**Python – Full Stack Assignment**

**Module 6 – Python Fundamentals:-**

**Introduction to Python**

* **Introduction to Python and its Features (simple, high-level, interpreted language).**

Python is a high-level, general-purpose, interpreted programming language known for its simple and readable syntax, making it beginner-friendly and widely used across various applications. Key features include:

* **Readability:**

Python's syntax is designed to be clear and concise, with keywords resembling English and using indentation to define code blocks.

* **Dynamic Typing:**

Variables do not need to be explicitly declared with a data type, making the development process faster.

* **Interpreted Language:**

Python code is executed directly by the interpreter, eliminating the need for compilation.

* **Object-Oriented:**

Python supports object-oriented programming paradigms, allowing for modular and reusable code.

* **Large Standard Library:**

Python comes with a vast collection of built-in modules that provide functionality for various tasks, including data manipulation, web development, and scientific computing.

* **History and evolution of Python.**

Python was created by Guido van Rossum in the late 1980s as a hobby project inspired by the ABC programming language. It was first publicly released in 1991. Over the years, Python has undergone several revisions and improvements, expanding its capabilities and features while maintaining its core design principles. The Python Software Foundation (PSF) now oversees the development and maintenance of the language.

* **Advantages of using Python over other programming languages.**
* **Ease of Learning:**

Python's simple syntax and clear structure make it relatively easy to learn for beginners, even those with no prior programming experience.

* **Versatility:**

Python can be used for a wide range of applications, including web development, data analysis, machine learning, scientific computing, automation, and more.

* **Large Community and Support:**

Python has a large and active community of developers and users, providing ample support and resources for learning and troubleshooting.

* **Cross-Platform Compatibility:**

Python code can run on different operating systems like Windows, macOS, and Linux without significant modifications.

* **Productivity:**

Python's focus on readability and ease of use allows developers to write code quickly and efficiently.

.

* **Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).**

There are several ways to install Python and set up a development environment. Some popular options include:

* **Downloading Python Directly:**

Users can download the latest version of Python from the official website and install it on their system.

* **Anaconda Distribution:**

Anaconda is a popular distribution of Python that includes a package manager (conda), various scientific computing libraries (like NumPy and Pandas), and other useful tools.

* **Integrated Development Environments (IDEs):**

IDEs like PyCharm and Visual Studio Code provide a comprehensive environment for Python development, offering features such as code editing, debugging, version control, and more.

* **Writing and executing your first Python program.**

print("Hello, World!").

**Programming Style**

* **Understanding Python’s PEP 8 guidelines.**

PEP 8 is Python's official style guide, providing guidelines for writing readable, consistent, and maintainable code. It covers various aspects like indentation, line length, naming conventions, and whitespace usage. Adhering to PEP 8 improves code clarity and makes it easier for others (and yourself) to understand and collaborate on.

* **Indentation, comments, and naming conventions in Python.**

Python relies on indentation to define code blocks, using four spaces per level as recommended by PEP 8. Comments are added using the # symbol for single-line or triple quotes (''' or """) for multi-line explanations. Naming conventions suggest lowercase\_with underscores for variables, functions, and modules, UpperCamelCase for classes, and ALL\_CAPS for constants, enhancing code readability and maintainability..

* **Writing readable and maintainable code.**

Python relies on indentation to define code blocks, using four spaces per level as recommended by PEP 8. Comments are added using the # symbol for single-line or triple quotes (''' or """) for multi-line explanations. Naming conventions suggest lowercase\_with underscores for variables, functions, and modules, UpperCamelCase for classes, and ALL\_CAPS for constants, enhancing code readability and maintainability.

