1.Introduction to Python:-

Python is a high-level, interpreted, general-purpose programming language. It was created by Guido van Rossum and first released in **1991**. Python emphasizes code readability and uses simple, easy-to-understand syntax. Because of its versatility and large community support, Python is widely used in web development, data analysis, artificial intelligence, machine learning, automation, and more.

Features:-

- Easy to Learn and Use
- Interpreted Language
- High-Level Language
- Object-Oriented
- Free and Open Source

History and evolution of Python:-

1 . Origin:

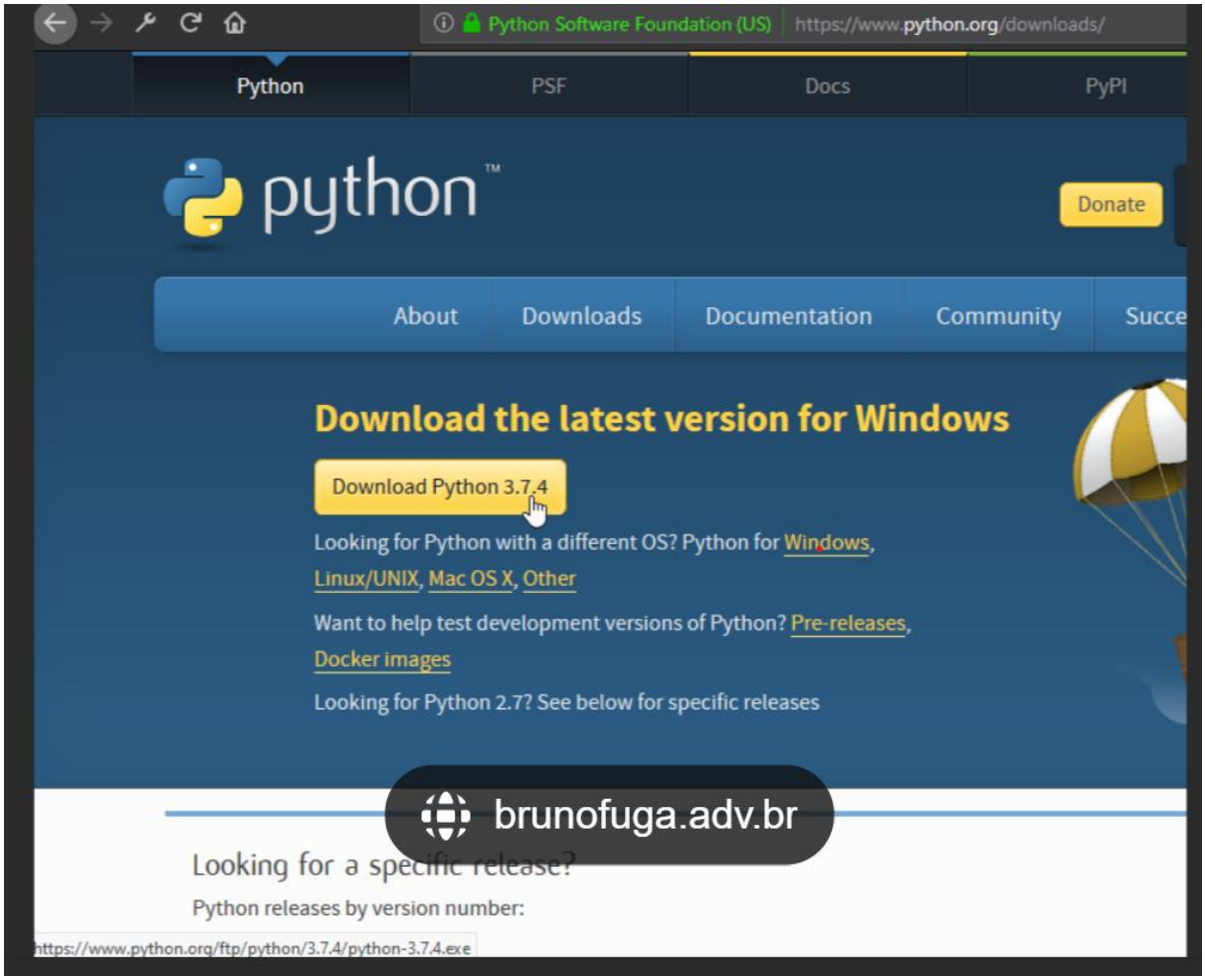
- **Creator:** Guido van Rossum
- **Developed at:** Centrum Wiskunde & Informatica (CWI), Netherlands
- **Year:** Late 1980s (Started in 1989), Released in **1991**
- **Python 1.0** released in **January 1991**
- **Python 2.0** released in **October 2000**
- **Python 3.0** released in **December 200**

