to have methods with the same name but with different behaviors. In Python, polymorphism is often achieved through **method overriding**, where a subclass provides a specific implementation for a method already defined in its parent class.

Example of inheritance and polymorphism:

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")
```

```
class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

dog = Dog()
cat = Cat()

dog.speak()  # Output: Woof!
cat.speak()  # Output: Meow!
```

In this example, both `Dog` and `Cat` inherit from the `Animal` class and override the `speak` method, showcasing polymorphism.

2.7 Type Identification

Q: How can you check the type of an object in Python?

A:
In Python, you can use the `type()` function to identify the type of an object. This is helpful when debugging or ensuring that variables hold the expected type.

Example:

```
x = 5
print(type(x))  # Output: <class 'int'>

y = "Hello"
print(type(y))  # Output: <class 'str'>
```

The `type()` function returns the class type of the object passed to it.

2.8 Python Libraries (Strings, Data Structures & Algorithms)

Q: What are some commonly used Python libraries for working with strings, data structures, and algorithms?

A:
Python offers a wide variety of **standard libraries** and **third-party libraries** for working with strings, data structures, and algorithms. Here are some of the most popular ones:

1. **Strings**:

- Python provides built-in methods for string manipulation, such as `lower()`, `upper()`, `split()`, and `join()`. You can also use regular expressions (via the `re` module) for advanced string operations.

2. **Data Structures**:
   - **`collections` module**: Contains useful data structures like `deque`, `Counter`, `OrderedDict`, and `defaultdict`.
   - **`heapq` module**: Provides heap-based priority queues.
   - **`array` module**: For arrays, which are more efficient than lists for numerical data.
3. **Algorithms**:
   - Python's standard library includes basic algorithms for sorting, searching, and manipulating data. For advanced algorithms, third-party libraries like **`NumPy`** (for numerical data), **`pandas`** (for data analysis), and **`networkx`** (for graph-based algorithms) are commonly used.

By combining these libraries with Python's built-in capabilities, you can efficiently implement algorithms and handle complex data structures.

2.9 Built-in Functions for Lists

Q: What are some of the built-in functions available for working with lists in Python?

A:
Python provides several built-in functions that are particularly useful for working with lists. Here are some key built-in functions:

1. **`len()`**: Returns the number of elements in a list.

```
lst = [1, 2, 3]
print(len(lst))  # Output: 3
```

4. **`min()`**: Returns the smallest item in the list.

```
lst = [1, 2, 3]
print(min(lst))  # Output: 1
```

7. **`max()`**: Returns the largest item in the list.

```
lst = [1, 2, 3]
print(max(lst))  # Output: 3
```

10. **`sum()`**: Returns the sum of all items in the list (for numeric lists).

```
lst = [1, 2, 3]
print(sum(lst))  # Output: 6
```

13. **`sorted()`**: Returns a sorted version of the list.

```
lst = [3, 1, 2]
print(sorted(lst))  # Output: [1, 2, 3]
```

16. **`all()`**: Returns `True` if all elements in the list evaluate to `True`.

```
lst = [True, True, True]
print(all(lst))  # Output: True
```

19. **`any()`**: Returns `True` if any element in the list evaluates to `True`.

```
lst = [False, False, True]
print(any(lst))  # Output: True
```

These functions can be used for various list manipulations, making list operations efficient and easy to implement.

2.10 List Type Built-in Methods

Q: What are some of the built-in methods available for lists in Python?

A:
Python provides several useful methods for manipulating lists. Here are some of the most commonly used ones:

1. **`append()`**: Adds an item to the end of the list.

```
lst = [1, 2]
lst.append(3)
print(lst)  # Output: [1, 2, 3]
```

5. **`extend()`**: Adds all items from another iterable (e.g., list, tuple) to the list.

```
lst = [1, 2]
lst.extend([3, 4])
print(lst)  # Output: [1, 2, 3, 4]
```

9. **`insert()`**: Adds an element at a specific index in the list.

```
lst = [1, 3]
lst.insert(1, 2)
print(lst)  # Output: [1, 2, 3]
```

13. **`remove()`**: Removes the first occurrence of an element from the list.

```
lst = [1, 2, 3]
lst.remove(2)
print(lst)  # Output: [1, 3]
```

17. **`pop()`**: Removes and returns the element at the specified index (default is the last item).

```
lst = [1, 2, 3]
item = lst.pop(1)
print(item)  # Output: 2
print(lst)   # Output: [1, 3]
```

22. **`sort()`**: Sorts the list in ascending order (can specify reverse sorting).

```
lst = [3, 1, 2]
lst.sort()
print(lst)  # Output: [1, 2, 3]
```

26. **`reverse()`**: Reverses the order of elements in the list.

```
lst = [1, 2, 3]
lst.reverse()
print(lst)  # Output: [3, 2, 1]
```

30. **`count()`**: Returns the number of occurrences of an element in the list.

