Assignment-3

1. Printing on Screen:-

Introduction to the print() function in Python:

The print() function in Python is used to display output on the console. It is one of the most commonly used functions in Python programming.

Syntax:

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

Basic Usage

Printing a String:

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

Formatting Outputs Using f-strings and .format() in Python:

Python provides multiple ways to format strings dynamically, and two of the most commonly used methods are **f-strings** (formatted string literals) and the .format() method.

Using f-strings (Formatted String Literals):

Introduced in **Python 3.6**, f-strings allow you to embed expressions inside string literals using curly braces {}.

Basic Syntax

```
variable = "World"
print(f"Hello, {variable}!")
```

Output:

```
Hello, World!
```

Using f-strings with Variables

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Alice and I am 25 years old.
```

Performing Calculations Inside f-strings

```
a = 5
b = 10
print(f"The sum of {a} and {b} is {a + b}.")
```

Output:

```
The sum of 5 and 10 is 15.
```

2. <u>Reading Data from Keyboard:-</u>

Using the input() Function to Read User Input in Python

The input() function in Python is used to take user input from the keyboard as a string. It allows interactive programs to accept user-provided data.

Basic Syntax

```
user_input = input("Enter something: ")
print("You entered:", user_input)
```

Example Interaction:

```
Enter something: Hello, Python!
You entered: Hello, Python!
```

- The **prompt** (inside input()) is displayed to the user.
- The **user types input and presses Enter**.
- The **input is stored as a string** in user_input.

Converting User Input into Different Data Types in Python

By default, the input() function **returns user input as a string**. To work with numbers or other data types, you need to **convert** the input into the appropriate type.

1. Converting Input to Integer (int)

If the user enters a number and you need it as an integer:

```
age = int(input("Enter your age: "))
print("Next year, you will be:", age + 1)
```

Example Interaction:

```
Enter your age: 25
Next year, you will be: 26
```

2. Converting Input to Float (float)

For decimal numbers, use float():

```
height = float(input("Enter your height in meters: "))
print("Your height in centimeters:", height * 100)
```

Example Interaction:

```
Enter your height in meters: 1.75
Your height in centimeters: 175.0
```

3. Converting Input to List (list)

Splitting Input into a List

If the user enters values separated by spaces or commas:

```
numbers = input("Enter numbers separated by spaces: ").split()
print("List of numbers:", numbers)
```

Example Interaction:

```
Enter numbers separated by spaces: 10 20 30
List of numbers: ['10', '20', '30']
```

4. Converting Input to Tuple (tuple)

Convert a space-separated string into a tuple:

```
values = tuple(input("Enter values separated by spaces: ").split())
print("Tuple:", values)
```

Example Interaction:

```
Enter values separated by spaces: apple banana cherry
Tuple: ('apple', 'banana', 'cherry')
```

6. Converting Input to Dictionary (dict)

Convert key-value pairs entered as a comma-separated string:

```
data = input("Enter key-value pairs (name=John, age=25): ")
dictionary = dict(item.split('=') for item in data.split(', '))
print("Dictionary:", dictionary)
```

Example Interaction:

```
Enter key-value pairs (name=John, age=25): name=Alice, age=30
Dictionary: {'name': 'Alice', 'age': '30'}
```

3. <u>Opening and Closing Files</u>

Opening Files in Different Modes in Python

Python provides built-in functions for handling files, mainly using the open() function. The mode parameter determines how the file will be opened and how data can be read or written.

1. Basic Syntax of open()

```
file = open("example.txt", mode)
```

- "example.txt" → Name of the file.
- mode → Specifies how the file should be opened (read, write, append, etc.).

File Opening Modes in Python

Mode	Description
`'r'`	Read mode (default). Opens the file for reading; throws an error if the file does not exist.
`'w'`	Write mode. Creates a new file or overwrites an existing file.
`'a'`	Append mode. Opens the file for writing but does not overwrite existing content.
`'r+'`	Read and write mode. File must exist, otherwise an error occurs.
`'w+'`	Read and write mode. Creates a new file or overwrites existing content.

2. Opening a File in Read Mode ('r')

- Used to **read** a file.
- Throws an error if the file does not exist.