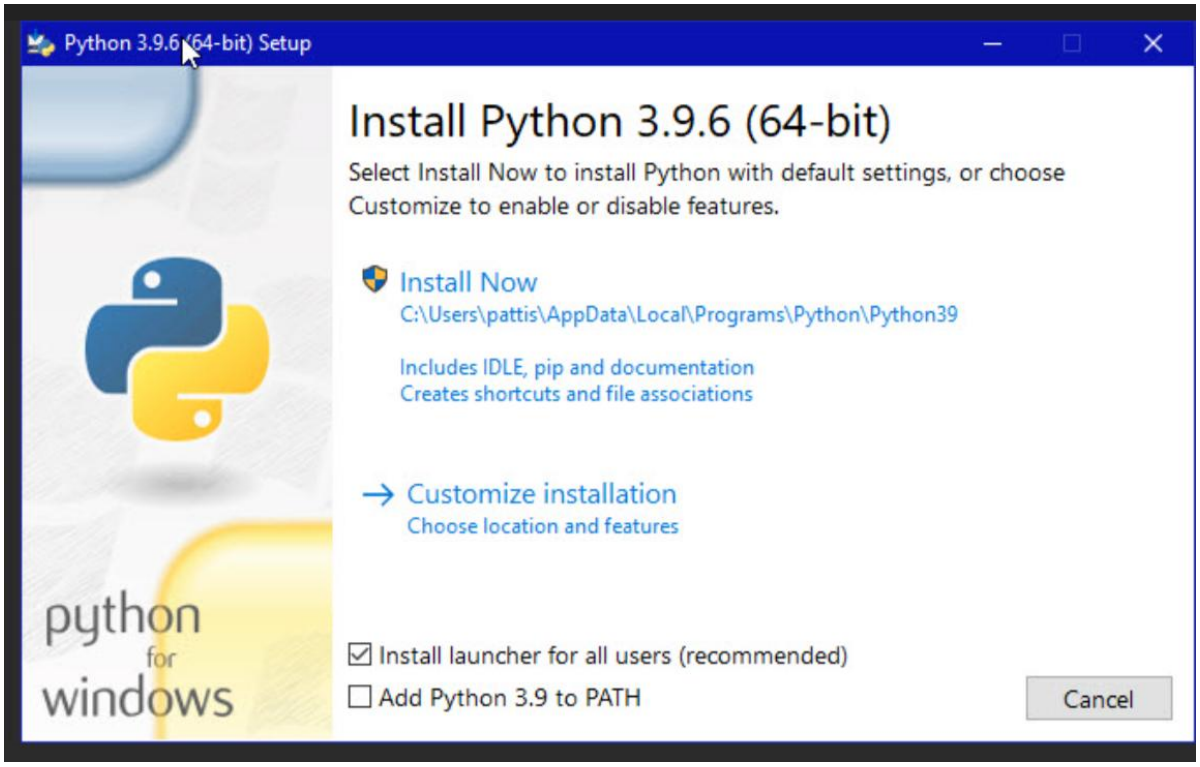
• Advantages of using Python over other programming languages