```
lst = [1, 2, 2, 3]
print(lst.count(2))  # Output: 2
```

These methods help you modify, retrieve, and manipulate lists efficiently in Python.

2.11 Tuples and Their Features

Q: What are tuples in Python, and what are their features?

A:
A **tuple** is a collection of ordered, immutable elements. Tuples are similar to lists but differ in that they cannot be changed (i.e., they are **immutable**). Here are some key features of tuples:

1. **Ordered**: The elements in a tuple have a defined order, and they can be accessed via indices, just like lists.
2. **Immutable**: Once a tuple is created, its contents cannot be modified (i.e., you cannot change, add, or remove elements).
3. **Can contain different data types**: A tuple can store a variety of data types, including integers, floats, strings, and even other tuples.
4. **Faster than lists**: Because tuples are immutable, they are generally faster than lists when it comes to iteration and access.
5. **Defined using parentheses**: Tuples are defined using parentheses `()` rather than square brackets `[]`.

Example:

```
t = (1, 2, 3)
print(t[1])  # Output: 2
```

2.12 Tuple Operators

Q: What operators can be used with tuples in Python?

A:
Tuples in Python support a variety of operators:

1. **Concatenation (`+`)**: Tuples can be concatenated together using the `+` operator.

```
t1 = (1, 2)
t2 = (3, 4)
t3 = t1 + t2
print(t3)  # Output: (1, 2, 3, 4)
```

6. **Repetition (`*`)**: A tuple can be repeated a specified number of times using the `*` operator.

```
t = (1, 2)
print(t * 3)  # Output: (1, 2, 1, 2, 1, 2)
```

9. **Membership (`in`, `not in`)**: You can check whether an element exists in a tuple using the `in` and `not in` operators.

```
t = (1, 2, 3)
print(2 in t)  # Output: True
print(4 not in t)  # Output: True
```

13. **Indexing and Slicing**: You can access elements of a tuple using indexing (`[]`), and you can slice tuples using the colon (`:`).

```
t = (1, 2, 3, 4)
print(t[1])  # Output: 2
print(t[1:3])  # Output: (2, 3)
```

2.13 Set: Introduction and Accessing Elements

Q: What is a set in Python, and how do you access elements in a set?

A:
A **set** is an unordered collection of unique elements. Sets are similar to lists and tuples, but they do not allow duplicate values and do not maintain order. Sets are defined using curly braces `{}`.

Key features of sets:

- **Unordered**: The elements in a set are not stored in any particular order.
- **No duplicates**: Sets automatically remove duplicate elements.
- **Mutable**: You can add or remove elements from a set after it has been created.

Example:

```
s = {1, 2, 3, 4}
print(s)  # Output: {1, 2, 3, 4}
```

To access elements, you can use iteration (sets do not support indexing or slicing like lists or tuples):

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

2.14 Set Methods (Add, Update, Clear, Copy, Discard, Remove)

Q: What are some common methods used to modify sets in Python?

A:
Python provides several methods for modifying sets:

1. **`add()`**: Adds an element to the set (if not already present).

```
s = {1, 2, 3}
s.add(4)
print(s)  # Output: {1, 2, 3, 4}
```

5. **`update()`**: Adds multiple elements to the set.

```
s = {1, 2, 3}
s.update([4, 5])
print(s)  # Output: {1, 2, 3, 4, 5}
```

9. **`clear()`**: Removes all elements from the set.

```
s = {1, 2, 3}
s.clear()
print(s)  # Output: set()
```

13. **`copy()`**: Returns a shallow copy of the set.

```
s = {1, 2, 3}
new_s = s.copy()
print(new_s)  # Output: {1, 2, 3}
```

17. **`discard()`**: Removes an element from the set if it exists. Does nothing if the element is not present.

```
s = {1, 2, 3}
s.discard(2)
print(s)  # Output: {1, 3}
```

21. **`remove()`**: Removes an element from the set. Raises a `KeyError` if the element is not present.

```
s = {1, 2, 3}
s.remove(2)
print(s)  # Output: {1, 3}
```

2.15 Set Operations (Union, Intersection, Difference)

Q: What are the common set operations in Python, and how do they work?

A:
Sets in Python support several common set operations that are useful for mathematical set theory:

1. **Union (`|` or `union()`)**: Returns a new set that contains all the elements from both sets.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1 | s2)  # Output: {1, 2, 3, 4, 5}
```

5. **Intersection (`&` or `intersection()`)**: Returns a new set that contains only the elements that are present in both sets.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1 & s2)  # Output: {3}
```

9. **Difference (`-` or `difference()`)**: Returns a new set that contains elements present in the first set but not in the second set.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1 - s2)  # Output: {1, 2}
```

13. **Symmetric Difference (`^` or `symmetric_difference()`)**: Returns a new set that contains elements in either of the sets, but not in both.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1 ^ s2)  # Output: {1, 2, 4, 5}
```

These set operations help you easily manipulate and compare sets in Python.

Chapter 3
Dictionaries, Files, and Regular Expressions

3.1 Introduction to Dictionaries

Q: What is a dictionary in Python, and what are its features?

A:
A **dictionary** in Python is an unordered collection of key-value pairs. Each key is unique, and each key maps to a value. Dictionaries are highly flexible and can store data of any type (e.g., integers, strings, lists, other dictionaries).

Key features of dictionaries:

1. **Unordered**: Dictionaries do not maintain the order of the elements (though order preservation is guaranteed starting from Python 3.7).
2. **Key-value pairs**: Each element in a dictionary consists of a key and a corresponding value.
3. **Mutable**: You can add, remove, or change items in a dictionary.
4. **Keys are unique**: No two keys in a dictionary can be the same.
5. **Keys are immutable**: The keys must be of a hashable type, such as strings, numbers, or tuples.

Example:

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

3.2 Built-in Functions for Dictionaries

Q: What are some built-in functions for dictionaries in Python?

A:
Python provides several built-in functions to work with dictionaries:

1. **`len()`**: Returns the number of key-value pairs in the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
print(len(my_dict))  # Output: 2
```

