**1. Printing on Screen**

**1.Introduction to the print() function in Python**

* print() is used to display text or variables on the screen.
* It can take multiple arguments separated by commas.
* By default, it adds a newline (\n) after each call.

### ****2.Formatting outputs using f-strings****

* Introduced in Python 3.6.
* Allows embedding variables directly inside strings using {}.

**2. Reading Data from Keyboard**

**1. • Using the input() function to read user input from the keyboard.**

 In Python, we use the **input()** function to take input from the user.

 By default, input() always returns data as a **string**.

 To convert input to other data types, we use type casting functions like int(), float(), etc.

2. **Converting user input into different data types (e.g., int, float, etc.).**

**name = input("Enter your name: ") # string**

**age = int(input("Enter your age: ")) # integer**

**height = float(input("Enter your height in meters: ")) # float**

**3. Opening and Closing Files**

## **1. Opening files in different modes**

In Python, the open() function is used to open a file.  
👉 Syntax:

file = open("filename", "mode")

Here:

* **filename** → the name of the file you want to open
* **mode** → tells Python how you want to open the file

### Common Modes:

| **Mode** | **Meaning** | **Behavior** |
| --- | --- | --- |
| 'r' | Read | Default mode. Opens file for reading. File **must exist**. |
| 'w' | Write | Opens file for writing. **Creates a new file** if not exists, otherwise **overwrites**. |
| 'a' | Append | Opens file for writing. Creates if not exists, but **does not overwrite**, instead adds new data at end. |
| 'r+' | Read + Write | File must exist. Allows both reading and writing. |
| 'w+' | Write + Read | Creates new file if not exists, otherwise overwrites. Allows both reading and writing. |

## **2. Using open() function to create and access files**

# Open file in write mode (creates new file if not exists)

file = open("example.txt", "w")

# Write content

file.write("This is the first line.\n")

file.write("This is the second line.\n")

print("Data written successfully!")

# Always close the file

file.close()

## **3. Closing files using close()**

**It’s important to close files after operations to free memory.**

**file = open("example.txt", "r")**

**print(file.read())**

**file.close()**

**4. Reading and Writing Files**

## **1. Reading from a file**

When you open a file in **read mode ('r')**, you can use:

### (a) read() → Reads the ****entire file**** as a single string

with open("example.txt", "r") as file:

content = file.read()

print("Using read():")

print(content)

### (b) readline() → Reads ****one line at a time****

with open("example.txt", "r") as file:

line1 = file.readline()

line2 = file.readline()

print("Using readline():")

print(line1.strip())

print(line2.strip())

### (c) readlines() → Reads ****all lines into a list****

with open("example.txt", "r") as file:

lines = file.readlines()

print("Using readlines():")

for line in lines:

print(line.strip())

## **2. Writing to a file**

When you open a file in **write ('w')** or **append ('a')** mode, you can use:

### (a) write() → Writes a single string

with open("write\_example.txt", "w") as file:

file.write("Hello, World!\n")

file.write("This is a second line.\n")

print("Data written using write()")

### (b) writelines() → Writes a ****list of strings****

lines\_to\_write = [

"Line 1: Python is powerful.\n",

"Line 2: File handling is easy.\n",

"Line 3: writelines() writes multiple strings.\n"

]

with open("write\_example.txt", "a") as file: # append mode

file.writelines(lines\_to\_write)

print("Data written using writelines()")

**5. Exception Handling**

**1. Handling Exceptions with try and except**

* Code that might cause an error goes inside **try** block.
* Errors are caught and handled in the **except** block.

**Example:**

try:

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

result = 10 / num

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except ValueError:

print("Error: Invalid input. Please enter a number.")

**2. Using finally**

* The **finally** block always runs (whether error occurs or not).
* Commonly used to **close files, release resources, or cleanup tasks**.

**Example:**

try:

file = open("data.txt", "r")

content = file.read()

print(content)

except FileNotFoundError:

print("Error: File not found.")

finally:

print("Closing file (if opened).")

try:

file.close()

except:

pass

**3. Handling Multiple Exceptions**

* Multiple exceptions can be caught separately or together.

**Example (multiple except):**

try:

x = int("abc") # will raise ValueError

y = 10 / 0 # will raise ZeroDivisionError

except ValueError:

print("Invalid conversion to integer.")

except ZeroDivisionError:

print("Cannot divide by zero.")

**Example (single except with tuple):**

try:

x = int("abc")

except (ValueError, TypeError):

print("Invalid input type or value.")