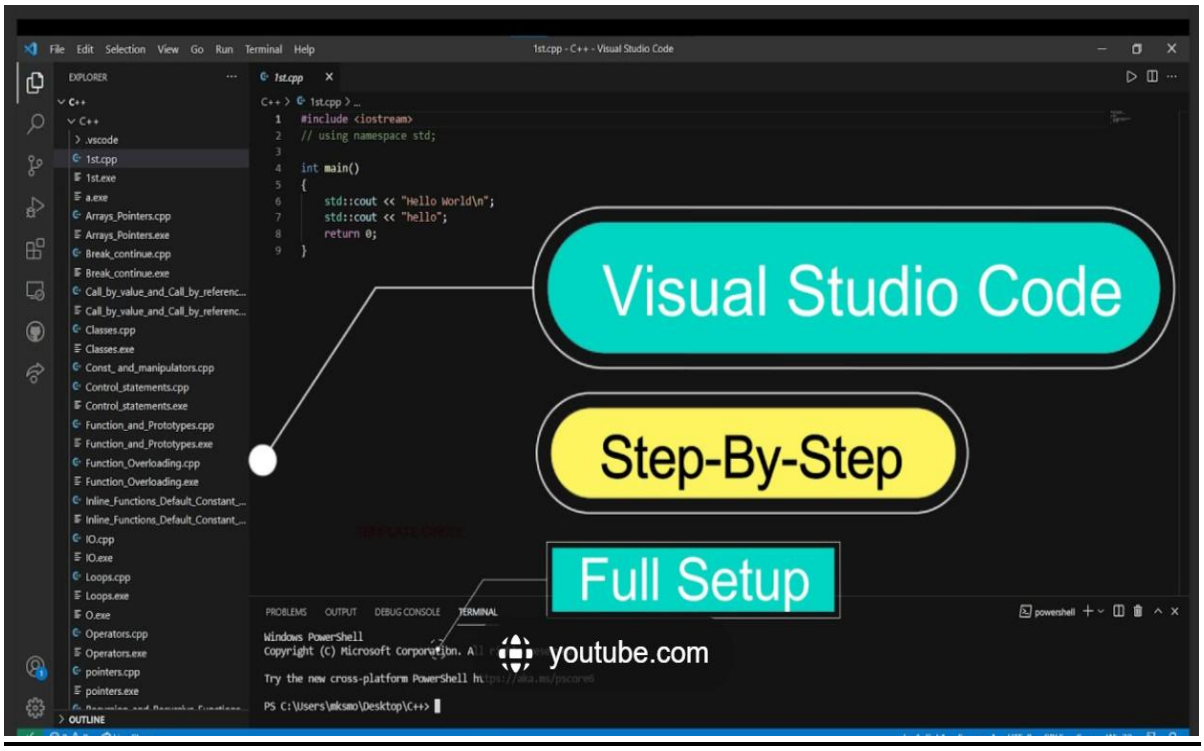
1. Easy to Read and Write
2. 2. Free and Open Source
3. Cross-Platform Language

4. 4. Huge Standard Library
5. Support for Multiple Programming Paradigms

• Installing Python and setting up the development environment







- **first Python program.**

```
s1=int(input("enter subject1 mark"))

s2=int(input("enter subject2 mark"))
```

```
s3=int(input("enter subject3 mark"))
s4=int(input("enter subject4 mark"))
total=s1+s2+s3+s4
print("total:”, total)
pr=total/4
print("PR: ,pr)
if pr>=70:
   print("dist. ")
elif pr>=60:
   print("first class")
elif pr>=50:
   print("second class")
else:
   print("pass class")
```

2. Understanding Python's PEP 8 guidelines.

PEP 8 (Python Enhancement Proposal 8) is the **style guide for Python code**. It was written to improve the **readability and consistency** of Python code across the Python community.

1. **Indentation**
2. **Line Length**
3. **Blank Lines**
4. Imports

- Making code easier to read
- Enhancing consistency
- Improving collaboration in teams

- **ındentation, comments, and naming conventions in Python**

1.Indentation in Python:

Python uses **indentation** (whitespace at the beginning of a line) to define **blocks of code**. Unlike other languages that use braces `{}` or keywords like `begin/end`, Python relies on indentation.

2. Comments in Python

Comments are used to explain code and make it more readable. Python ignores comments during execution.