4. **`min()`**: Returns the smallest key in the dictionary based on the lexicographical order.

```
my_dict = {'name': 'Alice', 'age': 25}
print(min(my_dict))  # Output: 'age'
```

7. **`max()`**: Returns the largest key in the dictionary based on lexicographical order.

```
my_dict = {'name': 'Alice', 'age': 25}
print(max(my_dict))  # Output: 'name'
```

10. **`keys()`**: Returns a view object that displays all the keys in the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.keys())  # Output: dict_keys(['name', 'age'])
```

13. **`values()`**: Returns a view object that displays all the values in the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.values())  # Output: dict_values(['Alice', 25])
```

16. **`items()`**: Returns a view object that displays a list of tuples, where each tuple is a key-value pair.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.items())      # Output: dict_items([('name', 'Alice'), ('age', 25)])
```

These functions are essential for accessing and manipulating dictionaries.

3.3 Dictionary Methods

Q: What are some common methods used to manipulate dictionaries in Python?

A:
Python dictionaries offer several useful methods for adding, removing, and updating key-value pairs:

1. **`get()`**: Returns the value associated with a key, or `None` if the key is not found.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.get('name'))  # Output: Alice
print(my_dict.get('address'))  # Output: None
```

5. **`update()`**: Updates the dictionary with the key-value pairs from another dictionary or iterable.

```
my_dict = {'name': 'Alice'}
my_dict.update({'age': 25})
print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

9. **`pop()`**: Removes and returns the value for a given key. Raises a `KeyError` if the key is not found.

```
my_dict = {'name': 'Alice', 'age': 25}
value = my_dict.pop('age')
print(value)  # Output: 25
print(my_dict)  # Output: {'name': 'Alice'}
```

14. **`popitem()`**: Removes and returns an arbitrary key-value pair as a tuple.

```
my_dict = {'name': 'Alice', 'age': 25}
item = my_dict.popitem()
print(item)  # Output: ('age', 25) (order is arbitrary)
```

18. **`clear()`**: Removes all the items from the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
my_dict.clear()
print(my_dict)  # Output: {}
```

22. **`setdefault()`**: Returns the value if the key is in the dictionary. If not, inserts the key with a default value.

```
my_dict = {'name': 'Alice'}
print(my_dict.setdefault('age', 25))  # Output: 25
print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

These methods provide powerful ways to interact with dictionaries in Python.

3.4 Dictionary Keys and Access

Q: How are keys accessed in a dictionary, and what are the rules for key types?

A:
In Python, dictionary keys are accessed using square brackets (`[]`), or the **`get()`** method. The key must be unique and hashable (e.g., strings, numbers, and tuples).

Example of accessing keys:

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict['name'])  # Output: Alice
```

- **Key Types**: Keys must be **immutable**. Thus, types like strings, numbers, and tuples can be used as keys, but lists or other dictionaries cannot.
- `my_dict = {(1, 2): 'tuple'}  # Valid key`
- `# my_dict = {[1, 2]: 'list'}  # Invalid key (will raise TypeError)`

The **`get()`** method can be used to safely access values without raising an exception:

```
print(my_dict.get('name'))  # Output: Alice
print(my_dict.get('address', 'Not Found'))  # Output: Not Found
```

3.5 Sorting and Looping in Dictionaries

Q: How can you loop through a dictionary and sort it in Python?

A:
In Python, dictionaries are unordered, but you can loop through them using `for` loops. You can also sort a dictionary by its keys or values using the `sorted()` function.

Looping through a dictionary:

You can loop through a dictionary in multiple ways:

1. **Loop through keys**:

```
my_dict = {'name': 'Alice', 'age': 25}
for key in my_dict:
    print(key)  # Output: name, age
```

5. **Loop through keys and values**:

```
for key, value in my_dict.items():
    print(key, value)  # Output: name Alice, age 25
```

Sorting a dictionary:

You can sort dictionaries by their keys or values:

1. **Sort by keys**:

```
my_dict = {'name': 'Alice', 'age': 25}
sorted_keys = sorted(my_dict.keys())
print(sorted_keys)  # Output: ['age', 'name']
```

5. **Sort by values**:

```
sorted_items = sorted(my_dict.items(), key=lambda x: x[1])
print(sorted_items)  # Output: [('age', 25), ('name', 'Alice')]
```

3.6 Nested Dictionaries

Q: What is a nested dictionary, and how can it be used?

A:
A **nested dictionary** is a dictionary that contains other dictionaries as its values. It allows for hierarchical data representation, where a dictionary can have another dictionary inside it.

Example of a nested dictionary:

```
nested_dict = {
    'student1': {'name': 'Alice', 'age': 25, 'course': 'Computer Science'},
    'student2': {'name': 'Bob', 'age': 24, 'course': 'Mathematics'}
}
```

To access the values of a nested dictionary:

```
print(nested_dict['student1']['name'])  # Output: Alice
print(nested_dict['student2']['course'])  # Output: Mathematics
```

Nested dictionaries are useful for representing complex data structures such as JSON data, student records, etc.

