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 other standard output devices. It is a fundamental tool for debugging and presenting information to the user.
* Basic Usage
* The most basic use of the print() function is to pass it a string, and it will print that string to the console.
* print("Hello, World!") # Output: Hello, World!
* Printing Multiple Items
* The print() function can accept multiple arguments, which will be printed on the same line, separated by a space by default.
* print("Hello", "how", "are", "you?")The print() function in Python is used to display output on the screen. Example:
* print("Hello, World!")

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

* Both f-strings and the format() method are used for string formatting in Python, but they offer different approaches and have varying levels of readability and flexibility.
* f-strings (Formatted String Literals):Introduced in Python 3.6, f-strings provide a concise and readable way to embed expressions inside string literals.
* They are prefixed with an f or F before the opening quote.
* Expressions are placed inside curly braces {} within the string.

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.

print(f"The price is {19.99:.2f} dollars") # Output: The price is 19.99 dollars

format() Method:

The format() method is a string method available since Python 2.6.

It uses curly braces {} as placeholders in the string, similar to f-strings.

Values to be inserted are passed as arguments to the format() method.

It supports both positional and named arguments for more flexibility.

The format() method also supports format specifiers, similar to f-strings.

name = "Bob"

score = 120

print("Player: {}, Score: {}".format(name, score)) # Output: Player: Bob, Score: 120

print("The value of pi is roughly {0:.3f}".format(3.14159265)) # Output: The value of pi is roughly 3.142

print("Coordinates: {latitude}, {longitude}".format(latitude="37.7749", longitude="-122.4194"))

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

* The input() function lets you take input from the keyboard as a string.

name = input("Enter your name: ")

* print("Hello,", name)
* Output:

Enter your name: Alice

Hello, Alice

Reading Numbers:

By default, input() returns a string, so you need to convert it:

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

print("You will be", age + 1, "next year.")

* Multiple Inputs:

x, y = input("Enter two numbers separated by space: ").split()

x = int(x)

y = int(y)

* print("Sum:", x + y)

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

* Converting User Input into Different Data Types in Python
* The input() function always returns a string, so you need to convert it when working with numbers or other types.
* 1. Convert to Integer

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

* print("In 5 years, you’ll be", age + 5)
* 2. Convert to Float
* price = float(input("Enter the price: "))
* print("Half price is", price / 2)
* 3. Convert to Boolean

answer = input("Do you agree? (yes/no): ")

* is\_agree = answer.lower() == "yes"
* print("Agreed?", is\_agree)
* 4. Convert to List (Splitting input)

numbers = input("Enter numbers separated by spaces: ").split()

numbers = [int(num) for num in numbers]

print("Sum:", sum(numbers))

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

* Introduction to Exceptions in Python
* An exception is an error that occurs during program execution. Instead of crashing your program, you can handle exceptions gracefully using try, except, and finally.
* 1. Basic Try-Except Structure
* try:

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

result = 10 / x

except ZeroDivisionError:

print("Cannot divide by zero!")

except ValueError:

print("Please enter a valid number.")

* If the code in try raises an error, control jumps to the matching except block.
* You can handle specific exceptions like ZeroDivisionError or ValueError.

2. Using finally

The finally block runs no matter what — whether an exception occurs or not.

try:

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

print(f.read())

except FileNotFoundError:

print("File not found.")

finally:

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

try:

f.close()

except:

Pass

3. Catching Any Exception (Not recommended for debugging)

try:

risky\_code()

except Exception as e:

print("An error occurred:", e)

1. Would you like examples for raising custom exceptions or using else in exception handling?Understanding multiple exceptions and custom exceptions.

* In Python, exceptions are events that disrupt the normal flow of a program. They are raised when errors occur during execution. Python provides a mechanism to handle these exceptions using try, except, else, and finally blocks.
* Handling Multiple Exceptions
* A single try block can be followed by multiple except blocks, each designed to handle a specific type of exception. This allows for different error handling strategies based on the type of exception that occurs.
* Python
* try:

# Code that might raise an exception

result = 10 / 0 # This will raise ZeroDivisionError

* except ZeroDivisionError:

print("Cannot divide by zero.")

except TypeError:

print("Type error occurred.")

except Exception as e:

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

* Custom Exceptions
* Python allows users to create their own custom exceptions by defining new classes that inherit from the built-in Exception class or one of its subclasses. This is useful for handling specific error conditions that are unique to your application.
* class InvalidInputError(Exception):
* def \_init\_(self, message):

self.message = message

super().\_init\_(self.message)

def process\_data(data):

if not isinstance(data, int):

raise InvalidInputError("Input must be an integer.")

