**Advance Python Programming**

**1. Printing on Screen**

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

* The print() function in Python is a built-in function used to display output to the console or standard output device. It is a fundamental tool for programmers to communicate with users, debug code, and present results during program execution.
* **Basic Usage:**

The simplest way to use print() is to pass a string literal as an argument:

* **Python**

print("Hello, World!")

This will output:

* **Code**

Hello, World!

* **Printing Variables and Expressions:**

You can also print the values of variables or the results of expressions:

* **Python**

name = "Alice"

age = 30

print("Name:", name)

print("Age:", age \* 2)

* **This will output:**
* **Code**

Name: Alice

Age: 60

Multiple Arguments and Separators:

The print() function can accept multiple arguments, separated by commas. By default, these arguments are separated by a single space:

* **Python**

print("One", "Two", "Three")

* **This will output:**
* **Code**

One Two Three

* You can customize the separator using the sep parameter:
* **Python**

print("Apple", "Banana", "Cherry", sep="-")

* **This will output:**

Apple-Banana-Cherry

* **Controlling the End of the Output:**

By default, print() adds a newline character (\n) at the end of the output, causing subsequent print() calls to start on a new line. You can change this behavior using the end parameter:

* **Python**

print("This is on one line.", end=" ")

print("This continues on the same line.")

* **This will output:**

This is on one line. This continues on the same line.

**2. Formatting outputs using f-strings and format().**

* Python provides robust mechanisms for formatting output, primarily through f-strings (formatted string literals) and the format() method. Both offer powerful ways to embed variables and expressions into strings and control their presentation.
* **F-Strings (Formatted String Literals)**

F-strings, introduced in Python 3.6, are a concise and readable way to embed expressions inside string literals. They are identified by an f or F prefix before the opening quote of the string.

* **Python**

name = "Alice"  
age = 30  
print(f"My name is {name} and I am {age} years old.")  
# Output: My name is Alice and I am 30 years old.

* **Key features of f-strings:**

Direct embedding: Variables and expressions are directly placed within curly braces {}.

Expression evaluation: Any valid Python expression can be included within the curly braces, and its result will be inserted into the string.

Format specifiers: Detailed formatting options (like decimal precision, padding, alignment) can be applied using a colon : followed by the specifier within the curly braces.

* **Python**

pi = 3.14159265  
print(f"The value of pi is approximately {pi:.2f}")   
number = 1234567  
print(f"Large number with comma separator: {number:,}")  
The format() Method

The format() method is a string method that provides a flexible way to format strings using placeholders.

* **Python**

name = "Bob"  
age = 25  
print("My name is {} and I am {} years old.".format(name, age))

Key features of format():

Positional arguments: Placeholders {} are filled in the order of arguments passed to format().

Named arguments: Placeholders can benamed, and corresponding named arguments are passed to format().

Format specifiers: Similar to f-strings, format specifiers can be used within the placeholders.

* **Python**

print("The value of pi is approximately {:.3f}".format(3.14159265))  
print("Hello, {name}! You are {age}.".format(name="Charlie", age=40))

**2. Reading Data from Keyboard**

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

* The input() function in Python is used to read user input from the keyboard.
* **Basic Usage:**
* **Prompt the User:**

The input() function can take an optional string argument, which serves as a prompt displayed to the user before they enter their input.

* **Read Input:**

When input() is called, the program pauses execution and waits for the user to type something and press the Enter key.

* **Return as String:**

The input() function always returns the user's input as a string, regardless of the data type entered by the user (e.g., numbers, text).

* **Python**  
  name = input("Enter your name: ")  
  print("Hello,", name)  
  age\_str = input("Enter your age: ")  
  age\_int = int(age\_str)  
  print("You are", age\_int, "years old.")

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

* Converting user input to different data types is a common task in programming. This process is known as type casting or type conversion. Most programming languages provide built-in functions or mechanisms to perform these conversions.
* Here's how you can convert user input into different data types, using Python as an example:
* **1. Converting to Integer (int)**

User input is initially read as a string. To perform arithmetic operations or treat it as a whole number, you need to convert it to an integer.

* **Python**

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

try:

user\_input\_int = int(user\_input\_str)

print(f"You entered the integer: {user\_input\_int}")

except ValueError:

print("Invalid input. Please enter a valid integer.")

* **2. Converting to Float (float)**

If the user is expected to enter a decimal number, you convert the input string to a float.