**Core Python Concepts:**

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

### **Integer (**int**) :**Whole numbers (positive or negative), no decimal point

### Float (float): Numbers with decimal points.

### String (str) : Sequence of characters, enclosed in quotes.

### List (list) : Ordered, **mutable** (changeable) collection of items. Allows duplicates.

### Tuple (tuple) : Ordered, **immutable** collection of items. Allows duplicates.

### Dictionary (dict) : Unordered, **key-value** pairs. Keys must be unique and immutable.

### Set (set) : Unordered collection of **unique** items (no duplicates). Useful for removing duplicates.

* **Python variables and memory allocation.**

#### **1. What is a Variable?**

* A **variable** in Python is a name that refers to a memory location used to store data.
* Variables are created by **assigning** a value to a name.

#### **2. How Python Handles Variables**

* **Dynamic Typing:** In Python, you don’t need to declare the type of a variable; the interpreter automatically determines the type based on the assigned value.
  + Example: x = 5 assigns an integer to x, and x = "hello" changes x to a string.

#### **3. Memory Allocation in Python**

* Python uses **automatic memory management** via **reference counting** and **garbage collection.**
  + **Reference Counting:** Each object in memory has a reference count. When a variable is assigned an object, the reference count increases.
  + **Garbage Collection:** Python automatically frees memory by deleting objects that are no longer in use (i.e., when their reference count drops to zero).

#### **4. Immutable vs Mutable Types**

* **Immutable types** (like integers, floats, strings, and tuples) cannot be changed after they are created. When you modify their value, a **new object** is created in memory.

#### **5. Memory Efficiency and Variables**

* Python tries to optimize memory usage by reusing immutable objects for small numbers or strings.
  + Example: Integers in the range -5 to 256 are often **cached** to improve efficiency.

#### **6. Variable Scope**

* **Local variables** are only accessible within the function or block where they are defined.
* **Global variables** are accessible from anywhere in the program after they are declared.

#### **7. Del and Memory Management**

* You can explicitly delete a variable using the del statement, which removes the variable reference, and the memory is freed if there are no other references to the object.
* **Python operators: arithmetic, comparison, logical, bitwise**

**Python Arithmetic Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | Adds a and b |
| - | Subtraction | a - b | Subtracts b from a |
| \* | Multiplication | a \* b | Multiplies a and b |
| / | Division | a / b | Divides a by b |
| // | Floor Division (round down) | a // b | Quotient of a divided by b, rounded down |
| % | Modulo (Remainder) | a % b | Remainder of a divided by b |
| \*\* | Exponentiation (Power) | a \*\* b | a raised to the power of b |

**Python Comparison Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | True if a equals b, otherwise False |
| != | Not equal to | a != b | True if a is not equal to b, otherwise False |
| > | Greater than | a > b | True if a is greater than b, otherwise False |
| < | Less than | a < b | True if a is less than b, otherwise False |
| >= | Greater than or equal to | a >= b | True if a is greater than or equal to b, otherwise False |
| <= | Less than or equal to | a <= b | True if a is less than or equal to b, otherwise False |

**Python Logical Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| and | Returns True if both operands are True | a and b | True if both a and b are True, otherwise False |
| or | Returns True if at least one operand is True | a or b | True if either a or b is True, otherwise False |
| not | Reverses the Boolean value | not a | True if a is False, and False if a is True |

**Python Bitwise Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| & | Bitwise AND | a & b | Performs bitwise AND on a and b |
| ` | ` | Bitwise OR | `a |
| ^ | Bitwise XOR (exclusive OR) | a ^ b | Performs bitwise XOR on a and b |
| ~ | Bitwise NOT | ~a | Inverts all bits of a |
| << | Bitwise left shift | a << 1 | Shifts bits of a left by 1 |
| >> | Bitwise right shift | a >> 1 | Shifts bits of a right by 1 |

**Conditional Statements**

* **Introduction to conditional statements: if, else, elif.**

**1. if Statement**

The if statement is used to check if a condition is True. If it is, the block of code under the if statement is executed.

**2. else Statement**

The else statement follows an if statement. If the condition in the if statement is False, the block of code under else is executed.

**3. elif (Else If) Statement**

The elif statement is used to check multiple conditions. If the if condition is False, Python checks the conditions in the elif statements. If one of the elif conditions is True, that block of code is executed.