```
file = open("example.txt", "r")  # Open in read mode
content = file.read()  # Read the entire file
print(content)
file.close()  # Close the file If example.txt contains:Hello, Python!
```

Output:

```
Hello, Python!
```

3. Opening a File in Write Mode ('w')

- Creates a **new file** or **overwrites** an existing file.
- **Existing content is deleted**.

```
file = open("example.txt", "w")  # Open in write mode
file.write("Hello, Python!")  # Write data
file.close()
```

After running this code, example.txt will contain:

```
Hello, Python!
```

4. Opening a File in Append Mode ('a')

- Opens a file for **writing without overwriting existing content**.
- If the file does not exist, it is created.

```
file = open("example.txt", "a")  # Open in append mode
file.write("\nAppending new line!")  # Append new content
file.close()
```

After running this, example.txt will contain:

```
Hello, Python!
Appending new line!
```

5. Opening a File in Read & Write Mode ('r+')

- Allows **reading and writing**.
- **File must exist**, or an error occurs.

```
file = open("example.txt", "r+")  # Open for reading & writing
print("Before writing:", file.read())  # Read existing content
file.write("\nNew content added!")  # Write new data
file.close()
```

Example Output (if example.txt contains "Hello, Python!")

```
Before writing: Hello, Python!
```

6. Opening a File in Read & Write Mode ('w+')

- Allows **reading and writing**.
- **Overwrites existing content**.

```
file = open("example.txt", "w+")  # Open for read & write
file.write("Overwriting content!")  # Write new data
file.seek(0)  # Move cursor back to start
print(file.read())  # Read new content
file.close()
```

Example Output:

```
Overwriting content!
```

Using the open() Function to Create and Access Files in Python

The open() function is used to **create**, **read**, **write**, and **append** files in Python. It provides different modes to control file operations.

1. Syntax of open()

```
file = open("filename.txt", mode)
```

- "filename.txt" → Name of the file.
- mode → Specifies how the file should be opened ("r", "w", "a", etc.).

2. Creating a New File

You can create a new file using:

- 'x' mode (exclusive creation)
- 'w' mode (creates if it doesn't exist)
- 'a' mode (creates if it doesn't exist)

Using 'x' Mode (Fails if File Exists)

```
file = open("newfile.txt", "x")  # Creates a file
file.close()
print("File created successfully!")
```

Using 'w' Mode (Creates or Overwrites)

```
file = open("newfile.txt", "w")  # Creates if not exists
file.write("This is a new file.")
file.close()
```

Using 'a' Mode (Creates if Not Exists)

```
file = open("newfile.txt", "a")  # Creates if not exists
file.write("\nAppending new data.")
file.close()
```

Closing Files Using close() in Python

When working with files in Python, it is important to **close** them after reading or writing to free system resources. The close() method is used for this purpose.

How to Close a File?

Use the close() method after performing file operations.

```
file = open("example.txt", "w")  # Open file in write mode
file.write("Hello, Python!")  # Write data
file.close()
```

4. Reading and Writing Files:

Reading from a File Using read(), readline(), and readlines() in Python

Opening a File for Reading

To read from a file, we use the open() function with the "r" mode:

```
file = open("example.txt", "r")  # Open file in read mode
```

Using with open() is recommended as it automatically closes the file:

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

   # File operations go here
```

Reading the Entire File Using read()

The read() function reads the entire file as a single string.

Example:

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

   content = file.read()

   print(content)
```

Output:

```
Line 1: Hello, Python!

Line 2: Learning file handling.

