Advance Python Programming

1. **Printing on Screen :**

* Introduction to the print() function in Python.
* The print() function in Python is one of the most commonly used functions for outputting information to the console. It is primarily used to display messages, variables, and the results of computations to the user or for debugging purposes.

**Syntax:**

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

* Formatting outputs using f-strings and format().
* Python provides powerful tools for formatting outputs through **f-strings** and the **format()** method. Both offer a way to embed expressions, control formatting, and improve readability when constructing strings.

**Using f-strings (Formatted String Literals)**

Introduced in Python 3.6, f-strings allow embedding expressions directly into string literals by prefixing the string with an f or F.

**Syntax:**

f"string with {expression}"

**Using format() Method**

The format() method is an older way to achieve similar functionality, supported in Python 2.7+.

**Syntax:**

"string with {} placeholders".format(values)

1. **Reading** **Data** **from** **Keyboard** :

* Using the input() function to read user input from the keyboard
* The input() function in Python allows you to read user input from the keyboard. This input is always returned as a string, regardless of what the user enters. If you need the input in a specific data type, you can convert it using functions like int(), float(), or others.

**Syntax:**

variable = input(prompt)

* Converting user input into different data types (e.g., int, float, etc.).
* The input received using the input() function is always of type str. If you need the input to be of a specific data type, you can use type conversion functions like int(), float(), or bool().

Converting to Integer (int):

user\_input = input("Enter an integer: ")

integer\_value = int(user\_input) # Convert to integer

print(f"You entered the integer: {integer\_value}")

1. **Opening and Closing Files:**

* Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').
* files can be opened in different modes using the open() function. Each mode serves a specific purpose:

Read Mode ('r'):-

\* Opens the file for reading only.

\* The file must exist, or an error will occur.

\* File pointer is positioned at the beginning

Write Mode ('w'):-

\* Opens the file for writing.

\* If the file exists, its contents are overwritten.

\* If the file does not exist, a new file is created.

Append Mode ('a'):-

\* Opens the file for appending.

\* If the file exists, new data is written at the end of the file without modifying existing content.

\* If the file does not exist, a new file is created.

Read and Write Mode ('r+'):-

\*Opens the file for both reading and writing.

\*The file must exist, or an error will occur.

\*File pointer is positioned at the beginning.

Write and Read Mode ('w+'):-

\*Opens the file for both writing and reading.

\*If the file exists, its contents are overwritten.

\*If the file does not exist, a new file is created

* Using the open() function to create and access files .
* The open() function in Python is used to create, read, write, and manipulate files. Here's a breakdown of how you can use it to create and access files:

**Syntax:**

file\_object = open(file\_name, mode, encoding)

* Closing files using close().
* When you open a file using the open() function, Python creates a file object. Once you are done with your file operations, you should close the file using file.close().

Example:

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

file.write("This is an example.")

file.close() # Close the file after writing

print("File closed successfully.")

* Reading from a file using read(), readline(), readlines().
* Using read():

**What it does**: Reads the entire content of the file as a single string.

**Use case**: When you want to load the full content of a file into memory.

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

content = file.read()

print("File content using read():")

print(content)

Using readline():

**What it does**: Reads one line at a time from the file.

**Use case**: When you need to process or print each line individually.

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

print("File content using readline():")

line = file.readline() # Read the first line

while line: # Loop until there are no more lines

print(line, end="") # Avoid double newlines

line = file.readline() # Read the next line

Using readlines():

**What it does**: Reads all the lines of the file and returns them as a list of strings.

**Use case**:When you want to process all lines of a file but keep them separate in a list.

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

lines = file.readlines()

print("File content using readlines():")

for line in lines:

print(line, end="") # Print each line

* Writing to a file using write() and writelines().
* Writing to a File Using write():

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

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

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

print("Data written using write().")

Writing to a File Using writelines():

lines = [

"Line 1: Hello, World!\n",

"Line 2: Writing to a file using writelines().\n",

"Line 3: Python file handling is easy!\n"

]

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

file.writelines(lines)

print("Data written using writelines().")

5. Exception Handling

* Introduction to exceptions and how to handle them using try, except, and finally
* Exceptions are events that disrupt the normal flow of a program's execution. They typically occur due to errors in the program, such as invalid user input, file not found, or division by zero. Python provides a robust mechanism to handle these exceptions using try, except, and finally blocks.

**try**: Defines a block of code to test for exceptions.

**except**: Catches and handles the exception.

**else**: (Optional) Executes if no exception occurs.

**finally**: (Optional) Executes code, regardless of whether an exception occurred or not.

* Understanding multiple exceptions and custom exceptions.
* you can handle multiple exceptions effectively and even define your own exceptions for specific use cases. This approach ensures precise error handling and makes debugging easier.

6. Class and Object (OOP Concepts

* Understanding the concepts of classes, objects, attributes, and methods in Python
* Python is an object-oriented programming (OOP) language that uses concepts like classes, objects, attributes, and methods to organize and structure code.

### **Classes:**

A class is a blueprint for creating objects.

It defines the attributes (data) and methods (functions) that an object of the class will have.

### **Objects:**

An object is an instance of a class.

It represents a specific example of the class blueprint.

You can create multiple objects from a single class.