* **Combining if else and elif**

You can combine multiple conditions to check complex logic using if, elif, and else in one structure.

* **Nested if-else conditions.**

**Nested if-else** refers to placing an if or else statement inside another if or else statement. This allows for more complex decision-making processes, where multiple conditions must be evaluated in a hierarchical way.

**Looping (For, While)**

* **Introduction to for and while loops.**

| **Aspect** | **for Loop** | **while Loop** |
| --- | --- | --- |
| **Usage** | Iterates over a sequence (list, tuple, range, etc.) | Repeats as long as the given condition is True |
| **Condition** | Loops through each item in a sequence | Loops until a condition becomes False |
| **Example** | for x in range(5): | while x < 5: |
| **Best for** | Known number of iterations or when iterating through items | Unknown number of iterations, or based on dynamic conditions |

* **How loops work in Python.**

In Python, loops allow you to automate repetitive tasks and execute a block of code multiple times. The loop executes the code in its block until a specific condition is met. Python provides two main types of loops: the **for loop** and the **while loop**.

Let’s break down how each type of loop works and how they control the flow of execution.

### **The for loop**

The for loop is typically used when you know how many times you want the loop to run. It iterates over a sequence (such as a list, string, or range of numbers) and executes the code inside the loop for each item in the sequence.

#### How it works:

1. The loop takes the sequence and extracts the first item.
2. Executes the code block inside the loop using that item.
3. Moves to the next item in the sequence.
4. Continues this process until all items have been iterated through.

#### Example 1: for loop with a list

fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:

print(fruit)

**Execution Flow**:

1. **fruit = 'apple'**: The loop runs the print(fruit) statement, printing "apple".
2. **fruit = 'banana'**: The loop runs again, printing "banana".
3. **fruit = 'cherry'**: The loop runs one last time, printing "cherry".
4. The loop ends once all items in the list are processed.

#### Example 2: for loop with a range()

The range() function generates a sequence of numbers.

for i in range(3):

print(i)

**Execution Flow**:

1. **i = 0**: The loop prints 0.
2. **i = 1**: The loop prints 1.
3. **i = 2**: The loop prints 2.
4. The loop stops because the range has no more numbers.

### **The While loop**

The while loop runs as long as the specified condition evaluates to True. It continuously checks the condition before each iteration. If the condition becomes False, the loop stops executing.

#### How it works:

1. The condition is evaluated.
2. If the condition is True, the loop executes the code inside the block.
3. After each execution, the condition is checked again.
4. If the condition is still True, the loop continues. If it’s False, the loop ends.

#### Example 1: Basic while loop

count = 0

while count < 3:

print(count)

count += 1

**Execution Flow**:

1. **count = 0**: The condition count < 3 is True, so the loop runs and prints 0.
2. **count = 1**: The loop runs again and prints 1.
3. **count = 2**: The loop runs and prints 2.
4. **count = 3**: The condition count < 3 is now False, so the loop stops.

#### Example 2: Infinite while loop (with break)

If you set the condition in a while loop to always be True, the loop will run indefinitely unless a stopping condition is introduced (like break).

count = 0

while True:

print(count)

count += 1

if count >= 5:

break # Stops the loop when count reaches 5

**Execution Flow**:

1. **count = 0 to 4**: The loop runs and prints count values.
2. When count reaches 5, the if count >= 5: condition becomes True, and the break statement is executed, stopping the loop.

### **3. Loop Control Statements**

Python also provides several control statements that can be used inside loops to change the flow of execution:

* **break**: Exits the loop immediately, even if the condition is still True.
* **continue**: Skips the current iteration and proceeds to the next iteration of the loop.
* **else**: Executes a block of code after the loop finishes (only if the loop was not exited by break).