Line 3: Reading data from files.
```

Reading a Single Line Using readline()

The readline() function reads only **one line** at a time.

Example:

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

    line1 = file.readline()

    print("First Line:", line1.strip())  # `.strip()` removes extra newline characters

    line2 = file.readline()

    print("Second Line:", line2.strip())
```

Output:

```
First Line: Line 1: Hello, Python!

Second Line: Line 2: Learning file handling.
```

Reading All Lines Using readlines()

The readlines() function reads **all lines** from the file and returns them as a **list**.

Example:

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

    lines = file.readlines()

    print(lines)
```

Output:

```
['Line 1: Hello, Python!\n', 'Line 2: Learning file handling.\n', 'Line 3: Reading data from files.\n']
```

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

write() function

The write() function will write the content in the file without adding any extra characters.

Example:

```
with open("example.txt", "w") as file:
    file.write("Hello, Python!\n")
    file.write("Writing to files is easy.\n")
```

Output (example.txt):

```
Hello, Python!
Writing to files is easy.
```

Writing Multiple Lines Using writelines()

The writelines() function writes a list of strings to a file.

Example:

```
lines = ["Line 1: Python is fun.\n", "Line 2: File handling is important.\n", "Line 3: Writing multiple lines.\n"]
with open("example.txt", "w") as file:
    file.writelines(lines)
```

Output (example.txt):

```
Line 1: Python is fun.
Line 2: File handling is important.
Line 3: Writing multiple lines.
```

5.Exception Handling:

Introduction to Exceptions in Python

An exception in Python is an error that occurs during program execution, disrupting the normal flow of the program. Instead of crashing the program, Python provides mechanisms to handle exceptions using try, except, and finally blocks.

Handling Exceptions Using try, except, and finally

try and except Block

The try block is used to enclose the code that may raise an exception. If an exception occurs, it is caught by the except block.

Example: Handling Division by Zero

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
    except ZeroDivisionError:
            print("Error: Division by zero is not allowed.")
```

Output:

```
Error: Division by zero is not allowed.
```

Using finally Block

The finally block contains code that **always executes**, regardless of whether an exception occurred or not.

Example: Using finally

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found.")
finally:
    print("Execution completed.")  # This will always run
```

Even if the file does not exist, the message **"Execution completed."** will be printed.

Understanding Multiple Exceptions and Custom Exceptions in Python

Handling Multiple Exceptions

Sometimes, a block of code can raise different types of exceptions. Python allows handling multiple exceptions using multiple except blocks or by grouping them in a single except block.

1. Handling Different Exceptions Separately

You can use multiple except blocks to handle different exceptions.

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2  # May raise ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input! Please enter a number.")
```

Custom Exceptions in Python

Python allows you to define custom exceptions by **creating a new exception class** that inherits from the built-in Exception class.

1. Creating and Raising a Custom Exception

You can define your own exception using a class.

```
class NegativeNumberError(Exception):
    """Custom exception for negative numbers."""
    pass
try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed!")
except NegativeNumberError as e:
    print(f"Custom Exception: {e}")
```

6. Class and Object (OOP Concepts)

Understanding Classes, Objects, Attributes, and Methods in Python

Python is an **object-oriented programming (OOP)** language, which means it revolves around creating and manipulating objects. Let's break down the core concepts:

1.Classes

A **class** is a blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class will have.

Example:

```
class Car:
    pass
```

Here, Car is a class, but it doesn’t have any properties or behaviors yet.

2.Objects

An **object** is an instance of a class. Each object has its own unique data but follows the structure defined by the class.

Example:

```
my_car = Car()  # Creating an object of the Car class

print(type(my_car))  # Output: <class '__main__.Car'>
```

Here, my_car is an **instance** of the Car class.

3.Attributes

Attributes are variables that store information about an object. They can be defined inside the class and assigned using __init__, which is a special method called the **constructor**.