* **Python**

user\_input\_str = input("Enter a decimal number: ")

try:

user\_input\_float = float(user\_input\_str)

print(f"You entered the float: {user\_input\_float}")

except ValueError:

print("Invalid input. Please enter a valid decimal number.")

* **3. Converting to String (str)**

While user input is already a string, you might explicitly convert other data types (like numbers) to strings for concatenation or specific string manipulations.

* **Python**

number = 123

number\_as\_str = str(number)

print(f"The number as a string is: {number\_as\_str}")

* **4. Converting to Boolean (bool)**

You can convert user input or other values to a boolean (True/False). In Python, non-empty strings and non-zero numbers evaluate to True, while empty strings and zero evaluate to False.

* **Python**

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

user\_input\_bool = bool(user\_input\_str)

print(f"The boolean value of your input is: {user\_input\_bool}")

number = 0

number\_as\_bool = bool(number)

print(f"The boolean value of 0 is: {number\_as\_bool}")

**3. Opening and Closing Files**

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

* File modes in programming languages like Python and C define how a file is opened and what operations can be performed on it. The common modes are:
* **'r' (Read Mode):**
* Opens a file for reading.
* The file pointer is positioned at the beginning of the file.
* If the file does not exist, an error is raised.
* **Python**

try:  
 with open("example.txt", "r") as f:  
 content = f.read()  
 print(content)  
 except FileNotFoundError:  
 print("File not found.")

* **'w' (Write Mode):**
* Opens a file for writing.
* If the file exists, its contents are truncated (deleted) before writing.
* If the file does not exist, a new file is created.
* **Python**

with open("example.txt", "w") as f:  
 f.write("This is new content.")

* **'a' (Append Mode):**
* Opens a file for writing, appending new data to the end of the file.

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

The file pointer is positioned at the end of the file.

* **Python**

with open("example.txt", "a") as f:  
 f.write("\nThis content is appended.")

* **'r+' (Read and Write Mode):**
* Opens a file for both reading and writing.
* The file must exist; otherwise, an error is raised.
* The file pointer is positioned at the beginning of the file.
* Writing will overwrite existing content from the current file pointer position.
* **Python**

try:  
 with open("example.txt", "r+") as f:  
 content = f.read()  
 print("Original content:", content)  
 f.seek(0)

f.write("Overwritten text.")  
 except FileNotFoundError:  
 print("File not found.")

* **'w+' (Write and Read Mode):**
* Opens a file for both reading and writing.
* If the file exists, its contents are truncated (deleted) before any operation.
* If the file does not exist, a new file is created.
* The file pointer is positioned at the beginning of the file.
* **Python**

with open("example.txt", "w+") as f:  
 f.write("Content for w+.")  
 f.seek(0)

print("Content after w+ write:", content)

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

* The open() function in Python is used to create and access files. It returns a file object, which can then be used to perform various file operations like reading, writing, and appending.
* Syntax:
* **Python**

file\_object = open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

* **Key Parameters:**

file: The path to the file you want to open or create.

mode: A string specifying the mode in which the file is opened. Common modes include:

'r' (read): Opens for reading (default).

'w' (write): Opens for writing. Creates the file if it doesn't exist, or truncates (empties) it if it does.

'a' (append): Opens for appending. Creates the file if it doesn't exist. If the file exists, new data is written to the end.

'x' (exclusive creation): Creates a new file and opens it for writing. Raises an error if the file already exists.

't' (text): Opens in text mode (default).

'b' (binary): Opens in binary mode.

You can combine modes, e.g., 'rt' for reading text, 'wb' for writing binary.

Creating and Writing to a File:

To create and write to a file, use the 'w' mode.

* **Python**  
  with open("my\_file.txt", "w") as f:  
   f.write("Hello, this is a new line.\n")  
   f.write("This is another line.\n")
* **Reading from a File:**

To read from a file, use the 'r' mode.

* **Python**  
  with open("my\_file.txt", "r") as f:  
   content = f.read()  
   print(content)  
    
  with open("my\_file.txt", "r") as f:  
   for line in f:  
   print(line.strip())

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

* Closing a file using close() is a crucial step in file handling, particularly in programming languages like Python. The close() method is invoked on a file object after all necessary operations (reading, writing, etc.) have been completed.
* **Purpose of close():**

Resource Management:

Files are system resources. Leaving them open consumes system resources and can lead to issues like exceeding the operating system's limit on open files.