Single-line Comment:

Start with #:

3. Naming Conventions in Python (PEP 8)

PEP 8 is the **style guide** for writing clean, readable Python code.

Variables and Functions:

- Use **lowercase letters** and **underscores**: user_name, calculate_total()

Constants:

- Use **all uppercase letters** with underscores: MAX_LIMIT, PI_VALUE

Classes:

Use **CamelCase (Pascal Case)**: Student Data, Bank Account

What Is Readable and Maintainable Code?

- **Readable Code**: Easy to understand by humans.
- **Maintainable Code**: Easy to fix bugs, add new features, and adapt to changes
- **EX**.:-

```
# Program to calculate the average of three numbers


def get_input():
    """Get three numbers from the user."""
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))
    c = float(input("Enter third number: "))
    return a, b, c


def calculate_average(a, b, c):
    """Return the average of three numbers."""
    return (a + b + c) / 3


def main():
    num1, num2, num3 = get_input()
    average = calculate_average(num1, num2, num3)
    print("Average:", average)
```

```
# Call main function
if __name__ == "__main__":
    main()
```

3.Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

Data Type	Description	Example
int	Whole numbers	10,-5
Float	Decimal numbers	3.14
Str	Text/characters	“hello”
list	Ordered, changeable collection	[“hello"]
tuple	Ordered, unchangeable collection	(“hii”)
dict	Key-value pairs	{4444}
set	Unordered, unique elements	{1,11,22}

- ***Python variables and memory allocation.***

A **variable** in Python is a name that refers to a value stored in memory. It acts as a reference or label to data.

------x = 10

name = "Alice"------

Python handles memory using an automatic memory management system, mainly via:

------a = 5

print(id(a))------

Python operators: arithmetic, comparison, logical, bitwise.

1. Arithmetic Operators:-

Used to perform basic mathematical operations.

Operator	Description	Example	Result
`+`	Addition	`10 + 5`	`15`
`-`	Subtraction	`10 - 5`	`5`
`*`	Multiplication	`10 * 5`	`50`
`/`	Division	`10 / 5`	`2.0`
`//`	Floor Division	`10 // 3`	`3`
`%`	Modulus	`10 % 3`	`1`
`**`	Exponentiation	`2 ** 3`	`8`

2. Comparison Operators

Used to compare values. Returns True or False.

Operator	Description	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`3 < 5`	`True`
`>=`	Greater or equal	`5 >= 5`	`True`
`<=`	Less or equal	`3 <= 5`	`True`

3. Logical Operators

Used to combine conditional statements

Operator	Description	Example	Result
`and`	True if both are True	`5 > 3 and 4 > 2`	`True`
`or`	True if at least one is True	`5 > 3 or 2 > 4`	`True`
`not`	Inverts the result	`not(5 > 3)`	`False`

4. Bitwise Operators

Used to perform bit-level operations

Operator	Description	Example	Binary Operation	Result
`&`	Bitwise AND	`5 & 3`	`0101 & 0011` → `0001`	`1`
`	`	Bitwise OR	`5	3`	`7`
`^`	Bitwise XOR	`5 ^ 3`	`0101 ^ 0011` → `0110`	`6`
`~`	Bitwise NOT	`~5`	`~0101` → `-0110`	`-6`
`<<`	Left Shift	`5 << 1`	`0101 << 1` → `1010`	`10`
`>>`	Right Shift	`5 >> 1`	`0101 >> 1` → `0010`	`2`

4. Conditional Statements:-

Conditional statements in Python are used to perform different actions based on different conditions. The primary keywords are.