Example:

```
class Car:

    def __init__(self, brand, model, year):

        self.brand = brand   # Instance attribute

        self.model = model

        self.year = year


# Creating objects with attributes

car1 = Car("Toyota", "Corolla", 2020)

car2 = Car("Honda", "Civic", 2022)


# Accessing attributes

print(car1.brand)  # Output: Toyota

print(car2.model)  # Output: Civic
```

Here, brand, model, and year are **attributes**.

4.Methods

Methods are functions inside a class that define behaviors of an object. They always take self as the first parameter, which refers to the instance of the class.

Example:

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        return f"{self.year} {self.brand} {self.model}"

# Creating an object
car1 = Car("Ford", "Mustang", 2023)
# Calling a method
print(car1.display_info())  # Output: 2023 Ford Mustang
```

Here, display_info() is a **method** that returns the car's details.

Difference between local and global variables.

Local Variables

A **local variable** is a variable declared inside a function. It is **only accessible within that function** and cannot be used outside.

Example of Local Variable

```
def my_function():
    x = 10
    print("Inside function:", x)
my_function()
print("Outside function:", x)  # This will cause an error
```

Output:

```
Inside function: 10
NameError: name 'x' is not defined
```

Since x is defined inside the function, it is **not available outside** and causes an error.

Global Variables

A **global variable** is declared outside of a function and can be accessed anywhere in the script.

Example of Global Variable

python

CopyEdit

```
y = 20

def my_function():

    print("Inside function:", y)

my_function()

print("Outside function:", y)
```

Output:

```
Inside function: 20

Outside function: 20
```

7. Inheritance:-

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

1. **Single Inheritance**

Single inheritance is simple but serves as the basis for understanding other forms of inheritance which are discussed next. It enables a child class to adopt or inherit properties and methods from one parent class only.

Example :

```
# Parent class

class Animal:

    def __init__(self, name):

        self.name = name

    def speak(self):

        return "Animal makes a sound"

# Child class inheriting from Animal

class Dog(Animal):

    def speak(self):

        return f"{self.name} barks"
```

```
# Creating an instance of Dog
dog = Dog("Buddy")
print(dog.speak())  # Output: Buddy barks
```

2. Multiple Inheritance

Multiple inheritances refer to a situation whereby one class is derived from more than one parent class. This comes in handy with the child class getting the functionality from the parent classes but also adds some complexity.

Example of Multiple Inheritance

```
# Parent class 1
class Animal:
    def __init__(self, name):
        self.name = name
# Parent class 2
class Pet:
    def __init__(self, owner):
        self.owner = owner
# Child class inheriting from both Animal and Pet
class Dog(Animal, Pet):
    def __init__(self, name, owner):
        Animal.__init__(self, name)
        Pet.__init__(self, owner)
    def details(self):
        return f"{self.name} is owned by {self.owner}"

dog = Dog("Rocky", "Alice")
print(dog.details())  # Output: Rocky is owned by Alice
```

3. **Multilevel Inheritance**

Here, a class is inherited from a class which in turn is inherited from another class. This leads to an inheritance pattern that gives the way to making new, more complicated hierarchies.

Example of Multilevel Inheritance

```
# Parent class

class Animal:

    def __init__(self, name):

        self.name = name

# Intermediate class

class Mammal(Animal):

    def feature(self):

        return f"{self.name} is a mammal"

# Child class

class Dog(Mammal):

    def speak(self):

        return f"{self.name} barks"


dog = Dog("Bruno")

print(dog.feature())  # Output: Bruno is a mammal

print(dog.speak())    # Output: Bruno barks
```

4. **Hybrid Inheritance**

Hybrid inheritance is also known as multiple type inheritance whereby an object can have two or more types of inheritance. It is used commonly in high-level designs and is more flexible than the first one.

Example:

```
# Parent class

class Animal:

    def __init__(self, name):

        self.name = name
    def speak(self):
        return "Animals make different sounds"
# Intermediate class inheriting from Animal (Multilevel Inheritance)


class Mammal(Animal):
    def __init__(self, name):
        super().__init__(name)
    def feature(self):
        return f"{self.name} is a mammal"