#### Example of break and continue:

for i in range(5):

if i == 2:

continue # Skips printing 2

if i == 4:

break # Stops the loop when i equals 4

print(i)

**Output:**

0

1

3

**Explanation**:

* When i is 2, the continue statement skips the current iteration and moves to i = 3.
* When i reaches 4, the break statement exits the loop.
* **Using loops with collections (lists, tuples, etc.).**

In Python, collections such as **lists**, **tuples**, **sets**, and **dictionaries** are commonly used to store multiple items. You can use loops to iterate over these collections and perform operations on the elements within them.

### **1. Looping through Lists**

A **list** is an ordered collection that allows duplicate elements and can store different data types.

#### Example 1: Using for loop with a list

fruits = ['apple', 'banana', 'cherry', 'date']

for fruit in fruits:

print(fruit)

**Output:**

apple

banana

cherry

date

### **2. Looping through Tuples**

A **tuple** is an ordered, immutable collection of elements. You can loop through tuples the same way as lists.

#### Example: Using for loop with a tuple

coordinates = (10, 20, 30)

for coordinate in coordinates:

print(coordinate)

**Output:**

10

20

30

### **3. Looping through Sets**

A **set** is an unordered collection of unique elements. Sets do not allow duplicates.

#### Example: Using for loop with a set

numbers = {1, 2, 3, 4, 5}

for number in numbers:

print(number)

**Output:**

1

2

3

4

5

### **4. Looping through Dictionaries**

A **dictionary** is an unordered collection of key-value pairs. You can loop through dictionaries in various ways:

* Iterate through **keys**
* Iterate through **values**
* Iterate through **key-value pairs**

#### Example 1: Looping through dictionary keys

person = {'name': 'Alice', 'age': 25, 'city': 'New York'}

for key in person:

print(key)

**Output:**

name

age

city

### **5. Nested Loops with Collections**

You can use **nested loops** to iterate over nested collections, such as a list of lists or a list of dictionaries.

#### Example: Nested loops with a list of lists

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:

for item in row:

print(item, end=" ")

print() # Moves to the next line after each row

**Output:**

1 2 3

4 5 6

7 8 9

**Explanation**: The outer loop iterates through each row (which is a list), and the inner loop iterates through each element in the row.

### **6. Using** enumerate() **to Access Index and Value**

The enumerate() function allows you to loop through a collection and get both the index and the value at the same time.

#### Example: Using enumerate() with a list

fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):

print(f"Index {index}: {fruit}")

**Output:**

Index 0: apple

Index 1: banana

Index 2: cherry

**Explanation**: The enumerate() function returns both the index and the value of each item, making it easier to work with both.

**Generators and Iterators**

* **Understanding how generators work in Python.**

A generator is a special type of function that returns an iterator. Instead of using the return statement like regular functions, **generators use yield** to return values one at a time. When the function is called, it doesn't execute immediately; instead, it returns a **generator object** that can be iterated over.

* **yield**: The key feature of a generator. It pauses the function’s execution and returns the value to the caller. The state of the function is saved, so when the generator function is called again, it continues from where it left off.
* **Difference between yield and return.**

| **Feature** | **return** | **yield** |
| --- | --- | --- |
| **Function Type** | Regular function | Generator function |
| **Execution Flow** | Terminates function and returns value | Pauses function, returns value, and resumes next time |
| **State Preservation** | Does not preserve function's state | Preserves function's state between calls |
| **Memory Usage** | Memory used for a single value at a time | Memory efficient as it yields one value at a time |
| **Typical Use** | Returning a final result from a function | Returning a sequence of values one at a time from a generator function |
| **Function End** | Ends execution after return | Function continues after yield (state saved) |

* **Understanding iterators and creating custom iterators.**

An **iterator** is an object that keeps track of the state of the iteration, remembering which element comes next in the sequence. In other words, it’s an object that can be looped over using a for loop, and it keeps returning values until there are no more items left.

#### Characteristics of an Iterator:

1. **Has an \_\_iter\_\_() method**: It is required for making an object iterable (iterable objects are containers, such as lists, tuples, or dictionaries).
2. **Has a \_\_next\_\_() method**: This retrieves the next element in the sequence. When there are no more items, it raises a StopIteration exception.