* **Data Integrity:**

When writing to a file, changes might be buffered in memory and not immediately written to the disk. Calling close() ensures that all buffered data is flushed to the file, preventing data loss or corruption.

* **Preventing Conflicts:**

Closing a file releases any locks on it, allowing other programs or processes to access and modify the file without encountering conflicts.

* **Python**

file\_object.close()

* **Python**  
  file = open("my\_document.txt", "w")  
  file.write("This is some text.")  
  file.close()
* Recommended Practice: Using with Statement (Context Manager):
* While close() directly achieves file closure, the preferred and safer method in Python is to use the with statement, which acts as a context manager. This automatically handles file closure, even if errors occur during file operations.
* **Python**

with open("my\_document.txt", "w") as file:  
 file.write("This is some text.")

**4. Reading and Writing Files**

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

* In Python, the read(), readline(), and readlines() methods are used to read content from a file object.
* **Read():**

The read() method reads the entire content of a file and returns it as a single string. An optional argument size can be provided to specify the number of characters or bytes to read from the file. If size is omitted, the entire file content is read.

* **Python**

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

content = file.read()

print(content)

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

first\_10\_chars = file.read(10)

print(first\_10\_chars)

* **Readline()**

The readline() method reads a single line from the file, including the newline character (\n) at the end, and returns it as a string. Each subsequent call to readline() reads the next line in the file. If the end of the file is reached, an empty string is returned.

* **Python**

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

line1 = file.readline()

line2 = file.readline()

print(line1)

print(line2)

* **Readlines()**

The readlines() method reads all lines from the file and returns them as a list of strings. Each element in the list represents a line from the file, including the newline character at the end of each line.

* **Python**

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

all\_lines = file.readlines()

for line in all\_lines:

print(line, end='')

**2. Writing to a file using write() and writelines().**

* In Python, the write() and writelines() methods are used to write data to a file. Both methods require the file to be opened in a write mode ('w'), append mode ('a'), or exclusive creation mode ('x'). Using a with statement is recommended for proper file handling, as it ensures the file is automatically closed.
* **Write() method**

The write() method is used to write a single string to a file. It takes one argument, which must be a string. It returns the number of characters written.

* **Python**

with open("my\_file.txt", "w") as f:

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

f.write("This is the second line.")

* **Writelines() method**

The writelines() method is used to write multiple strings to a file. It takes an iterable (like a list or tuple) of strings as its argument. Each string in the iterable is written to the file, but writelines() does not automatically add newline characters. You must include newline characters (\n) within the strings themselves if you want them to appear on separate lines in the file.

* **Python**

lines\_to\_write = ["Line 1 with writelines.\n", "Line 2 with writelines.\n", "Line 3 with writelines."]

with open("my\_file.txt", "a") as f:

f.writelines(lines\_to\_write)

**5. Exception Handling**

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

* **Introduction to Exceptions**

Exceptions are events that disrupt the normal flow of a program's execution. They are signals that something unexpected or erroneous has occurred, such as dividing by zero, trying to access a non-existent file, or attempting to use an undefined variable. When an exception occurs, if not handled, it can lead to program termination. Exception handling provides a structured way to gracefully manage these errors, preventing crashes and allowing the program to recover or provide informative feedback.

Handling Exceptions with try, except, and finally

Python uses try, except, and finally blocks to manage exceptions.

* **Try block:** This block contains the code that might potentially raise an exception. The interpreter attempts to execute the statements within this block.
* **Python**

try:  
 result = 10 / 0

* except block: If an exception occurs within the try block, the control flow immediately transfers to the corresponding except block. This block contains the code to handle the specific exception (or a general exception if no specific type is mentioned). You can have multiple except blocks to handle different types of exceptions.
* **Python**

try:  
 result = 10 / 0  
 except ZeroDivisionError:  
 print("Error: Cannot divide by zero!")  
 except Exception as e:

print(f"An unexpected error occurred: {e}")

* **finally block:** This block contains code that will always execute, regardless of whether an exception occurred in the try block or not. It is typically used for cleanup operations, such as closing files or releasing resources, ensuring these actions happen even if an error disrupts the main process.
* **Python**

try:  
 file = open("my\_file.txt", "r")  
 content = file.read()  
 except FileNotFoundError:  
 print("Error: File not found.")  
 finally:  
 if 'file' in locals() and not file.closed: file.close()  
 print("File closed.")