# Process the data

return data

* try:

result = process\_data("abc")

except InvalidInputError as e:

* print(f"Error: {e}")
* In this example, InvalidInputError is a custom exception. The process\_data function raises this exception if the input is not an integer. The try block attempts to process the data, and the except block catches the custom exception, printing the error message.
* else and finally blocks
* The else block is executed if no exceptions are raised in the try block.
* The finally block is always executed, regardless of whether an exception occurred or not. It is typically used for cleanup tasks and releasing resources.

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

* Here's a breakdown of classes, objects, attributes, and methods in Python:

Classes:

A class acts as a blueprint for creating objects.

It defines the structure and behavior that objects of that class will have.

Think of it as a template for creating specific kinds of things.

* Objects:

An object is an instance of a class.

It's a concrete entity created based on the class blueprint.

Each object has its own unique set of data and can perform actions defined by its class.

Attributes:

These are variables that store data associated with an object.

They represent the characteristics or properties of an object.

Attributes can be specific to each object (instance attributes) or shared by all objects of the class (class attributes).

Methods:

Methods are functions defined within a class.

They define the actions that objects of that class can perform.

Methods operate on the object's data (attributes) and can modify the object's state.

1. Difference between local and global variables.

* Local and global variables differ primarily in their scope (accessibility) and lifetime. Local variables are defined within a specific block of code (like a function), making them accessible only within that block. Global variables are defined outside any block, giving them accessibility throughout the entire program.

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

* Inheritance is a mechanism in object-oriented programming that allows a class to inherit properties and methods from another class. Here is an overview of the different types of inheritance in Python:
* Single Inheritance: A child class inherits from only one parent class. This is the most basic form of inheritance and creates a simple hierarchy.
* class Parent:

def method\_a(self):

print("Method A from parent class")

* class Child(Parent):

def method\_b(self):

print("Method B from child class")

* child\_obj = Child()

child\_obj.method\_a() # Output: Method A from parent class

child\_obj.method\_b() # Output: Method B from child class

* Multilevel Inheritance: A class inherits from another class, which in turn inherits from another class. This creates a chain of inheritance.

class Grandparent:

* def method\_a(self):
* print("Method A from grandparent class")
* class Parent(Grandparent):
* def method\_b(self):
* print("Method B from parent class")
* class Child(Parent):

def method\_c(self):

print("Method C from child class")

child\_obj = Child()

child\_obj.method\_a() # Output: Method A from grandparent class

child\_obj.method\_b() # Output: Method B from parent class

child\_obj.method\_c() # Output: Method C from child class

Multiple Inheritance: A class inherits from multiple parent classes. This allows a class to combine features from different sources.

class Parent1:

* def method\_a(self):

print("Method A from parent 1 class")

* class Parent2:

def method\_b(self):

print("Method B from parent 2 class")

* class Child(Parent1, Parent2):
* def method\_c(self):
* print("Method C from child class")

child\_obj = Child()

child\_obj.method\_a() # Output: Method A from parent 1 class

child\_obj.method\_b() # Output: Method B from parent 2 class

* child\_obj.method\_c() # Output: Method C from child class
* Hierarchical Inheritance: Multiple classes inherit from a single parent class. This creates a tree-like structure where several child classes share common features.

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

* The super() function is a built-in function in Python that allows you to access methods and properties of a parent class from within a child class. It is particularly useful when dealing with inheritance, enabling you to extend or modify the behavior of parent classes while still leveraging their existing functionality.
* Accessing Parent Class Methods:
* When a child class inherits from a parent class, it can override methods defined in the parent class.
* To call the overridden method in the parent class from the child class, you use super(). This allows you to execute the parent's method before or after executing the child's method.
* Accessing Parent Class Properties:
* While super() is primarily used for methods, it can also indirectly access parent class properties.
* If you have a method in the parent class that accesses a property, youcan call that method using super() to access the property.

class Parent:

def \_init\_(self, name):

self.name = name

def display(self):

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

class Child(Parent):

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

super().\_init(name) # Call parent's \_\_init\_

self.age = age

def display(self):

super().display() # Call parent's display method

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

child = Child("Alice", 10)

child.display()

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

* Method overloading is a powerful tool in object-oriented programming that promotes code reusability and clarity. It allows developers to create methods with the same name that perform different operations depending on the input parameters

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

* Method overriding is a feature in object-oriented programming that allows a subclass to provide a specific implementation of a method that is already defined in its parent class. This enables a child class to modify or extend the behavior of an inherited method while maintaining the same method signature.