**4. Custom Exceptions**

* Python allows creating user-defined exceptions by extending the Exception class.

**Example:**

class AgeError(Exception):

pass

age = int(input("Enter your age: "))

try:

if age < 18:

raise AgeError("You must be at least 18 years old.")

print("Access granted.")

except AgeError as e:

print("Custom Exception:", e)

6. Class and Object (OOP Concepts)

**1. Classes and Objects in Python**

* **Class** → A blueprint for creating objects. It defines attributes (variables) and methods (functions).
* **Object** → An instance of a class (created using the class).
* **Attributes** → Variables inside a class that hold data.

### ****2. Local vs Global Variables****

* **Global Variable** → Declared **outside** a function/class, accessible everywhere.
* **Local Variable** → Declared **inside** a function/method, accessible only within it.

7. Inheritance

### ****Inheritance in Python****

Inheritance allows a class (**child class**) to use properties and methods of another class (**parent class**).

### ****1. Single Inheritance****

One child class inherits from one parent class.

### ****2. Multilevel Inheritance****

Inheritance across multiple levels (Grandparent → Parent → Child).

### ****3. Multiple Inheritance****

A child class inherits from **more than one parent class**.

### ****4. Hierarchical Inheritance****

Multiple child classes inherit from the same parent class.

### ****5. Hybrid Inheritance****

Combination of two or more types of inheritance.

### ****2.Using**** super() ****function****

super() is used to call methods of the **parent class**.

class Parent:

def \_\_init\_\_(self, name):

self.name = name

def show(self):

print("Parent:", self.name)

class Child(Parent):

def \_\_init\_\_(self, name, age):

super().\_\_init\_\_(name) # Call Parent constructor

self.age = age

def show(self):

super().show() # Call Parent method

print("Child Age:", self.age)

c = Child("John", 20)

c.show()

8. Method Overloading and Overriding

### ****1. Method Overloading****

* **Definition**: Defining multiple methods with the same name but different parameters.
* **Note in Python**: Python does **not support true method overloading** like Java or C++.
* Instead, Python allows **default arguments** or \*args, \*\*kwargs to achieve similar behavior.

### ****2. Method Overriding****

* **Definition**: When a **child class defines a method with the same name as a parent class method**, it **overrides** the parent’s method.
* Used for **runtime polymorphism** (behavior changes at runtime).

9. SQLite3 and PyMySQL (Database Connectors)

**1. Introduction to SQLite3 and PyMySQL for Database Connectivity**

* **SQLite3**:
  + A lightweight, serverless, and self-contained database engine.
  + It comes built-in with Python (sqlite3 module).
  + Useful for small to medium-sized applications where you don’t need a dedicated database server.
* **PyMySQL**:
  + A Python library used to connect Python applications with **MySQL databases**.
  + Requires installation (pip install pymysql).
  + Suitable for larger applications where a powerful relational database like MySQL is needed.

**2. Creating and Executing SQL Queries in Python**

Steps (common to both SQLite3 and PyMySQL):

1. Import the connector
   * import sqlite3 for SQLite3
   * import pymysql for MySQL
2. Connect to the database
   * SQLite3 → sqlite3.connect("mydb.db")
   * PyMySQL → pymysql.connect(host="localhost", user="root", password="", database="testdb")
3. Create a cursor object
   * cursor = conn.cursor()
4. Execute SQL queries
   * Use cursor.execute("SQL\_QUERY")
   * For multiple queries → use cursor.executemany()
5. Commit changes (for INSERT, UPDATE, DELETE)
   * conn.commit()
6. Fetch results (for SELECT queries)
   * cursor.fetchone() → fetches one record
   * cursor.fetchall() → fetches all records
7. Close the connection
   * conn.close()

10. Search and Match Functions

1.Using re.search() and re.match() functions in Python’s re module for pattern matching.

### ****1.**** re.search() ****Function****

* Used to **search for a pattern anywhere in the string**.
* If the pattern is found **at any position**, it returns a match object; otherwise, returns None.

### ****2.**** re.match() ****Function****

* Used to **check for a match only at the beginning of the string**.
* If the pattern is present at the start, it returns a match object; otherwise, returns None.

2. Difference Between re.search() and re.match()

| **Feature** | **re.match()** | **re.search()** |
| --- | --- | --- |
| **Search location** | Only at the **beginning** of the string | **Anywhere** in the string |
| **Return type** | Match object if pattern at start | Match object if pattern found anywhere |
| **Use case** | To check if string starts with a pattern | To find a pattern anywhere in text |