# Another Parent class (Multiple Inheritance)
class Pet:
    def __init__(self, owner):
        self.owner = owner
    def get_owner(self):
        return f"Owned by {self.owner}"


# Child class inheriting from Mammal and Pet (Hybrid Inheritance)
class Dog(Mammal, Pet):
    def __init__(self, name, owner):
        Mammal.__init__(self, name)
        Pet.__init__(self, owner)
    def speak(self):
        return f"{self.name} barks"


# Creating an object
dog = Dog("Buddy", "Alice")
print(dog.feature())   # Output: Buddy is a mammal
print(dog.get_owner()) # Output: Owned by Alice
print(dog.speak())     # Output: Buddy barks
```

5. **Hierarchical Inheritance**

In this concept one parent class is used to create multiple classes and all the classes have the same parent. Such inheritance is most useful when creating classes that should contain common sets of methods and properties.

Example:

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return "Animals make sounds"

# Child class 1
class Dog(Animal):
    def speak(self):
        return f"{self.name} barks"

# Child class 2
class Cat(Animal):
    def speak(self):
        return f"{self.name} meows"

# Creating objects
dog = Dog("Buddy")
cat = Cat("Kitty")

print(dog.speak())  # Output: Buddy barks
print(cat.speak())  # Output: Kitty meows
```

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

In Python, the super() function is used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

super() function in Python Example

In the given example, The **Emp** class has an **__init__** method that initializes the **id**, and **name** and **Adds** attributes. The **Freelance** class inherits from the **Emp** class and adds an additional attribute called **Emails. It calls the parent class's __init__ method super() to initialize the inherited attribute.**

Ex:

```
class Emp():

    def __init__(self, id, name, Add):

        self.id = id

        self.name = name

        self.Add = Add


# Class freelancer inherits EMP

class Freelance(Emp):

    def __init__(self, id, name, Add, Emails):

        super().__init__(id, name, Add)

        self.Emails = Emails


Emp_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")

print('The ID is:', Emp_1.id)

print('The Name is:', Emp_1.name)

print('The Address is:', Emp_1.Add)

print('The Emails is:', Emp_1.Emails)
```

Output :

```
The ID is: 103
The Name is: Suraj kr gupta
The Address is: Noida
The Emails is: KKK@gmails
```

8. Method Overloading and Overriding

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

method overloading is a feature in many programming languages that allows defining multiple methods with the same name but different parameters (either in number, type, or both). However, Python does not support method overloading in the traditional sense like Java or C++.

Example:

```
class Demo:
    def show(self, a):
        print(f"One argument: {a}")

    def show(self, a, b):  # This will override the previous method
        print(f"Two arguments: {a}, {b}")

# Usage
obj = Demo()
obj.show(10, 20)   # Works: Two arguments: 10, 20
obj.show(10)       # Error: TypeError: show() missing 1 required positional argument: 'b'
```

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

Method Overriding in Python

Method **overriding** occurs when a **child class** provides a **new implementation** of a method that is already defined in its **parent class**. This allows modifying or extending the behavior of inherited methods.

Example of Method Overriding:

```
class Parent:
    def show(self):
        print("This is the parent class method.")

class Child(Parent):
    def show(self):  # Overriding the parent class method
        print("This is the child class method.")
```

```
# Usage

obj1 = Parent()

obj1.show()  # Output: This is the parent class method.

obj2 = Child()

obj2.show()  # Output: This is the child class method.
```

9. SQLite3 and PyMySQL (Database Connectors):

Introduction to SQLite3 and PyMySQL for database connectivity.

1) SQLite3

When working with databases in Python, two popular choices for database connectivity are **SQLite3** (for lightweight, file-based databases) and **PyMySQL** (for connecting to MySQL servers). Below is a brief introduction to both.