**2. Understanding multiple exceptions and custom exceptions.**

* **Multiple catch blocks:**

This involves having a separate catch block for each specific exception type you want to handle. The order of these blocks matters, as the first matching catch block will be executed. It is generally recommended to place more specific exception types before more general ones.

* **Java**

try {  
} catch (ArithmeticException e) {  
 System.out.println("Arithmetic error: " + e.getMessage());  
} catch (ArrayIndexOutOfBoundsException e) {  
 System.out.println("Array index out of bounds: " + e.getMessage());  
} catch (Exception e) {

System.out.println("General error: " + e.getMessage());  
}

* **Single catch block for multiple exceptions (Java 7+):**

Java 7 and later versions allow you to catch multiple exception types in a single catch block using the | operator. This is useful when the handling logic for these exceptions is similar.

* **Java**

try {  
} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {  
 System.out.println("An arithmetic or array index error occurred: " + e.getMessage());  
}

* **Understanding Custom Exceptions**

Custom exceptions, also known as user-defined exceptions, allow you to create your own specific error types that are tailored to the logic of your application. This provides more meaningful and context-specific error messages and better control over error handling.

* **Creating Custom Exceptions:**

To create a custom exception, you typically define a new class that extends a built-in exception class (e.g., Exception or RuntimeException in Java, Exception in Python). This custom class can include constructors and methods to provide detailed information about the error.

* **Java**  
  class InvalidAgeException extends Exception {  
   public InvalidAgeException(String message) {  
   super(message);  
   }  
  }  
  public class VotingSystem {  
   public void checkAge(int age) throws InvalidAgeException {  
   if (age < 18) {  
   throw new InvalidAgeException("User is not eligible to vote: age is less than 18.");  
   }  
   System.out.println("User is eligible to vote.");  
   }  
  }
* **Python**

class CustomError(Exception):  
 """Custom exception for specific error conditions."""  
 def \_\_init\_\_(self, message="A custom error occurred."):  
 self.message = message  
 super().\_\_init\_\_(self.message)  
def process\_data(value):  
 if value < 0:  
 raise CustomError("Negative values are not allowed.")  
print(f"Processing value: {value}")

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

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

* **Class:**

A class serves as a blueprint or a template for creating objects. It defines the structure and behavior that its instances (objects) will possess. Think of it as a cookie cutter: the class defines the shape, but you can make many individual cookies (objects) from it.

* **Python**

class Dog:  
 pass

* **Object:**

An object is an instance of a class. It is a concrete entity created from the class blueprint, possessing its own unique set of data (attributes) and capabilities (methods). You can create multiple objects from a single class, each with potentially different attribute values.

* **Python**

my\_dog = Dog()

your\_dog = Dog()

* **Attributes:**

Attributes are variables associated with a class or an object, representing its characteristics or data.

Class attributes

are shared by all instances of the class and are defined directly within the class body.

Instance attributes

are unique to each object and are typically defined within the \_\_init\_\_ method using self.

* **Python**

class Dog:  
 species = "Canis familiaris"   
 def \_\_init\_\_(self, name, breed):  
 self.name = name   
 self.breed = breed   
my\_dog = Dog("Buddy", "Golden Retriever")  
print(my\_dog.name)   
print(my\_dog.species)

* **Methods:**

Methods are functions defined within a class that describe the behaviors or actions an object can perform. They operate on the object's data (attributes) and are called using dot notation on an object. Methods usually take self as their first parameter, which refers to the instance itself.

* **Python**

class Dog:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def bark(self):   
 print(f"{self.name} says Woof!")  
my\_dog = Dog("Max")  
my\_dog.bark()

**2. Difference between local and global variables.**

* The main difference is scope: local variables are only accessible within the function or block where they are declared, while global variables are accessible from anywhere in the program. This also affects their lifetime: local variables are created when the function is called and destroyed when it exits, whereas global variables are created when the program starts and persist until it ends.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Local Variable** | **Global Variable** |
| Scope | Limited to the function or block where it is defined | Accessible from any part of the program |
| Declaration | Inside a function or block | Outside of any function |
| Lifetime | Exists only during the execution of its function or block | Exists for the entire duration of the program's execution |
| Memory | Allocated when the function is called and deallocated when it returns | Allocated when the program starts and deallocated when it ends |
| Purpose | Storing temporary data within a specific function | Storing data that needs to be shared across multiple functions |

**7. Inheritance**

**1. Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.**

* Python supports various types of inheritance, enabling code reusability and establishing relationships between classes. The primary types are:
* **Single Inheritance:**

A derived class inherits from only one base class.