1. if Statement

Used to **check a condition**. If the condition is True, the block runs.

```
x = 10
if x > 5:
    print("x is greater than 5")----------
```

2. if...else Statement

Adds an **alternative block** that runs if the condition is False

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is 5 or less")----------
```

3. if...elif...else Statement

Checks **multiple conditions** in order

```
x = 10

if x < 0:
    print ("Negative number")
elif x == 0:
    print("Zero")
```

```
else:
    print("Positive number”) ----------
```

Nested if-else conditions.

A **nested if-else** means using one if or else statement **inside another**.

It allows you to check **multiple levels of conditions**

```
num = 5
if num >= 0:
    if num == 0:
        print ("Number is zero")
    else:
        print("Number is positive")
else:
    print ("Number is negative") ----------
```

5. Looping (For, While)

1.introduction **to for and while loops.**

Loops are used to execute a block of code **repeatedly** as long as a condition is true or until a sequence is exhausted.

Ex:-

```
for i in range(1, 6):
    print(i)
```

2. while Loop:-

Repeats as long as a condition is True.

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

- **How loops work in Python.**

Loops in Python are used to **repeat a block of code** multiple times, either a specific number of times or until a certain condition is met.

1. for loop

Used when you want to **iterate over a sequence** (like a list, string, tuple, or range).

```
for i in range(5):
    print(i)
```

2.while loop

Used when you want to **repeat code as long as a condition is True**.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Using loops with collections (lists, tuples, etc.).

Lists:-

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

tuples:-

```
colors = ("red", "green", "blue")
for color in colors:
    print(color)
```

6. Generators and Iterators

Understanding how generators work in Python:-

A generator is a special type of iterator in Python that allows you to iterate through a sequence of values lazily (on demand) using the yield keyword, rather than computing all values at once and storing them in memory.

1. **Using a Function with yield**

```
def my_generator():
    yield 1
    yield 2
    yield 3
```

2. **Generator Expression (Like List Comprehension)**

```
gen = (x * x for x in range(3))
print(next(gen))  # Output: 0
print(next(gen))  # Output: 1
print(next(gen))  # Output: 4
```

Difference between yield and return.

Feature	`yield`	`return`
Used In	Generator functions	Normal functions
Purpose	To produce a value and pause function execution	To exit the function and send back a value
Returns	A generator object (an iterator)	A single value
Execution Flow	Pauses the function and can resume later	Terminates the function completely
Can Be Used Multiple Times	Yes, can `yield` multiple values in a loop	Only one `return` is executed per call
Memory Usage	Low (values are generated lazily)	Higher (all values usually computed at once)
Example Use Case	Streaming data, infinite sequences, large datasets	Simple computations, returning a final result

Understanding iterators and creating custom iterators.

An iterator is an object in Python that allows traversing through all the elements of a collection, one at a time, using the built-in next() function.

An **iterator** must implement two methods:

- __iter__() → returns the iterator object itself
- __next__() → returns the next item (raises StopIteration when done)

```
nums = [10, 20, 30]

itr = iter(nums)  # Get an iterator from list


print(next(itr))  # 10

print(next(itr))  # 20
```

```
print(next(itr))  # 30
# print(next(itr))  # Raises StopIteration
```

7. Functions and Methods

• Defining and calling functions in Python.

Functions in Python are blocks of reusable code that perform a specific task. You define a function using the def keyword and call it by its name followed by parentheses ().

1. **Defining a Function:-**

```
def greet(name):
    print("Hello,", name)
```

2. **Calling a Function:-**

```
greet("Alice")
# Output: Hello, Alice
```

🔹 Summary Table

Component	Example
Define function	`def greet(name):`
Call function	`greet("Alice")`
Return value	`return a + b`
Default parameter	`def greet(name="User"):`
No parameter	`def say_hello():`

Function arguments (positional, keyword, default).

In Python, there are three main types of function arguments:

1. Positional Arguments

These are the most common — values are **assigned based on their position** in the function call.

Example:

```
def student_info(name, age):
    print("Name:", name)
    print("Age:", age)

student_info("Alice", 20)
```

2. Keyword Arguments

You can pass arguments **by name**, so the order doesn't matter.

```
student_info(age=20, name="Alice")
```

3. Default Arguments

You can provide a **default value** for one or more parameters. If not passed during the function call, the default is used.