3.7 File Objects and File Modes

Q: What are file objects and file modes in Python?

A:
In Python, **file objects** represent the interface to a file on the disk, allowing you to interact with the file, such as reading from it, writing to it, or modifying it. File objects are created using the `open()` function.

When opening a file, you need to specify a **file mode**, which determines how the file will be accessed (read, write, etc.). Some common file modes include:

1. `'r'`: Read mode. The file must exist.
2. `'w'`: Write mode. Creates a new file or overwrites an existing file.
3. `'a'`: Append mode. Adds content to the end of the file.
4. `'rb'`: Read mode in binary.

5. **`'wb'`**: Write mode in binary.
6. **`'x'`**: Exclusive creation mode. Creates a new file, but raises an error if the file already exists.

Example:

```
file = open('example.txt', 'r')  # Open file for reading
content = file.read()
print(content)
file.close()
```

3.8 File Built-in Functions

Q: What are some built-in functions for working with files in Python?

A:
Python provides several built-in functions to work with file objects:

1. **`open()`**: Opens a file and returns a file object.
2. `file = open('file.txt', 'r')`
3. **`read()`**: Reads the content of the file. You can specify the number of bytes to read or leave it empty to read the whole file.
4. `content = file.read()`
5. **`readline()`**: Reads the next line from the file.
6. `line = file.readline()`
7. **`readlines()`**: Reads all the lines from the file and returns a list.
8. `lines = file.readlines()`
9. **`write()`**: Writes a string to the file.
10. `file.write('Hello, World!')`
11. **`writelines()`**: Writes a list of strings to the file.
12. `file.writelines(['Line 1\n', 'Line 2\n'])`
13. **`close()`**: Closes the file object.
14. `file.close()`
15. **`flush()`**: Flushes the internal buffer to the file.
16. `file.flush()`
17. **`seek()`**: Moves the file pointer to a specified position.
18. `file.seek(0)`
19. **`tell()`**: Returns the current position of the file pointer.

```
position = file.tell()
```

3.9 File Methods and Attributes

Q: What are some methods and attributes used for file objects?

A:
File objects in Python come with a number of methods and attributes for file manipulation:

1. **`file.read(size)`**: Reads up to the specified `size` bytes from the file.

2. **`file.readline()`**: Reads a single line from the file.
3. **`file.readlines()`**: Reads all lines from the file and returns them as a list.
4. **`file.write(string)`**: Writes the specified `string` to the file.
5. **`file.writelines(list)`**: Writes a list of strings to the file.
6. **`file.seek(offset)`**: Moves the file pointer to the specified `offset`.
7. **`file.tell()`**: Returns the current position of the file pointer.
8. **`file.flush()`**: Flushes the internal buffer to the file.
9. **`file.truncate(size)`**: Truncates the file to the specified size.
10. **`file.close()`**: Closes the file object.

File Attributes:

1. **`file.name`**: Returns the name of the file.
2. **`file.mode`**: Returns the mode in which the file was opened.
3. **`file.closed`**: Returns `True` if the file is closed.

Example:

```
with open('file.txt', 'r') as file:
    print(file.name)  # Output: file.txt
    print(file.mode)  # Output: r
    print(file.read())
```

3.10 Standard Files

Q: What are standard files in Python?

A:
In Python, **standard files** refer to the predefined file objects for the input and output streams. These are:

1. **`sys.stdin`**: Standard input stream. It is used to read input from the user or from other programs.
2. **`sys.stdout`**: Standard output stream. It is used to write output to the terminal or console.
3. **`sys.stderr`**: Standard error stream. It is used to write error messages.

You can use these streams like normal file objects for input and output operations.

Example:

```
import sys
sys.stdout.write('Hello, World!')  # Output to standard output
```

3.11 Command-line Arguments

Q: How do you use command-line arguments in Python?

A:
Python allows you to pass arguments to a script through the command line. These arguments are stored in `sys.argv`, which is a list of strings.

1. The first element (`sys.argv[0]`) is the script name.
2. Subsequent elements are the command-line arguments passed to the script.

To access and use these arguments, you need to import the `sys` module.

Example:

```
import sys
print(f'Script name: {sys.argv[0]}')
print(f'Arguments passed: {sys.argv[1:]}')
```

Running the script:

```
python script.py arg1 arg2 arg3
```

Output:

```
Script name: script.py
Arguments passed: ['arg1', 'arg2', 'arg3']
```

3.12 File System and Execution

Q: How can you interact with the file system in Python?

A:
Python provides several modules for interacting with the file system. The **`os`** module is commonly used for file and directory operations.

1. **`os.getcwd()`**: Gets the current working directory.
2. **`os.chdir(path)`**: Changes the current working directory to the specified path.
3. **`os.listdir(path)`**: Lists all files and directories in the specified path.
4. **`os.mkdir(path)`**: Creates a new directory.
5. **`os.remove(path)`**: Deletes a file.

The **`os`** module also includes functions for checking file existence, permissions, and more.

Example:

```
import os
print(os.getcwd())  # Get current directory
os.mkdir('new_directory')  # Create a new directory
os.remove('old_file.txt')  # Remove a file
```

3.13 Persistent Storage Modules

Q: What are persistent storage modules in Python?