### **reating Custom Iterators**

You can create your own custom iterator by defining a class that implements both \_\_iter\_\_() and \_\_next\_\_() methods.

Here’s an example of how to create a custom iterator:

#### **Example 1: Simple Custom Iterator**

class CountdownIterator:

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

self.start = start

self.current = start

# The \_\_iter\_\_ method returns the iterator object itself

def \_\_iter\_\_(self):

return self

# The \_\_next\_\_ method returns the next value

def \_\_next\_\_(self):

if self.current > 0:

self.current -= 1

return self.current

else:

raise StopIteration # Stops the iteration when the countdown is over

# Create an instance of the iterator

countdown = CountdownIterator(5)

# Use the iterator with a for loop

for number in countdown:

print(number)

**Functions and Methods**

* **Defining and calling functions in Python.**

**1. Defining a Function**

A function is defined using the def keyword, followed by the function name, parentheses () (which can include parameters), and a colon :. The function body is indented and contains the code that will be executed when the function is called.

Sytex:

def function\_name(parameters):

# Function body

# Do something

return value

**Calling Functions**

A function is called by writing its name followed by parentheses. If the function requires parameters, the arguments are passed inside the parentheses.

Syntax:

Func\_name(parameters)

* **Function arguments (positional, keyword, default).**

| **Argument Type** | **Description** | **Example** |
| --- | --- | --- |
| **Positional Arguments** | Values are assigned based on their position in the function call. | greet("Alice", 30) |
| **Keyword Arguments** | Values are assigned based on parameter names in the function call. | greet(age=30, name="Alice") |
| **Default Arguments** | Parameters that have a default value if no argument is passed. | greet(name="Guest", age=25) |
| **Variable Length \*args** | Accepts a variable number of positional arguments as a tuple. | add\_numbers(1, 2, 3) |
| **Variable Length \*\*kwargs** | Accepts a variable number of keyword arguments as a dictionary. | display\_info(name="Alice", age=30) |

* **Scope of variables in Python.**

**Local Scope (L)**: Variables defined within a function or block.

**Enclosing Scope (E)**: Variables defined in enclosing (outer) functions.

**Global Scope (G)**: Variables defined at the top level of the program.

**Built-in Scope (B)**: Predefined variables and functions in Python (like print(), len(), etc.).

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

**1. String Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| lower() | Converts string to lowercase | 'HELLO'.lower() → 'hello' |
| upper() | Converts string to uppercase | 'hello'.upper() → 'HELLO' |
| strip() | Removes whitespace from both ends | ' hello '.strip() → 'hello' |
| replace(a, b) | Replaces substring a with b | 'hello'.replace('l', 'x') → 'hexxo' |
| split() | Splits string into list | 'a,b,c'.split(',') → ['a', 'b', 'c'] |
| join() | Joins elements of list into a string | '-'.join(['a', 'b']) → 'a-b' |
| find() | Finds index of first occurrence | 'hello'.find('e') → 1 |
| startswith() | Checks if string starts with specified prefix | 'hello'.startswith('he') → True |
| endswith() | Checks if string ends with specified suffix | 'hello'.endswith('lo') → True |
| isdigit() | Checks if all characters are digits | '123'.isdigit() → True |

**2. List Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| append(x) | Adds element x to the end | [1, 2].append(3) → [1, 2, 3] |
| extend(iter) | Appends elements from another iterable | [1, 2].extend([3, 4]) → [1, 2, 3, 4] |
| insert(i, x) | Inserts x at index i | [1, 2].insert(1, 9) → [1, 9, 2] |
| remove(x) | Removes first occurrence of x | [1, 2, 1].remove(1) → [2, 1] |
| pop(i) | Removes and returns element at index i | [1, 2, 3].pop(1) → 2, list becomes [1, 3] |
| index(x) | Returns first index of x | [1, 2, 3].index(2) → 1 |
| count(x) | Counts occurrences of x | [1, 1, 2].count(1) → 2 |
| sort() | Sorts the list in ascending order | [3, 1, 2].sort() → [1, 2, 3] |
| reverse() | Reverses the list | [1, 2, 3].reverse() → [3, 2, 1] |
| copy() | Returns a shallow copy | a.copy() |
| clear() | Removes all items from the list | [1, 2].clear() → [] |