```
def greet(name="Guest"):
    print("Hello,", name)

greet("Dharm")  # Hello, Dharm
greet()         # Hello, Guest
```

Scope of variables in Python:-

The scope of a variable refers to the part of the program where the variable is accessible.

Python has four types of variable scopes, following the LEGB Rule:

1. Local Scope

A variable declared inside a function is local and can only be used within that function.

Example:

```
def greet():
    name = "Alice"  # local variable
    print("Hello", name)

greet()
# print(name)  # ❌ Error: name is not defined
```

3. Global Scope

A variable defined **outside any function** is global and **can be accessed inside functions** using the global keyword if modification is needed.

Example:

```
x = 10  # global variable

def show():
    print("Global x:", x)

show()
```

Built-in methods for strings, lists, etc:-

Python provides many built-in methods that help manipulate strings, lists, and other data types easily and efficiently.

1. **String Methods:-**

```
s = " Hello World "
```

Method	Description	Example
`s.lower()`	Converts to lowercase	`'hello world'`
`s.upper()`	Converts to uppercase	`'HELLO WORLD'`
`s.strip()`	Removes leading/trailing spaces	`'Hello World'`
`s.replace("World", "Python")`	Replaces text	`' Hello Python '`
`s.split()`	Splits string into list by spaces	`['Hello', 'World']`
`s.find("World")`	Finds index of substring	`7`
`s.startswith("He")`	Checks start of string	`False` (space at start)
`s.endswith("d")`	Checks end of string	`False`
`len(s)`	Length of string (including spaces)	`13`

2. **List Methods:-**

```
fruits = ["apple", "banana", "cherry"]
```

Method	Description	Example
`fruits.append("mango")`	Adds an item to end	`["apple", "banana", "cherry", "mango"]`
`fruits.insert(1, "kiwi")`	Inserts at index	`["apple", "kiwi", "banana", "cherry"]`
`fruits.remove("banana")`	Removes a specific item	`["apple", "cherry"]`
`fruits.pop()`	Removes and returns last item	`'cherry'`
`fruits.sort()`	Sorts list alphabetically	`['apple', 'banana', 'cherry']`
`fruits.reverse()`	Reverses the list	`['cherry', 'banana', 'apple']`
`fruits.count("apple")`	Counts occurrences	`1`
`fruits.index("banana")`	Returns index of item	`1`
`len(fruits)`	Number of items	`3`

3. **Common Built-in Functions:-**

Function	Description	Example
`len()`	Returns length	`len("hello") → 5`
`type()`	Returns type of object	`type(5) → <class 'int'>`
`max()`	Maximum value	`max([1, 2, 3]) → 3`
`min()`	Minimum value	`min("abc") → 'a'`
`sum()`	Sum of items (numeric list)	`sum([1, 2, 3]) → 6`
`sorted()`	Returns a sorted list	`sorted([3, 1, 2]) → [1,2,3]`

8. Control Statements (Break, Continue, Pass)

- Understanding the role of break, continue, and pass in Python loops.

These three **control flow statements** are used to **manage loop execution** — when to skip, stop, or do nothing.

1. break

Exits the loop immediately, even if the condition is still true.

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

2. continue

Skips the current iteration and moves to the next one.

Example:

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

9. String Manipulation

- Understanding how to access and manipulate strings.

1. Accessing Characters in a String

Python uses **indexing** (starting from 0) to access characters.

```
text = "Python"

print(text[0])   # P
print(text[3])   # h
print(text[-1])  # n (last character)
```

2. Slicing Strings

You can extract parts of a string using the slice syntax: string[start:end]

python

CopyEdit

```
text = "Python"

print(text[0:2])  # Py (from index 0 to 1)
print(text[:4])   # Pyth
print(text[2:])   # thon
print(text[-3:])  # hon
```

String Concatenation and Repetition

```
a = "Hello"
b = "World"

print(a + " " + b)     # Hello World
print(a * 3)           # HelloHelloHello
```

- ## Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

Strings in Python are powerful and support many **basic operations** such as **concatenation**, **repetition**, and **built-in methods** for transformation and querying

1. Concatenation

Joining two or more strings using the + operator.