Connecting to an SQLite Database

```
import sqlite3


# Connect to database (creates file if not exists)

conn = sqlite3.connect("mydatabase.db")

cursor = conn.cursor()


# Create table

cursor.execute('''

    CREATE TABLE IF NOT EXISTS users (

        id INTEGER PRIMARY KEY AUTOINCREMENT,

        name TEXT NOT NULL,

        email TEXT UNIQUE NOT NULL

    )

''')

# Insert data

cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ("John Doe",
"john@example.com"))
```

```
# Commit and close connection
conn.commit()
conn.close()
```

Fetching Data

```
conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
for row in rows:
    print(row)
conn.close()
```

2) PyMySQL

What is PyMySQL?

- A **Python library** for connecting to a **MySQL or MariaDB** database.
- Requires a **running MySQL server**.
- Uses standard SQL queries.

Connecting to a MySQL Database

```
import pymysql


# Establish connection
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="mydatabase"
)
cursor = conn.cursor()
```

```
# Create a table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        email VARCHAR(255) UNIQUE NOT NULL
    )
''')

# Insert data
cursor.execute("INSERT INTO users (name, email) VALUES (%s, %s)", ("Alice",
"alice@example.com"))

conn.commit()
conn.close()
```

Fetching Data

```
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="mydatabase"
)

cursor = conn.cursor()
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
for row in rows:
    print(row)

conn.close()
```

Creating and executing SQL queries from Python using these connectors.

You can execute SQL queries from Python using different database connectors. The choice depends on the database you are using. Here are some commonly used connectors:

1. SQLite (Built-in)

```
import sqlite3

conn = sqlite3.connect("database.db")

cursor = conn.cursor()


cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, email TEXT)")

cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ("John Doe", "john@example.com"))


conn.commit()

conn.close()
```

2. MySQL (mysql-connector-python or PyMySQL)

```
import mysql.connector


conn = mysql.connector.connect(host="localhost", user="root", password="password", database="testdb")

cursor = conn.cursor()


cursor.execute("CREATE TABLE IF NOT EXISTS users (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100), email VARCHAR(100))")

cursor.execute("INSERT INTO users (name, email) VALUES (%s, %s)", ("Jane Doe", "jane@example.com"))


conn.commit()

conn.close()
```

10. Search and Match Functions:

Difference between search and match.

1. re.match()

- **Matches only at the beginning of the string.**
- If the pattern is found at the start, it returns a match object; otherwise, it returns None.

Example:

```
import re

pattern = r"\d+"  # Matches one or more digits

text1 = "123abc"

text2 = "abc123"


match1 = re.match(pattern, text1)

match2 = re.match(pattern, text2)


print(match1.group() if match1 else "No match")  # Output: 123

print(match2.group() if match2 else "No match")  # Output: No match
```

2. re.search()

- **Searches for the pattern anywhere in the string.**
- Returns the first occurrence of the pattern as a match object.

Example:

```
import re

pattern = r"\d+"

text = "abc123xyz"


search = re.search(pattern, text)


print(search.group() if search else "No match")  # Output: 123
```

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

Python's re module provides two important functions for pattern matching: re.search() and re.match(). Although both are used to find patterns in a string, they behave differently.

1. re.match()

- **Matches only at the beginning of the string.**
- If the pattern is found at the start, it returns a match object; otherwise, it returns None.

Example:

```
import re

pattern = r"\d+"  # Matches one or more digits
text1 = "123abc"
text2 = "abc123"

match1 = re.match(pattern, text1)
match2 = re.match(pattern, text2)

print(match1.group() if match1 else "No match")  # Output: 123
print(match2.group() if match2 else "No match")  # Output: No match
```

2. re.search()

- **Searches for the pattern anywhere in the string.**
- Returns the first occurrence of the pattern as a match object.

Example:

```
import re

pattern = r"\d+"
text = "abc123xyz"

search = re.search(pattern, text)
print(search.group() if search else "No match")  # Output: 123
```