1. Introduction to SQLite3 and PyMySQL for database connectivity.

* After connecting to the database and creating the cursor object let's see how to execute the queries. And executing the command is very easy. Call the cursor method execute() and pass the name of the sql command as a parameter in it. Save a number of commands as the sql\_comm and execute them

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

* Creating and Executing SQL Queries from Python
* You can interact with a database directly from Python using SQLite3 or PyMySQL. Here's how to create tables, insert, query, update, and delete using both connectors.
* ---
* 1. Using sqlite3 (Built-in)
* import sqlite3
* # Connect to SQLite database
* conn = sqlite3.connect('example.db')
* cursor = conn.cursor()
* # Create table
* cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER, name TEXT)")
* # Insert data
* cursor.execute("INSERT INTO users (id, name) VALUES (?, ?)", (1, 'Alice'))
* # Select data
* cursor.execute("SELECT \* FROM users")
* print(cursor.fetchall())
* # Update data
* cursor.execute("UPDATE users SET name = ? WHERE id = ?", ('Alicia', 1))
* # Delete data
* cursor.execute("DELETE FROM users WHERE id = ?", (1,))
* conn.commit()
* conn.close()
* ---
* 2. Using PyMySQL (MySQL)
* import pymysql
* # Connect to MySQL
* conn = pymysql.connect(
* host='localhost',
* user='your\_username',
* password='your\_password',
* database='your\_database'
* )
* cursor = conn.cursor()
* # Create table
* cursor.execute("CREATE TABLE IF NOT EXISTS users (id INT, name VARCHAR(50))")
* # Insert data
* cursor.execute("INSERT INTO users (id, name) VALUES (%s, %s)", (1, 'Bob'))
* # Select data
* cursor.execute("SELECT \* FROM users")
* print(cursor.fetchall())
* # Update data
* cursor.execute("UPDATE users SET name = %s WHERE id = %s", ('Bobby', 1))
* # Delete data
* cursor.execute("DELETE FROM users WHERE id = %s", (1,))
* conn.commit()
* conn.close()
* ---
* Key Notes:
* Always use parameterized queries (? or %s) to prevent SQL injection.
* Call commit() after INSERT, UPDATE, or DELETE.
* Always close() the connection to free resources.
* Let me know if you'd like to use these queries in a real project or add error handling!

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

* Here is the information about using re.search() and re.match() for pattern matching in Python's re module:
* re.search()
* This function scans through the entire string looking for the first location where the regular expression pattern produces a match.
* It returns a match object if a match is found, otherwise it returns None.
* This method is suitable for finding a pattern anywhere within the string.
* Python
* import re
* string = "The cat sat on the mat"
* pattern = r"cat"
* match = re.search(pattern, string)
* if match:
* print("Match found:", match.group())
* else:
* print("Match not found") # Output: Match found: cat
* re.match()
* This function attempts to match the regular expression pattern to the beginning of the string.
* It returns a match object if the pattern matches at the start of the string, otherwise it returns None.
* This method is suitable for validating the start of a string or checking for a specific pattern at the beginning.
* Python
* import re
* string = "The cat sat on the mat"
* pattern = r"The"
* match = re.match(pattern, string)
* if match:
* print("Match found:", match.group()) # Output: Match found: The
* else:
* print("Match not found")
* pattern = r"cat"
* match = re.match(pattern, string)
* if match:
* print("Match found:", match.group())
* else:
* print("Match not found") # Output: Match not foun

1. Difference between search and match.

* Both search() and match() come from Python’s re module and are used for pattern matching in strings — but they work differently.
* re.match()

Checks for a match only at the beginning of the string.

* Returns None if the pattern is not at the start.

import re

text = "Hello world"

result = re.match("Hello", text)

print(result) # Match object

result = re.match("world", text)

print(result) # None (not at the start)

* re.search()

Searches the entire string for a match.

Returns the first match found anywhere in the string.

result = re.search("world", text)

print(result) # Match object (found in the middle)

Match location Start of string only Anywhere in the string

Use case Precise start matching General pattern search

Returns Match object or None

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

search() checks for a match anywhere in the string.