### **Attributes:**

Attributes are variables that belong to a class or an object.

They define the properties or state of an object.

Types of attributes:

* + **Class Attributes:** Shared by all instances of the class.
  + **Instance Attributes:** Specific to each object, defined using the \_\_init\_\_ method.

### **Methods:**

Methods are functions defined inside a class.

They define the behavior of objects.

Types of methods:

**Instance Methods:** Operate on instance attributes and require self as the first parameter.

**Class Methods:** Operate on class attributes and require @classmethod and cls as the first parameter.

**Static Methods:** Do not operate on instance or class attributes and use @staticmethod

* Difference between local and global variables
* **Local Variables**

1. **Scope**: Accessible only within the function or block where they are declared.
2. **Lifetime**: Exist only during the execution of the function or block.
3. **Declaration**: Declared inside a function or block.
4. **Usage**: Commonly used to store temporary data specific to a function.
5. **Effect**: Changes to local variables do not affect the program outside the function or block

### **Global Variables**

1. **Scope**: Accessible throughout the entire program, including inside all functions and blocks (if not shadowed).
2. **Lifetime**: Exist for the duration of the program's execution.
3. **Declaration**: Declared outside any function or block.
4. **Usage**: Often used to store data that needs to be shared across multiple parts of the program.
5. **Effect**: Changes to global variables affect the entire program

7. Inheritance

* Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python
* Single: One child class inherits from one parent class.

Multilevel: A chain of inheritance where a class inherits from another class, and so on.

Multiple: A class inherits from more than one parent class.

Hierarchical: Multiple child classes inherit from a single parent class.

Hybrid: A combination of two or more types of inheritance.

* Using the super() function to access properties of the parent class
* The super() function in Python is used to access methods or properties of the parent class from a child class. It is commonly used in inheritance to call the parent class's methods or initialize its properties.

class Parent:

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

self.name = name

def show\_name(self):

print(f"Name: {self.name}")

class Child(Parent):

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

# Call the parent class's \_\_init\_\_ method using super()

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

self.age = age

def show\_details(self):

# Call the parent class's show\_name method

super().show\_name()

print(f"Age: {self.age}")

# Create an object of the Child class

child = Child("Alice", 25)

# Access parent and child class properties and methods

child.show\_details()

8. Method Overloading and Overriding

* Method overloading: defining multiple methods with the same name but different parameters
* method overloading (defining multiple methods with the same name but different parameters) is not supported in the same way as some other programming languages like Java or C++. Python allows only the last defined method with a given name to exist.

However, you can achieve similar functionality using default arguments or by handling different numbers and types of arguments inside a single method.

* Method overriding: redefining a parent class method in the child class
* **Method overriding** occurs when a child class redefines a method from the parent class. It allows the child class to provide a specific implementation of the method that replaces or modifies the parent class’s implementation.

9. SQLite3 and PyMySQL (Database Connectors)

* Introduction to SQLite3 and PyMySQL for database connectivity.
* Introduction to SQLite3 and PyMySQL for Database Connectivity

Python provides several libraries for database connectivity, two of the most commonly used are SQLite3 and PyMySQL.

SQLite3: A lightweight, disk-based database that doesn’t require a separate server process. It is built into Python’s standard library, making it easy to use for small applications.

PyMySQL: A MySQL client library for Python. It allows Python code to interact with a MySQL database. It is an external library, and you need to install it before using it.

SQLite3 in Python

SQLite is a self-contained, serverless, zero-configuration database engine. Python comes with the sqlite3 module in the standard library, making it easy to work with SQLite databases.

Basic SQLite3 Operations

Creating a Database SQLite automatically creates a new database if it doesn't already exist.

Connecting to a Database Use the sqlite3.connect() method to establish a connection to the database.

Executing SQL Queries Use a cursor object to execute SQL commands.

Committing Transactions After making changes to the database, commit the transaction to save changes.

Closing the Connection Always close the connection after performing operations.

* Creating and executing SQL queries from Python using these connectors.
* SQLite3 is part of the Python standard library, and you don't need to install any external package to use it. Below is an example of how to create a database, execute queries, and interact with SQLite using Python.

10. Search and Match Functions

* Using re.search() and re.match() functions in Python’s re module for pattern matching.
* Both search() and match() are methods provided by the re (regular expression) module. They are used for pattern matching, but they differ in how they search through the text. Here's the key difference between the two:

re.match():

Behavior: match() checks for a match only at the beginning of the string.

Use Case: It returns a match object if the regular expression pattern matches at the start of the string. If the pattern doesn't match the beginning of the string, it returns None.

* Difference between search and match.
* Search Scope

re.search() − Scours the entire input string in search of the pattern.

re.match() − Restricts its search to the beginning of the input string.

Position of Match

re.search() − finds the first occurrence of the pattern, regardless of its position.

re.match() − Permits a match only if the pattern is discovered at the sacred start of the input string.

Return Value

re.search() − returns a match object if the pattern is found; else, returns None.

re.match() − returns a match object if the pattern is found at the start; else, returns None.

Performance

re.search() − Undertakes a thorough search of the entire input string, potentially affectingperformance for extensive strings.

re.match() − Exhibits efficiency by focusing on matching the pattern only at the sacred beginning of the string.