A:
Persistent storage refers to storing data in a way that allows it to persist after the program execution ends. Python provides several modules for handling persistent storage:

1. **`pickle`**: Used to serialize and deserialize Python objects (i.e., save and load Python objects to/from a file).

```
import pickle
with open('data.pkl', 'wb') as file:
    pickle.dump(my_object, file)  # Serialize and save object
```

5. **`shelve`**: A module for storing Python objects in a dictionary-like format, supporting persistent storage.

```
import shelve
with shelve.open('data_shelf') as shelf:
    shelf['key'] = 'value'  # Store data
    print(shelf['key'])  # Access data
```

10. **`sqlite3`**: A database interface module to interact with SQLite databases, allowing persistent data storage in databases.

```
import sqlite3
conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER, name TEXT)')
conn.commit()
conn.close()
```

3.14 Introduction and Motivation

Q: What is the motivation behind using regular expressions (REs) in Python?

A:
Regular expressions (REs) provide a powerful way to perform pattern matching and text manipulation in Python. They allow developers to define search patterns that can match specific parts of a string, such as finding words, extracting data, or replacing text.

Motivation for using REs includes:

1. **Text Search and Extraction**: Regular expressions are particularly useful when you need to find or extract specific data within a large volume of text, such as emails, dates, or phone numbers.
2. **String Validation**: REs are widely used for validating input strings, like checking if an email address or phone number matches the required format.
3. **Text Transformation**: Regular expressions also allow text to be modified, for example, replacing specific patterns in a string, or splitting text into more manageable parts based on delimiters.

4. **Efficiency**: Regular expressions provide an efficient way to search and manipulate text compared to writing manual string-processing code.

In Python, the **`re`** module provides all the necessary functionality to work with regular expressions.

3.15 Special Symbols and Characters in REs

Q: What are the special symbols and characters used in regular expressions?

A:
In regular expressions, special symbols and characters are used to define patterns that can match multiple types of strings. Here are some of the most commonly used special symbols in REs:

1. **`.` (Dot)**: Matches any single character except for a newline.
   - Example: `a.b` will match "acb", "axb", "a1b", etc.
2. **`^` (Caret)**: Matches the start of a string.
   - Example: `^abc` will match "abc" only if it appears at the beginning of the string.
3. **`$` (Dollar Sign)**: Matches the end of a string.
   - Example: `abc$` will match "abc" only if it appears at the end of the string.
4. **`*` (Asterisk)**: Matches zero or more occurrences of the preceding character or group.
   - Example: `a*b` will match "b", "ab", "aab", "aaab", etc.
5. **`+` (Plus Sign)**: Matches one or more occurrences of the preceding character or group.
   - Example: `a+b` will match "ab", "aab", "aaab", etc., but not "b".
6. **`?` (Question Mark)**: Matches zero or one occurrence of the preceding character or group.
   - Example: `a?b` will match "b" or "ab".
7. **`[]` (Square Brackets)**: Matches any one of the characters inside the brackets.
   - Example: `[aeiou]` matches any vowel in the string.
8. **`|` (Pipe)**: Acts as a logical OR, matching either the pattern before or after the `|`.
   - Example: `cat|dog` will match either "cat" or "dog".
9. **`()` (Parentheses)**: Used for grouping and capturing patterns.
   - Example: `(abc)+` will match one or more occurrences of the string "abc".
10. **`\` (Backslash)**: Escapes a special character, or specifies a special sequence.
    - Example: `\\` matches a single backslash, `\d` matches any digit, `\w` matches any word character (alphanumeric).
11. **`{}` (Braces)**: Specifies the exact number of occurrences for the preceding character or group.
    - Example: `a{2,4}` matches "aa", "aaa", or "aaaa".
12. **`\d`**: Matches any digit (0-9).
    - Example: `\d{3}` matches any three-digit number.
13. **`\w`**: Matches any alphanumeric character (letters, digits, and underscores).
    - Example: `\w+` matches one or more word characters.
14. **`\s`**: Matches any whitespace character (spaces, tabs, newlines).
    - Example: `\s+` matches one or more spaces.

3.16 Using Regular Expressions in Python

Q: How can regular expressions be used in Python to match and manipulate strings?

A:
In Python, the **`re`** module provides all the functions you need to work with regular expressions. Some of the most commonly used functions are:

1. **`re.match(pattern, string)`**: Tries to match the pattern at the start of the string. It returns a match object if the pattern is found at the beginning; otherwise, it returns `None`.
   - Example:
   - `import re`
   - `result = re.match(r'^abc', 'abcdef')`
   - `if result:`
   - `    print('Match found:', result.group())`
   - `else:`
   - `    print('No match')`
2. **`re.search(pattern, string)`**: Scans through the string and finds the first location where the pattern matches.
   - Example:
   - `result = re.search(r'abc', 'abcdef')`
   - `if result:`
   - `    print('Match found:', result.group())`
3. **`re.findall(pattern, string)`**: Returns all non-overlapping matches of the pattern in the string as a list of strings.
   - Example:
   - `result = re.findall(r'\d+', 'My phone number is 123-4567.')`
   - `print(result)  # Output: ['123', '4567']`
4. **`re.sub(pattern, replacement, string)`**: Replaces occurrences of the pattern in the string with the replacement text.
   - Example:
   - `result = re.sub(r'\d+', '###', 'My phone number is 123-4567.')`
   - `print(result)  # Output: 'My phone number is ###-###.'`
5. **`re.split(pattern, string)`**: Splits the string at each match of the pattern.
   - Example:
   - `result = re.split(r'\s+', 'This is a sentence with spaces.')`
   - `print(result)  # Output: ['This', 'is', 'a', 'sentence', 'with', 'spaces.']`
6. **`re.compile(pattern)`**: Compiles a regular expression pattern into a regex object that can be used multiple times.
   - Example:
   - `pattern = re.compile(r'\d+')`
   - `result = pattern.findall('My number is 1234 and my other number is 5678.')`
   - `print(result)  # Output: ['1234', '5678']`