**3. Tuple Methods**

Tuples are **immutable**, so they have fewer methods:

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| count(x) | Counts occurrences of x | (1, 2, 1).count(1) → 2 |
| index(x) | Returns first index of x | (1, 2, 3).index(2) → 1 |

**4. Set Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| add(x) | Adds x to the set | {1, 2}.add(3) → {1, 2, 3} |
| remove(x) | Removes x; error if not found | {1, 2}.remove(2) → {1} |
| discard(x) | Removes x; no error if not found | {1}.discard(2) → {1} |
| pop() | Removes and returns an arbitrary element | {1, 2}.pop() |
| clear() | Removes all elements | {1, 2}.clear() → set() |
| union(set2) | Returns a union of two sets | {1}.union({2}) → {1, 2} |
| intersection(set2) | Returns elements common to both sets | {1, 2}.intersection({2, 3}) → {2} |
| difference(set2) | Returns elements in set1 but not in set2 | {1, 2}.difference({2}) → {1} |
| update(set2) | Adds elements from set2 | {1}.update({2, 3}) → {1, 2, 3} |

**5. Dictionary Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| get(key) | Returns value for key; None if not found | {'a': 1}.get('a') → 1 |
| keys() | Returns all keys | {'a': 1}.keys() → dict\_keys(['a']) |
| values() | Returns all values | {'a': 1}.values() → dict\_values([1]) |
| items() | Returns all key-value pairs as tuples | {'a': 1}.items() → dict\_items([('a', 1)]) |
| update() | Updates with another dictionary | {'a': 1}.update({'b': 2}) → {'a': 1, 'b': 2} |
| pop(key) | Removes key and returns its value | d.pop('a') |
| popitem() | Removes and returns last inserted item (LIFO) | d.popitem() |
| clear() | Removes all items | d.clear() → {} |
| copy() | Returns a shallow copy | new\_dict = d.copy() |
| setdefault(k,v) | Returns value if key exists, else inserts with value | d.setdefault('x', 0) |

**Control Statements (Break, Continue, Pass)**

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

| **Statement** | **Purpose** | **Effect** |
| --- | --- | --- |
| break | Exit the loop completely | Stops loop execution immediately |
| continue | Skip current iteration | Skips to the next iteration |
| pass | Do nothing (placeholder) | Used when a statement is required |

**String Manipulation**

* **Understanding how to access and manipulate strings.**

## **1. Accessing Characters in a String**

You can access individual characters in a string using **indexing**:

text = "Python"

print(text[0]) # 'P' (first character)

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

 Indexing starts at 0 for the first character.

 Negative indices count from the end.

## 2. Slicing Strings

Slicing allows extracting a **substring**:

text = "Python"

print(text[1:4]) # 'yth' (characters from index 1 to 3)

print(text[:3]) # 'Pyt' (from start to index 2)

print(text[3:]) # 'hon' (from index 3 to end)

**syntax**: string[start:stop:step]

## 3. Common String Manipulation Methods

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| lower() | Converts to lowercase | 'Hello'.lower() → 'hello' |
| upper() | Converts to uppercase | 'hello'.upper() → 'HELLO' |
| strip() | Removes leading/trailing whitespace | ' hello '.strip() → 'hello' |
| replace(old, new) | Replaces substring | 'a+b'.replace('+', '-') → 'a-b' |
| split(sep) | Splits string into list | 'a,b,c'.split(',') → ['a', 'b', 'c'] |
| join(iterable) | Joins list into string | '-'.join(['a','b']) → 'a-b' |
| find(sub) | Finds first index of substring | 'hello'.find('e') → 1 |
| count(sub) | Counts occurrences of substring | 'banana'.count('a') → 3 |
| startswith() | Checks prefix | 'hello'.startswith('he') → True |
| endswith() | Checks suffix | 'hello'.endswith('lo') → True |