This is the simplest form, where a child class gains all attributes and methods of a single parent.

* **Python**

class Parent:  
 def func1(self):  
 print("This is from Parent")  
 class Child(Parent):  
 def func2(self):  
 print("This is from Child")

* **Multilevel Inheritance:**

A derived class inherits from another derived class, creating a chain of inheritance.

Features of the base class and the intermediate derived class are inherited into the final derived class.

* **Python**

class Grandparent:  
 def func\_grand(self):  
 print("This is from Grandparent")  
 class Parent(Grandparent):  
 def func\_parent(self):  
 print("This is from Parent")  
 class Child(Parent):  
 def func\_child(self):  
 print("This is from Child")

* **Multiple Inheritance:**

A derived class inherits from multiple base classes.

This allows combining functionalities from several parent classes into a single child class. Python's Method Resolution Order (MRO) handles potential conflicts when methods with the same name exist in different parent classes.

* **Python**

class Father:  
 def skill\_father(self):  
 print("Father's skill")  
 class Mother:  
 def skill\_mother(self):  
 print("Mother's skill")  
 class Child(Father, Mother):  
 def skill\_child(self):  
 print("Child's skill")

* **Hierarchical Inheritance:**

Multiple derived classes inherit from a single base class.

Each derived class shares the base class's attributes and methods but can also have its unique functionalities.

* **Python**

class Animal:  
 def eat(self):  
 print("Animal is eating")  
 class Dog(Animal):  
 def bark(self):  
 print("Dog is barking")  
 class Cat(Animal):  
 def meow(self):  
 print("Cat is meowing")

* **Hybrid Inheritance:**

A combination of two or more types of inheritance (e.g., combining multiple and multilevel inheritance).

This creates more complex class hierarchies, allowing for flexible code organization and reuse.

* **Python**

class A:  
 def method\_a(self):  
 print("Method A")  
 class B(A):  
 def method\_b(self):  
 print("Method B")  
 class C:  
 def method\_c(self):  
 print("Method C")  
 class D(B, C):

def method\_d(self):  
 print("Method D")

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

* The super() function in Python provides a way to access methods and properties of a parent or superclass from a child or subclass. This is particularly useful in object-oriented programming when dealing with inheritance and method overriding.
* **Accessing Parent Class Properties:**

While super() is primarily used to call parent class methods, especially the \_\_init\_\_ constructor, it can indirectly help in accessing properties. When super().\_\_init\_\_() is called in a child class's constructor, it ensures that the parent class's initialization logic is executed, which often involves setting up attributes (properties) defined in the parent.

* **Python**

class Parent:

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

self.name = name

class Child(Parent):

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

super().\_\_init\_\_(name) # Calls Parent's \_\_init\_\_ to set 'name'

self.age = age

def display\_info(self):

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

child\_obj = Child("Alice", 10)

print(child\_obj.name)

child\_obj.display\_info()

* **In this example:**

The Parent class has an \_\_init\_\_ method that sets a name attribute.

The Child class inherits from Parent. Its \_\_init\_\_ method calls super().\_\_init\_\_(name), which effectively runs the Parent class's constructor, thereby initializing the self.name attribute for the Child object.

The Child class can then access self.name because it was properly set during the parent's initialization.

**8. Method Overloading and Overriding**

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

* Method overloading is a feature in object-oriented programming languages that allows a class to have multiple methods with the same name, but with different parameters. This is a form of compile-time polymorphism, where the specific method to be invoked is determined at compile time based on the method's signature.
* **The "signature" of a method includes:**

The number of parameters: Methods with the same name can have a different count of arguments.

The data types of parameters: Methods with the same name can accept arguments of different data types.

The order of parameters: Methods with the same name can have parameters of the same data types but in a different order.

* Key characteristics of method overloading:
* **Increased readability:**

It allows you to perform similar operations using a consistent method name, making the code more intuitive and easier to understand.

* **Flexibility:**

It enables a single method name to handle various input types or quantities, adapting to different scenarios without requiring distinct method names for each.