Example Program:

```
import re

# Matching a simple pattern
```

```
text = 'My name is John and my phone number is 1234567890'
pattern = r'\d{10}'

# Search for a 10-digit phone number
match = re.search(pattern, text)
if match:
    print("Found a phone number:", match.group())
else:
    print("No phone number found.")
```

Explanation:
In this example, the pattern `\d{10}` is used to search for a sequence of 10 digits in the string. If a match is found, it prints the matched phone number.

Conclusion:

Regular expressions in Python offer a powerful and flexible way to work with text, from simple string searches to complex text manipulation. By mastering regular expressions, you can greatly improve your ability to process and analyze strings efficiently.

Chapter 4
Exceptions, Database Interaction, and Multithreading

4.1 Concept of Exceptions

Q: What is an exception in Python, and why is exception handling important?
A:
An **exception** in Python is an error that occurs during the execution of a program, disrupting its normal flow. These errors are often caused by unforeseen events like trying to divide by zero, accessing a file that doesn't exist, or trying to access an index in a list that is out of range.

Exceptions in Python are raised when something goes wrong, and the **program flow is interrupted**. If not handled, exceptions can cause the program to terminate abruptly, leading to loss of data or bad user experience.

Exception handling is the process of anticipating and dealing with errors in a program. The primary goal of handling exceptions is to **prevent the program from crashing** and to provide a meaningful way of responding to errors, so the program can recover or gracefully exit.

Python provides a `try-except` mechanism for handling exceptions. When an exception occurs in the `try` block, Python jumps to the corresponding `except` block to handle it. This allows the program to continue running even if an error occurs.

Example:

```
try:
    x = 5 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

4.2 Handling Exceptions in Python

Q: How does Python handle exceptions using the `try-except` block, and why is it useful?
A:
In Python, **exceptions are handled** using the `try` and `except` blocks. The purpose of using this mechanism is to **catch** and **handle errors gracefully** without terminating the entire program. Here's a breakdown of how it works:

1. **`try` block**: This is where you place the code that might raise an exception. It is executed line by line until an exception occurs.

2. **`except` block**: If an exception occurs in the `try` block, the program immediately jumps to the `except` block, where you can specify the type of exception you want to handle.
3. **`else` block**: (Optional) This block runs if no exception occurs in the `try` block.
4. **`finally` block**: (Optional) This block always executes, regardless of whether an exception occurred or not. It's often used for cleanup operations (e.g., closing files or releasing resources).

The **`try-except`** mechanism helps **maintain control** over errors, making the program more **robust** and **user-friendly**.

Example:

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Please enter valid integers.")
else:
    print(f"The result is: {result}")
finally:
    print("Execution completed.")
```

4.3 Exception Objects and Strings

Q: What are exception objects in Python, and how can you access them?
A:
An **exception object** is an instance of an exception class that represents the error that occurred during the program execution. When an exception is raised, an object is created, containing details about the exception (e.g., error type, message, stack trace).

In Python, exceptions are **objects** that can be caught in the `except` block. You can access the exception's details, including the error message, by assigning it to a variable. This allows you to print or log the error message to inform the user about the problem.

You can use the `as` keyword in the `except` block to **capture** the exception object and then access its attributes, such as the error message.

Example:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    print(f"Error occurred: {e}")  # e is the exception object containing the
message
```

Output:

```
Error occurred: division by zero
```

4.4 Raising Exceptions

Q: How do you manually raise exceptions in Python, and why might this be necessary?
A:
In Python, you can raise exceptions manually using the `raise` keyword. This is useful when you want to trigger an exception based on certain conditions in your code, even when no error has occurred yet.

Raising exceptions manually is commonly used for input validation or when certain conditions in the program need to be enforced. For example, you might raise a `ValueError` if a user enters an invalid value.

You can raise built-in exceptions like `ValueError`, `TypeError`, or define your own custom exceptions. Raising exceptions helps to **control the flow of execution** and allows the program to behave in a predictable way when an invalid state is encountered.

Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)  # Output: Cannot divide by zero!
```

4.5 Assertions

Q: What is an assertion in Python, and how does it work?
A:
An **assertion** in Python is a debugging aid that tests whether a condition is true at a specific point in the program. It helps verify that certain assumptions or invariants are correct during the execution of the program. Assertions are used to check conditions that should never fail and help catch bugs early.

If an assertion fails (i.e., the condition is `False`), Python raises an `AssertionError` exception, which can optionally include a message. If the condition is `True`, the program continues without interruption.

Assertions are typically disabled when Python is run in **optimized mode** (with the `-O` flag), so they should not be used for critical checks that the program depends on.