## 4. String Immutability

Strings in Python are **immutable**, meaning you **cannot change characters** directly:

text = "Python"

new\_text = 'J' + text[1:]

print(new\_text) # 'Jython'

## 5. Looping Through a String

You can iterate through each character:

for char in "cat":

print(char)

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

Strings in Python support a variety of operations that are useful for combining, repeating, and manipulating text data. Below are some of the most common and essential operations.

## **1. Concatenation (+)**

Used to **join two or more strings** together.

first = "Hello"

second = "World"

result = first + " " + second

print(result) # Output: Hello World

## **2. Repetition (\*)**

Used to **repeat a string** multiple times.

word = "Hi"

print(word \* 3) # Output: HiHiHi

* **String slicing.**

**Slicing** allows you to extract a **portion (substring)** of a string using the syntax:

string[start:stop:step]

**Basic Slicing**

text = "Python"

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

print(text[2:5]) # 'tho'

**Shorthand Notation**

| **Slice** | **Meaning** | **Example** |
| --- | --- | --- |
| text[:n] | From start to index n-1 | text[:3] → 'Pyt' |
| text[n:] | From index n to end | text[3:] → 'hon' |
| text[:] | Entire string | text[:] → 'Python' |

**Using Negative Indexes**

Negative indices count from the **end**:

text = "Python"

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

print(text[-3:-1]) # 'ho'

print(text

**Advanced Python (map(), reduce(), filter(), Closures and Decorators)**

* **How functional programming works in Python.**

| **Feature** | **Description** |
| --- | --- |
| Pure Functions | No side effects, predictable output |
| First-Class Functions | Functions as variables/arguments |
| Lambdas | Anonymous, inline functions |
| map, filter, reduce | Functional data processing |
| Recursion | Function calls itself |
| Immutability | Data should not change after creation |

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

## **1. map() – Apply a Function to Each Element**

**Purpose:** Transforms each item in an iterable using a given function.

**Syntax:**

map(function, iterable)

### Example: Squaring a List of Numbers

nums = [1, 2, 3, 4]

squared = list(map(lambda x: x\*\*2, nums))

print(squared) # [1, 4, 9, 16]

## **2. filter() – Filter Items Based on a Condition**

**Purpose:** Keeps only elements where the function returns True.

**Syntax:**

filter(function, iterable)

### Example: Filtering Even Numbers

nums = [1, 2, 3, 4, 5, 6]

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens) # [2, 4, 6]

## **3. reduce() – Apply a Function Cumulatively**

**Purpose:** Reduces the iterable to a single value by applying a function.

**Syntax:**

from functools import reduce

reduce(function, iterable)

### Example: Summing All Numbers

from functools import reduce

nums = [1, 2, 3, 4, 5]

total = reduce(lambda x, y: x + y, nums)

print(total) # 15

### Example: Finding the Maximum Value

max\_num = reduce(lambda x, y: x if x > y else y, nums)

print(max\_num) # 5

* **Introduction to closures and decorators.**

## **1. Closures**

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

### Structure of a Closure

def outer(x):

def inner(y):

return x + y # x is remembered from outer

return inner

add\_five = outer(5) # x = 5 is enclosed

print(add\_five(10)) # Output: 15

### Key Points:

* inner() has access to x from outer(), even after outer() is done.
* Closures help in **data hiding** and **function factories**.

## **2. Decorators**

A **decorator** is a function that **wraps another function** to add or modify its behavior, without changing its source code.

### Basic Decorator Example

def my\_decorator(func):

def wrapper():

print("Before the function runs")

func()

print("After the function runs")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

**Output:**

Before the function runs

Hello!

After the function runs

### