1. **Printing on Screen**

• Introduction to the print() function in Python.

* The print() function in Python is used to display output to the console. It allows you to print strings, numbers, and other objects, with several options for customizing the output. By default, print() separates multiple items with a space and ends with a newline, but you can modify these behaviors using the sep and end parameters. Additionally, you can redirect output to a file using the file parameter and control when the output is written with the flush parameter. Overall, print() is a fundamental tool for displaying results and debugging in Python.

• Formatting outputs using f-strings and format().

In Python, you can format strings using **f-strings** (available in Python 3.6+) and the **format()** method.

* **F-strings** allow you to embed expressions directly in strings with {} and prefix the string with f:

python

Copy code

name = "Alice"

age = 30

print(f"Hello, my name is {name} and I am {age} years old.")

* **format()** replaces {} placeholders with specified values:

python

Copy code

print("Hello, my name is {} and I am {} years old.".format(name, age))

Both methods support formatting options, such as controlling decimal places:

python

Copy code

pi = 3.14159

print(f"The value of pi is approximately {pi:.2f}.")

F-strings are more concise and preferred for newer code, while format() is flexible and works in older Python versions.

1. **Reading Data from Keyboard**

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

In Python, the input() function is used to read data from the user via the keyboard. By default, input() always returns the input as a string. You can then convert this string into other data types (like int, float, etc.) as needed.

**Basic Syntax:**

user\_input = input("Enter something: ")

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

Here are some examples that demonstrate how to read input and convert it into different data types:

**1. Reading a String Input:**

name = input("Enter your name: ")

print(f"Hello, {name}!")

**Example Output:**

mathematica

Copy code

Enter your name: Alice

Hello, Alice!

**2. Converting to an Integer (int):**

To convert the input string to an integer, use the int() function.

age = input("Enter your age: ")

age = int(age) # Convert the input to an integer

print(f"You are {age} years old.")

**Example Output:**

Enter your age: 25

You are 25 years old.

If the user enters a non-numeric value, this will raise a ValueError. To handle this, you can use error handling (e.g., try/except).

**3. Converting to a Float (float):**

For decimal numbers, convert the input to a float using the float() function.

price = input("Enter the price: ")

price = float(price) # Convert the input to a float

print(f"The price is {price:.2f}.")

**Example Output:**

Enter the price: 19.99

The price is 19.99.

**4. Converting with Type Handling:**

You can combine user input and type conversion in a single line for clarity.

num1 = int(input("Enter an integer: "))

num2 = float(input("Enter a float number: "))

sum\_result = num1 + num2

print(f"The sum is {sum\_result}")

**Example Output:**

bash

Enter an integer: 10

Enter a float number: 20.5

The sum is 30.5

1. **Opening and Closing Files**

• Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

In Python, the open() function is used to open files. The mode in which the file is opened determines how you can interact with the file (e.g., reading, writing, appending). Here are the different file modes you can use when opening files:

**File Modes:**

**1.'r'** – **Read Mode** (default)

* + Opens the file for reading. The file must exist, and the pointer is placed at the beginning of the file.

**2.'w'** – **Write Mode**

* Opens the file for writing. If the file already exists, it **overwrites** the file. If the file does not exist, it creates a new empty file.

**3. 'a'** – **Append Mode**

* Opens the file for appending (writing at the end of the file). If the file does not exist, it creates a new file. It does **not overwrite** existing content.

**4.'r+'** – **Read and Write Mode**

* + Opens the file for both reading and writing. The file must exist. The pointer is placed at the beginning of the file, and you can read and modify the content.

**5.'w+'** – **Write and Read Mode**

* Opens the file for both reading and writing. If the file exists, it is **overwritten**. If the file does not exist, a new file is created.

• Using the open() function to create and access files.

In Python, the open() function is used to open a file for reading, writing, or appending. If the file doesn't exist, it can also be created depending on the mode used. Here's how to create and access files using the open() function.

**Syntax:**

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

* "filename": The name of the file you want to create or access.
* "mode": The mode in which you want to open the file (e.g., 'r', 'w', 'a', 'x', etc.).