Example:

```
x = 10
assert x > 0, "x must be positive"  # This will pass

assert x < 0, "x must be negative"  # This will raise an AssertionError with
the message "x must be negative"
```

4.6 Standard Exceptions

Q: What are some common standard exceptions in Python, and when are they typically raised?
A:
Python provides several built-in exceptions that represent common types of errors that occur during program execution. Here are some standard exceptions and their typical use cases:

1. **`ZeroDivisionError`**: Raised when attempting to divide a number by zero.
2. `x = 10 / 0  # ZeroDivisionError`
3. **`FileNotFoundError`**: Raised when trying to open a file that does not exist.
4. `with open("nonexistent_file.txt") as file:  # FileNotFoundError`
5. `    content = file.read()`
6. **`IndexError`**: Raised when trying to access an element in a list using an index that is out of range.
7. `lst = [1, 2, 3]`
8. `print(lst[5])  # IndexError`
9. **`TypeError`**: Raised when an operation or function is applied to an object of inappropriate type.
10. `x = "Hello" + 5  # TypeError: cannot concatenate 'str' and 'int' objects`
11. **`ValueError`**: Raised when a function receives an argument of the right type but an inappropriate value.
12. `x = int("abc")  # ValueError: invalid literal for int() with base 10: 'abc'`

Python provides many other built-in exceptions that allow you to handle specific error cases in a more granular way. By using specific exceptions, you can **handle different error scenarios** appropriately and provide meaningful feedback to users.

4.7 SQL Database Connection using Python

Q: How do you connect to an SQL database in Python using the `sqlite3` module? What steps are involved in making a database connection, and how do you interact with it?
A:
To connect to an SQL database in Python, we can use the `sqlite3` module, which is part of Python's standard library. Here are the steps involved:

1. **Import the sqlite3 module**: First, you import the `sqlite3` module to work with SQLite databases in Python.

2. **Establish a connection**: Use the `sqlite3.connect()` method to connect to an SQLite database. If the specified database does not exist, SQLite will automatically create it. If it exists, it connects to the existing database.
3. **Create a cursor object**: A cursor is used to execute SQL queries. After connecting to the database, we create a cursor object using `connection.cursor()`.
4. **Perform operations**: You can use SQL queries to perform various database operations such as creating tables, inserting data, or fetching data.
5. **Commit the changes**: After making changes to the database (e.g., inserting data), use `connection.commit()` to save the changes.
6. **Close the connection**: Always close the connection to the database using `connection.close()` when you're done.

Example:

```
import sqlite3

# Establish a connection to the database
connection = sqlite3.connect("example.db")

# Create a cursor object to interact with the database
cursor = connection.cursor()

# Perform operations like creating tables, inserting data, etc.

# Commit changes and close the connection
connection.commit()
connection.close()
```

4.8 Creating and Searching Tables

Q: How can you create a table in SQLite using Python and search for data in it?
A:

1. **Creating a Table**: You can create a table using the `CREATE TABLE` SQL statement. A table in an SQL database contains columns that store data values. Each column in the table has a specific data type, such as `INTEGER`, `TEXT`, or `REAL`.
2. **Searching for Data**: After creating a table and inserting data, you can retrieve data from the table using the `SELECT` SQL query. This allows you to search the table based on conditions (e.g., retrieving rows where the age is greater than 20).

Example:

```
import sqlite3

# Establish a connection to the database
connection = sqlite3.connect("example.db")
cursor = connection.cursor()

# Create a table
```

```
cursor.execute('''CREATE TABLE IF NOT EXISTS students (id INTEGER PRIMARY KEY,
name TEXT, age INTEGER)''')

# Insert data into the table
cursor.execute("INSERT INTO students (name, age) VALUES ('Alice', 22)")

# Search for data
cursor.execute("SELECT * FROM students WHERE age > 20")
results = cursor.fetchall()

# Print the results
for row in results:
    print(row)

# Commit and close
connection.commit()
connection.close()
```

4.9 Reading & Storing Configuration Info

Q: How can you store and retrieve configuration information in an SQLite database using Python?
A:
In many applications, configuration data such as settings, preferences, or options needs to be stored and accessed easily. You can store configuration data in an SQLite database by creating a dedicated table to hold the configuration keys and their corresponding values.

1. **Create a configuration table**: Define a table with columns for storing configuration keys and values. The key can be a string (e.g., "theme") and the value can be a string or another type (e.g., "dark", "light").
2. **Store configuration data**: Insert data into the configuration table using SQL `INSERT` statements. Each key-value pair represents a configuration setting.
3. **Retrieve configuration data**: Use the `SELECT` statement to fetch configuration values based on the key. This allows the program to load settings when it starts.