* **Return type is not a factor:**

Method overloading is determined solely by the parameter list (number, type, and order). The return type of the methods does not play a role in distinguishing overloaded methods.

* **Compile-time polymorphism:**

The compiler determines which overloaded method to call based on the arguments provided at the time of invocation.

* **Example in Java:**

public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
 public int add(int a, int b, int c) {  
 return a + b + c;  
 }  
 public double add(double a, double b) {  
 return a + b;  
 }  
 public static void main(String[] args) {  
 Calculator calc = new Calculator();  
System.out.println(calc.add(5, 10)); System.out.println(calc.add(5, 10, 15)); System.out.println(calc.add(5.5, 10.2));

}  
}

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

* Method overriding is a core concept in object-oriented programming where a subclass (child class) provides its own specific implementation for a method that is already defined in its superclass (parent class). This means that a method in the child class has the same name, return type (or a compatible type), and parameters (signature) as a method in the parent class.
* Here's a breakdown of method overriding:
* **Inheritance is Required:**

Method overriding can only occur when there's an inheritance relationship between classes, meaning a child class extends or inherits from a parent class.

* **Same Signature:**

The method in the child class must have the exact same name and parameter list as the method in the parent class. In some languages, the return type must also be the same or a covariant return type (a more specific type).

* **Different Implementation:**

The purpose of overriding is to provide a specialized implementation of the method in the child class that is different from the parent class's implementation.

* **Runtime Polymorphism:**

Method overriding is a key mechanism for achieving runtime polymorphism (also known as dynamic method dispatch or late binding). This means that the specific method implementation that gets executed is determined at runtime based on the actual type of the object, not the type of the reference variable.

* **super() Keyword (in some languages):**

In many languages like Java and Python, the super() keyword can be used within an overridden method in the child class to call and execute the parent class's version of the method. This is useful when the child class wants to extend or modify the parent's behavior rather than completely replacing it.

* **Restrictions:**

Some methods cannot be overridden. For example, in Java, final methods cannot be overridden, and static methods are considered "hidden" rather than overridden when a child class defines a static method with the same signature.

**9. SQLite3 and PyMySQL (Database Connectors)**

**1. Introduction to SQLite3 and PyMySQL for database connectivity.**

* **SQLite3**

SQLite3 is a lightweight, serverless, and self-contained relational database engine that stores data in a single file. It is ideal for small-scale applications, prototyping, testing, or embedded systems. The sqlite3 module is part of Python's standard library, meaning no additional installation is required.

* **Key features:**

File-based: Databases are stored in a single file, making them easy to manage and transport.

Serverless: No separate server process is required, simplifying deployment.

Built-in: Included with Python, eliminating the need for external installations.

* Example of basic usage:
* **Python**

import sqlite3  
conn = sqlite3.connect('example.db')  
cursor = conn.cursor()  
cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

email TEXT UNIQUE

)

''')  
cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Alice', 'alice@example.com'))  
conn.commit()  
conn.close()

* **PyMySQL**

PyMySQL is a pure-Python MySQL client library that allows Python applications to connect and interact with MySQL databases. Unlike SQLite, MySQL is a powerful, server-based relational database management system often used for larger, more complex applications such as web applications and e-commerce platforms. PyMySQL requires installation using pip.

* **Key features:**

Server-based: Connects to a separate MySQL server.

Full-featured: Supports advanced MySQL features like transactions and concurrency.

External library: Requires installation via pip.

Example of basic usage:

* **Python**

import pymysql  
try:  
 conn = pymysql.connect(  
 host='localhost',  
 user='root',  
 password='your\_password',  
 database='testdb'  
 )  
 cursor = conn.cursor()  
 cursor.execute('''

CREATE TABLE IF NOT EXISTS products (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

price DECIMAL(10, 2)

)

''')  
 cursor.execute("INSERT INTO products (name, price) VALUES (%s, %s)", ('Laptop', 1200.00))  
 conn.commit()  
 conn.close()  
except pymysql.Error as e:  
 print(f"Error connecting to MySQL: {e}")

**2. Creating and executing SQL queries from Python using these connectors.**

* Creating and executing SQL queries from Python using database connectors involves several key steps:
* **Install the appropriate connector library:** Install the Python library specific to your database (e.g., mysql-connector-python for MySQL, psycopg2 for PostgreSQL, pyodbc for SQL Server, sqlite3 which is built-in for SQLite).
* **Code**

pip install mysql-connector-python

Import the library: Import the necessary module into your Python script.