**Summary:**

* **'w'**: Creates or overwrites a file for writing.
* **'a'**: Creates or appends to a file.
* **'w+'**: Creates or overwrites a file for both reading and writing.
* **'x'**: Creates a file exclusively (raises an error if it already exists).
* **'r'**: Opens an existing file for reading.

• Closing files using close().

In Python, when you open a file using the open() function, it’s important to **close** the file after you're done with it using the close() method. This ensures that all changes are saved and resources (such as memory) are released.

**Basic Syntax to Close a File:**

file\_object.close()

**Example of Using close():**

**Opening and Closing a File in Write Mode**:

python

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

file.write("Hello, World!")

file.close() # Close the file after writing

* After writing the content to "example.txt", we call file.close() to ensure the file is properly closed.

1. **Reading and Writing Files**

• Reading from a file using read(), readline(), readlines().

In Python, there are several methods available to read the content of a file, depending on how you want to process the data. The most common methods are read(), readline(), and readlines(). These methods allow you to retrieve data from a file in different ways.

**1. read() Method**

The read() method reads the entire content of the file as a single string. It can be used when you want to read the whole file at once.

**Syntax:**

python

file\_object.read(size=-1)

* **size**: The number of bytes to read. If omitted or set to -1, the entire file will be read.

**Example:**

python

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

content = file.read()

print(content)

file.close()

**Explanation**:

* This reads the entire content of "example.txt" and prints it.

**Example with size argument:**

python

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

first\_10\_chars = file.read(10) # Read the first 10 characters

print(first\_10\_chars)

file.close()

**Explanation**:

* This reads only the first 10 characters of the file.

**2. readline() Method**

The readline() method reads a single line from the file. Each time you call readline(), it reads the next line until the end of the file is reached. This is useful when you need to process a file line by line.

**Syntax:**

file\_object.readline(size=-1)

* **size**: The maximum number of bytes to read. If omitted, the entire line will be read.

**Example:**

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

line1 = file.readline()

line2 = file.readline()

print(line1)

print(line2)

file.close()

**Explanation**:

* The first call to readline() reads the first line, and the second call reads the second line of the file.

**3. readlines() Method**

The readlines() method reads all the lines in a file and returns them as a list of strings, where each string represents a line from the file. This is useful if you want to work with the file's lines as individual list items.

**Syntax:**

file\_object.readlines(hint=-1)

* **hint**: The number of bytes to read. If omitted or set to -1, the entire file is read.

**Example:**

python

Copy code

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

lines = file.readlines()

for line in lines:

print(line.strip()) # strip() removes the newline character

file.close()

**Explanation**:

* This reads all lines in the file and stores them as a list. Each line is printed, and strip() is used to remove the newline character at the end of each line.

• Writing to a file using write() and writelines().

In Python, the write() and writelines() methods are used to write data to a file. These methods allow you to write strings or sequences of strings to a file, depending on your requirements.

**1. write() Method**

The write() method is used to write a single string to a file. It writes the content exactly as you provide it, without any additional newline characters unless you explicitly include them.

**Syntax:**

file\_object.write(string)

* **string**: The string you want to write to the file.

**Example:**

# Writing a single line to a file

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

file.write("Hello, World!")

**Explanation**:

* This opens (or creates) the file "example.txt" in write mode ('w'), and writes "Hello, World!" to the file. If the file already exists, it will be overwritten.

**Writing multiple strings with write():**

python

Copy code

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

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

file.write("This is a new line.")

**Explanation**:

* This writes two lines to the file by explicitly adding a newline character (\n) at the end of the first line.

**2. writelines() Method**

The writelines() method writes a sequence of strings (usually a list) to a file. Unlike write(), writelines() doesn't add newline characters between the strings, so you need to include them explicitly in the strings if required.

**Syntax:**

python

Copy code

file\_object.writelines(sequence)

* **sequence**: A sequence of strings (e.g., a list or tuple) that you want to write to the file.

**Example:**

lines = ["Hello, World!\n", "This is the second line.\n", "And this is the third line.\n"]

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

file.writelines(lines)

1. **Exception Handling**

• Introduction to exceptions and how to handle them using try, except, and finally.

Exceptions in Python are errors that occur during program execution. They can disrupt the normal flow of the program, but Python provides a way to handle them using try, except, and finally blocks. This allows the program to continue running smoothly, even when errors occur.