```
a = "Hello"
b = "World"

result = a + " " + b
print(result)  # Output: Hello World
```

2. Repetition

Repeat a string multiple times using the * operator.

```
word = "Hi"
```

print(word * 3) # Output: HiHiHi

3. String Methods

Let’s assume:

text = "Python Programming"

Method	Description	Example Output
`text.upper()`	Converts to **uppercase**	`"PYTHON PROGRAMMING"`
`text.lower()`	Converts to **lowercase**	`"python programming"`
`text.title()`	Converts to **title case**	`"Python Programming"`
`text.capitalize()`	Capitalizes **first letter only**	`"Python programming"`
`text.strip()`	Removes leading/trailing spaces	`"Python Programming"`
`text.replace("Python", "Java")`	Replaces text	`"Java Programming"`
`text.find("gram")`	Finds substring index	`10`
`text.count("m")`	Counts occurrence of char/word	`2`

4. **Length of a String**

text = "hello"

print(len(text)) # Output: 5

5. **Using in and not in**

print("Py" in "Python")   # True

print("Java" not in "Python") # True

String slicing.

String slicing is used to extract a **substring** from a string using the

syntax:

string[start:stop:step]

1. **Basic Slicing**
   text = "Python"

   print(text[0:2]) # 'Py' → characters at index 0 and 1
   print(text[2:5]) # 'tho'

2. **Omitting Indices**
   text = "Programming"

   print(text[:4]) # 'Prog' → from beginning to index 3
   print(text[5:]) # 'amming' → from index 5 to end
   print(text[:]) # 'Programming' → full string

3. Negative Indexing

You can use negative numbers to count from the **end** of the string.

text = "Python"

print(text[-1]) # 'n' → last character

print(text[-3:-1]) # 'ho' → from 3rd last to 2nd last

print(text[:-2]) # 'Pyth'

4. Using Step in Slicing

The step defines the gap between characters.

```
text = "abcdefg"

print(text[::2])   # 'aceg' → every second character
print(text[1::2])  # 'bdf'  → from index 1, every second char
```

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators):-

How functional programming works in Python.

Functional programming is a **programming paradigm** that treats **computation as the evaluation of mathematical functions** and avoids changing state and mutable data.

- Key Concepts of Functional Programming in Python:

5. Map, Filter, Reduce

- These functions allow you to apply functions across sequences in a functional way.

map(): Apply a function to all items in an iterable.

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, nums))
print(squared)  # [1, 4, 9, 16]
```

filter(): Filter items using a function that returns True or False.

```
even = list(filter(lambda x: x % 2 == 0, nums))
print(even)  # [2, 4]
```

reduce(): Apply a function cumulatively to the items in a sequence

```
from functools import reduce
product = reduce(lambda x, y: x * y, nums)
print(product)  # 24
```

Advantages of Functional Programming

- Easier to test and debug (due to pure functions).
- Better modularity and reusability.
- Avoids side effects and mutable data.

- **Using map(), reduce(), and filter() functions for processing data.**

map(): Apply a function to all items in an iterable.

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, nums))
print(squared)  # [1, 4, 9, 16]
```

filter(): Filter items using a function that returns True or False.

```
even = list(filter(lambda x: x % 2 == 0, nums))
print(even)  # [2, 4]
```

reduce(): Apply a function cumulatively to the items in a sequence

```
from functools import reduce
product = reduce(lambda x, y: x * y, nums)
print(product)  # 24
```

Advantages of Functional Programming

- Easier to test and debug (due to pure functions).
- Better modularity and reusability.
- Avoids side effects and mutable data.

Introduction to closures and decorators.

What is a Closure?

A closure is a function object that remembers values in enclosing scopes, even if the outer function has finished executing.

Requirements for a Closure:

1. There must be a **nested function**.
2. The nested function must **refer to a value from the enclosing function**.

3. The enclosing function must **return the nested function**.

```
def outer(msg):
    def inner():
        print(f"Message: {msg}")
    return inner

my_func = outer("Hello!")
my_func()  # Output: Message: Hello!
```