* **Python**

import mysql.connector

Establish a database connection: Use the connector's connect() method to establish a connection to your database, providing credentials like host, user, password, and database name.

* **Python**

connection = mysql.connector.connect(  
 host="localhost",  
 user="your\_user",  
 passwd="your\_password",  
 database="your\_database"  
 )

* **Create a cursor object:** A cursor object acts as an intermediary, allowing you to execute SQL queries and fetch results.
* **Python**

cursor = connection.cursor()

Write and execute SQL queries: Define your SQL query as a string and execute it using the cursor's execute() method.

* **Python**

sql\_query = "SELECT \* FROM your\_table WHERE column\_name = %s"  
 value = "some\_value"  
 cursor.execute(sql\_query, (value,)) # Use parameterized queries for security

* **Fetch results (for SELECT queries):** If your query retrieves data, use methods like fetchone(), fetchall(), or fetchmany() to retrieve the results.
* **Python**

results = cursor.fetchall()  
 for row in results:  
 print(row)

* **Commit changes (for DDL/DML operations):** For operations that modify the database (e.g., INSERT, UPDATE, DELETE, CREATE TABLE), you need to commit the changes to make them permanent.
* **Python**

connection.commit()

Close the cursor and connection: Release database resources by closing the cursor and then the connection when you are finished.

* **Python**

cursor.close()  
 connection.close()

**10. Search and Match Functions**

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

* The re module in Python provides tools for working with regular expressions, including re.search() and re.match() for pattern matching.
* **1. re.match(pattern, string, flags=0)**

re.match() attempts to match the pattern only at the beginning of the string.

If the pattern is found at the start of the string, it returns a Match object.

If the pattern is not found at the beginning, it returns None.

* **Python**

import re  
text1 = "Hello, world!"  
text2 = "world, Hello!"   
match\_obj1 = re.match("Hello", text1)  
if match\_obj1:  
 print(f"re.match() found: {match\_obj1.group()}")  
else:  
 print("re.match() did not find 'Hello' in text1.")  
match\_obj2 = re.match("Hello", text2)  
if match\_obj2:  
 print(f"re.match() found: {match\_obj2.group()}")  
else:  
 print("re.match() did not find 'Hello' in text2.")

* **Output:**

re.match() found: Hello  
re.match() did not find 'Hello' in text2.

* **2. re.search(pattern, string, flags=0)**

re.search() scans the entire string for the first location where the pattern produces a match.

If a match is found anywhere in the string, it returns a Match object.

If no match is found, it returns None.

* **Python**

import re  
text1 = "Hello, world!"  
text2 = "world, Hello!"  
search\_obj1 = re.search("world", text1)  
if search\_obj1:  
 print(f"re.search() found: {search\_obj1.group()}")  
else:  
 print("re.search() did not find 'world' in text1.")  
search\_obj2 = re.search("Hello", text2)  
if search\_obj2:  
 print(f"re.search() found: {search\_obj2.group()}")  
else:  
 print("re.search() did not find 'Hello' in text2.")

* **Output:**

re.search() found: world  
re.search() found: Hello

**2. Difference between search and match.**

* The terms "search" and "match" can have different meanings depending on the context, but in the realm of string manipulation and regular expressions, they refer to distinct operations:
* **1. Scope of Operation:**

Match:

A "match" operation typically attempts to find a pattern only at the beginning of a string. If the pattern is not found at the very start, the operation fails.

Search:

A "search" operation scans the entire string to find the first occurrence of a pattern anywhere within it.

* **2. Return Value:**

Match (in some contexts, e.g., JavaScript's match()):

Can return an array containing all occurrences of the pattern found in the string, or information about the match.

Search (in some contexts, e.g., Python's re.search() or JavaScript's search()):

Often returns the index of the first match, or a match object containing details about the first match found.

* **Python**

import re  
text = "hello world"  
pattern = "world"   
match\_result = re.match(pattern, text)   
print(f"re.match() result: {match\_result}")   
search\_result = re.search(pattern, text)  
print(f"re.search() result: {search\_result}")

* **Output:**

re.match() result: None  
re.search() result: <re.Match object; span=(6, 11), match='world'>