Example:

```
import sqlite3

# Connect to the database
connection = sqlite3.connect("config.db")
cursor = connection.cursor()

# Create a configuration table
cursor.execute('''CREATE TABLE IF NOT EXISTS config (key TEXT PRIMARY KEY, value
TEXT)''')

# Insert a configuration setting
cursor.execute("INSERT INTO config (key, value) VALUES ('theme', 'dark')")

# Retrieve a configuration value
cursor.execute("SELECT value FROM config WHERE key = 'theme'")
theme = cursor.fetchone()[0]
```

```
print(f"The selected theme is: {theme}")

# Commit changes and close the connection
connection.commit()
connection.close()
```

4.10 Programming with Database Connections

Q: How do you programmatically interact with a database using Python's `sqlite3` module, including committing transactions and closing the connection?
A:
To interact with a database in Python, the following steps are typically followed:

1. **Connect to the Database**: Use `sqlite3.connect()` to create a connection to the SQLite database.
2. **Create a Cursor Object**: After establishing a connection, a cursor object is created using `connection.cursor()`. The cursor is used to execute SQL queries.
3. **Execute SQL Queries**: You can execute any SQL query using the cursor object, such as `INSERT`, `UPDATE`, or `SELECT` queries.
4. **Commit Changes**: After executing any modification queries (e.g., inserting or updating data), you need to call `connection.commit()` to save the changes to the database. Without this, changes will not be persisted.
5. **Close the Connection**: Once all operations are complete, you should always close the database connection using `connection.close()` to free up system resources.

Example:

```
import sqlite3

# Connect to the database
connection = sqlite3.connect("products.db")
cursor = connection.cursor()

# Create a table for products
cursor.execute('''CREATE TABLE IF NOT EXISTS products (id INTEGER PRIMARY KEY,
name TEXT, price REAL)''')

# Insert data into the table
cursor.execute("INSERT INTO products (name, price) VALUES ('Laptop', 1200.00)")

# Commit the transaction to save changes
connection.commit()

# Close the connection
connection.close()
```

These examples show how to interact with a database using Python, covering the basics of connecting, querying, storing configuration data, and committing changes.

Recommended exercises

1. Write a program to demonstrate basic data type in python

2. Create a list and perform the following methods 1) insert() 2) remove() 3) append() 4) len() 5) pop() 6) clear()

3. Create a tuple and perform the following methods 1) Add items 2) len() 3) check for item in
4. tuple 4)Access items

5. Create a dictionary and apply the following methods 1) Print the dictionary items 2) access items 3) use get() 4)change values 5) use len()

6. Write a program to create a menu with the following options 1. TO PERFORM ADDITITON 2.TO PERFORM SUBTRACTION 3. TO PERFORM MULTIPICATION 4. TO PERFORM DIVISION Accepts users input and perform the operation accordingly. Use functions with arguments.

7. Write a python program to print a number is positive/negative using if-else.

8. Write a program for filter() to filter only even numbers from a given list.

9. Write a python program to print date, time for today and now

10. Write a python program to add some days to your present date and print the date added.

11. Write a program to count the numbers of characters in the string and store them in a dictionary data structure

12. Write a program to count frequency of characters in a given file.

13. Using a numpy module create an array and check the following: 1. Type of array 2. Axes of array Shape of array 4. Type of elements in array

14. Write a python program to concatenate the dataframes with two different objects

15. Write a python code to read a csv file using pandas module and print the first and last five lines of a file.

16. Write a python program which accepts the radius of a circle from user and computes the area (use math module)

BIYANI GIRLS COLLEGE

First Mid Term Exam 2024

Class : BCA **Marks : 30** **Unit Covered: 2**

Subject with code : Python

SET – A

Very Short Type Questions	**M.M.**	**CO Mapping**	**Bloom's Taxonomy Level**	**PO & PSO Mapping**
Q1. What is keyword ? Name any four keywords in Python	1	**CO1**	**1**	PO1, PSO1
Q2. What is List?	1	**CO2**	**1**	PO2, PSO1

Short Type Questions	**M.M.**	**CO Mapping**	**Bloom's Taxoomy Level**	**Pos & Psos Mapping**
Q1 Explain Tuple and Dictionary in Python with example.	4	CO2	2	PO1, PSO1
Q2 What do you understand by looping statements in Python? Describe its types with example.	4	CO2	1	PO2, PSO1

Long Type Questions	**M.M.**	**CO Mapping**	**Bloom's Taxoomy Level**	**Pos & Psos Mapping**
Q1. Explain Identity operator and Membership Operator.	10	CO2	2	PO1, PSO2
Q2. What do you mean by Function in Python? Explain its types: Arbitrary argument, Keyword argument and Arbitrary Keyword Arguments with example.	10	CO2	1	PO1, PSO1

BIYANI GIRLS COLLEGE

First Mid Term Exam 2024

Class : BCA **Marks : 30** **Unit Covered: 2**

Subject with code : Python

SET – B

Very Short Type Questions	**M.M.**	**CO Mapping**	**Bloom's Taxonomy Level**	**PO & PSO Mapping**
Q1. What is variable and data type?	1	**CO1**	**1**	PO1, PSO1
Q2. What is Lamda Function?	1	**CO2**	**1**	PO2, PSO1

Short Type Questions	**M.M.**	**CO Mapping**	**Bloom's Taxoomy Level**	**Pos & Psos Mapping**
Q1 What do you understand by looping statements in Python? Describe its types with example.	4	CO2	1	PO1, PSO1
Q2 Explain List and Tuple with example.	4	CO2	2	PO2, PSO1

Long Type Questions	**M.M.**	**CO Mapping**	**Bloom's Taxoomy Level**	**Pos & Psos Mapping**
Q1. What do you mean by Function in Python? Explain its types: Arbitrary argument, Keyword argument and Arbitrary Keyword Arguments with example.	10	CO2	1	PO1, PSO2
Q2. Explain Identity operator and Membership Operator.	10	CO2	2	PO1, PSO1