The try block contains code that might raise an exception. If an error occurs, Python jumps to the corresponding except block to handle the error. You can catch specific types of exceptions, such as ValueError or ZeroDivisionError, or use a generic except to catch all exceptions.

The finally block is used for cleanup actions, such as closing files or releasing resources, and it always runs regardless of whether an exception occurred. The finally block ensures that necessary final steps are taken, even if the code in the try block fails.

In summary, exceptions allow for more robust error handling, and using try, except, and finally blocks makes it easier to control how errors are handled and ensure that cleanup tasks are always performed.

• Understanding multiple exceptions and custom exceptions.

**Multiple Exceptions in Python**

Python allows you to handle multiple exceptions by using multiple except blocks or by grouping exceptions together in a tuple within a single except block. This enables you to manage different types of errors in a tailored way.

**Custom Exceptions**

You can create custom exceptions by defining a new class that inherits from Python's built-in Exception class. This lets you raise and handle specific errors relevant to your application, making error handling more flexible and organized.

1. **Class and Object (OOP Concepts)**

• Understanding the concepts of classes, objects, attributes, and methods in Python.

In Python, **classes** and **objects** are key concepts of **Object-Oriented Programming (OOP)**. OOP allows you to model real-world entities and behaviors in your programs.

* **Class**: A class is a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from the class will have.
* **Object**: An object is an instance of a class. It represents a specific entity with the attributes and behaviors defined by the class.
* **Attributes**: Attributes are variables that hold data or state for an object. They are defined inside the class and are accessed via objects.
* **Methods**: Methods are functions defined inside a class that describe the behaviors or actions an object can perform. They can modify the object's attributes or perform other tasks.

• Difference between local and global variables.

1 **Local Variables**:

* A **local variable** is declared inside a function or block of code and is only accessible within that specific function or block.
* It is created when the function is called and destroyed once the function execution is finished.
* Local variables are used to store temporary data and are not accessible outside their defined scope.

**Example**:

python

Copy code

def my\_function():

x = 10 # x is a local variable

print(x)

my\_function()

# print(x) # This would cause an error because x is local to my\_function

 **Global Variables**:

* A **global variable** is defined outside any function or class and can be accessed throughout the entire program, including inside functions.
* Global variables persist for the entire duration of the program and can be read or modified by any part of the program.
* You can access global variables inside functions by using the global keyword if you want to modify them.

**Example**:

x = 10 # x is a global variable

def my\_function():

print(x) # Can access global variable

my\_function() # Output: 10

To modify a global variable inside a function:

x = 10 # Global variable

def modify\_global():

global x # Declare x as global

x = 20 # Modify global variable

modify\_global()

print(x) # Output: 20

1. **Inheritance**

• Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

n Python, **inheritance** allows a class to inherit attributes and methods from another class. This enables code reuse and the creation of more specialized classes based on general ones. There are several types of inheritance in Python, including **single inheritance**, **multilevel inheritance**, **multiple inheritance**, **hierarchical inheritance**, and **hybrid inheritance**.

**1. Single Inheritance:**

In single inheritance, a class inherits from just one parent class.

**Example**:

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def bark(self):

print("Dog barks")

dog = Dog()

dog.speak() # Inherited method

dog.bark() # Dog's own method

**2. Multilevel Inheritance:**

In multilevel inheritance, a class inherits from another class, which itself is a subclass of another class.

**Example**:

class Animal:

def speak(self):

print("Animal speaks")

class Mammal(Animal):

def walk(self):

print("Mammal walks")

class Dog(Mammal):

def bark(self):

print("Dog barks")

dog = Dog()

dog.speak() # Inherited from Animal

dog.walk() # Inherited from Mammal

dog.bark() # Dog's own method

**3. Multiple Inheritance:**

In multiple inheritance, a class inherits from more than one parent class, gaining attributes and methods from all of them.

**Example**:

class Animal:

def speak(self):

print("Animal speaks")

class Walker:

def walk(self):

print("Walks on legs")

class Dog(Animal, Walker):

def bark(self):

print("Dog barks")

dog = Dog()

dog.speak() # Inherited from Animal

dog.walk() # Inherited from Walker

dog.bark() # Dog's own method

**4. Hierarchical Inheritance:**

In hierarchical inheritance, multiple classes inherit from a single parent class, sharing its attributes and methods.

**Example**:

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def bark(self):

print("Dog barks")

class Cat(Animal):

def meow(self):

print("Cat meows")

dog = Dog()

cat = Cat()

dog.speak() # Inherited from Animal

cat.speak() # Inherited from Animal

**5. Hybrid Inheritance:**

Hybrid inheritance is a combination of two or more types of inheritance. It can involve single, multiple, or hierarchical inheritance in different levels.

**Example**:

class Animal:

def speak(self):

print("Animal speaks")

class Walker:

def walk(self):

print("Walks on legs")

class Mammal(Animal, Walker):

def has\_hair(self):

print("Has hair")

class Dog(Mammal):

def bark(self):

print("Dog barks")

dog = Dog()

dog.speak() # Inherited from Animal

dog.walk() # Inherited from Walker

dog.has\_hair() # Inherited from Mammal

dog.bark() # Dog's own method

• Using the super() function to access properties of the parent class.

The super() function in Python is used to call methods or access properties from a parent class. It is especially useful in **inheritance** when you want to invoke the methods or access attributes from the **parent class** within the child class, typically when the method is overridden.

**How super() Works:**

* **super()** allows you to call methods or access properties from the **parent class** without explicitly mentioning the parent's name.
* It is often used in the \_\_init\_\_() method to ensure that the parent class is properly initialized in the child class.
* It helps to avoid direct references to the parent class, making your code more maintainable and flexible.

**Example 1: Using super() in the \_\_init\_\_() Method**

class Animal:

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

self.name = name

def speak(self):

print(f"{self.name} makes a sound")

class Dog(Animal):

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

# Using super() to call the \_\_init\_\_() of the parent class

super().\_\_init\_\_(name)

self.breed = breed

def speak(self):

super().speak() # Calls the speak() method of the parent class

print(f"{self.name} barks!")

# Creating an instance of Dog

dog = Dog("Buddy", "Golden Retriever")

dog.speak()

**Explanation:**

1. The Dog class inherits from the Animal class.
2. Inside the \_\_init\_\_() method of Dog, we use super().\_\_init\_\_(name) to call the \_\_init\_\_() method of the Animal class, allowing us to initialize the name attribute without repeating the code.
3. The speak() method in Dog calls super().speak() to execute the speak() method of Animal before executing its own behavior.

**Example 2: Accessing Parent Class Methods Using super()**

class Animal:

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

self.name = name

def sound(self):

return "Some generic animal sound"

class Dog(Animal):

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

super().\_\_init\_\_(name)

def sound(self):

parent\_sound = super().sound() # Accessing the parent class's sound method

return f"{parent\_sound}, but barks!"

# Creating a Dog object

dog = Dog("Rex")

print(dog.sound())

**Explanation:**

1. In the Dog class, the sound() method is overridden, but before customizing it, we call super().sound() to access the original behavior from the Animal class.
2. The super() function ensures that we are invoking the sound() method of the **parent class** (Animal), then extending its behavior in the child class (Dog)
3. **Method Overloading and Overriding**

• Method overloading: defining multiple methods with the same name but different parameters.

In Python, **method overloading** refers to the ability to define multiple methods with the same name but with different parameters (such as a different number or types of arguments). However, unlike some other programming languages (e.g., Java or C++), Python does not support **true method overloading** where you can define methods with the same name and different parameter types. Instead, Python provides a more flexible way to achieve similar functionality using default arguments, variable-length arguments, or by manually checking the number of arguments within a single method.

• Method overriding: redefining a parent class method in the child class.

**Method overriding** occurs when a **child class** defines a method with the same name and signature as a method in its **parent class**. This allows the child class to provide its own specific implementation of that method, replacing the behavior inherited from the parent class. Method overriding is a core concept in **Object-Oriented Programming (OOP)**, enabling polymorphism, where the method to be executed is determined at runtime based on the object calling the method.

**Key Points:**

1. **Parent Class Method**: The method defined in the parent class that can be overridden by the child class.
2. **Child Class Method**: The method in the child class that has the same name as the parent class's method but provides a new implementation.
3. **Polymorphism**: This allows different classes to define methods with the same name but with different behaviors.
4. **SQLite3 and PyMySQL (Database Connectors)**

• Introduction to SQLite3 and PyMySQL for database connectivity.

two commonly used libraries for interacting with databases are **SQLite3** and **PyMySQL**. These libraries enable you to connect to, query, and manipulate databases within your Python applications. Each library is designed to work with a specific type of database:

* **SQLite3** is a lightweight, file-based database that doesn't require a server to run. It's ideal for small-scale applications and testing purposes.
* **PyMySQL** is a MySQL client library for Python, allowing you to interact with MySQL databases, which require a running MySQL server.

**SQLite3**

**SQLite3** is a part of Python’s standard library, meaning it doesn't require an external installation. It allows you to work with SQLite databases, which are stored as files on your system. SQLite is serverless, which makes it great for applications where a full-fledged relational database system is not required.

• Creating and executing SQL queries from Python using these connectors.

To interact with databases and execute SQL queries from Python, we use the **SQLite3** and **PyMySQL** libraries. Both libraries allow you to perform operations like creating tables, inserting data, updating records, and querying data. Below is a guide on how to use these connectors to execute SQL queries.

**1. Using SQLite3**

SQLite3 is a lightweight, serverless database that is useful for small applications. Python’s sqlite3 library allows you to execute SQL queries and manipulate an SQLite database.

**Steps for SQLite3:**

1. **Create a connection to the database.**
2. **Create a cursor object** to execute SQL queries.
3. **Execute SQL queries** using cursor.execute().
4. **Fetch results** using fetchall() or fetchone().
5. **Commit changes** (for data-modifying queries).
6. **Close the connection** after operations.

**Example of Executing SQL Queries with SQLite3:**

python

Copy code

import sqlite3

# Step 1: Connect to SQLite database (it will create the file if it doesn't exist)

connection = sqlite3.connect('example.db')

# Step 2: Create a cursor object

cursor = connection.cursor()

# Step 3: Execute SQL queries

# Create a table (if not already exists)

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY,

name TEXT,

age INTEGER

)

''')

# Insert data

cursor.execute("INSERT INTO users (name, age) VALUES ('John', 30)")

cursor.execute("INSERT INTO users (name, age) VALUES ('Alice', 25)")

# Commit the transaction (save the changes)

connection.commit()

# Query data

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

for row in rows:

print(row) # Output the records

# Step 6: Close the connection

connection.close()

**Key SQL Queries with SQLite3:**

* **CREATE**: Create tables and schemas.
* **INSERT**: Add new records into a table.
* **SELECT**: Retrieve data from a table.
* **UPDATE**: Modify existing records.
* **DELETE**: Remove records from a table.

**2. Using PyMySQL**

**PyMySQL** is used to connect to MySQL databases, and it allows executing SQL queries against a MySQL server. You must have a MySQL server running and valid credentials (host, username, password, database).

**Steps for PyMySQL:**

1. **Connect to the MySQL database** using the pymysql.connect() method.
2. **Create a cursor object** for executing SQL queries.
3. **Execute SQL queries** using cursor.execute().
4. **Fetch results** using fetchall() or fetchone().
5. **Commit changes** if modifying data.
6. **Close the connection** after operations.

**Example of Executing SQL Queries with PyMySQL:**

python

Copy code

import pymysql

# Step 1: Connect to MySQL database

connection = pymysql.connect(

host='localhost',

user='root', # Your MySQL username

password='your\_password', # Your MySQL password

database='test\_db' # Database name

)

# Step 2: Create a cursor object

cursor = connection.cursor()

# Step 3: Execute SQL queries

# Create a table (if not already exists)

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

age INT

)

''')

# Insert data

cursor.execute("INSERT INTO users (name, age) VALUES ('John', 30)")

cursor.execute("INSERT INTO users (name, age) VALUES ('Alice', 25)")

# Commit the transaction (save the changes)

connection.commit()

# Query data

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

for row in rows:

print(row) # Output the records

# Step 6: Close the connection

connection.close()

**Key SQL Queries with PyMySQL:**

* **CREATE**: Define a new table or database.
* **INSERT**: Add new records.
* **SELECT**: Fetch records from the table.
* **UPDATE**: Update records.
* **DELETE**: Remove records.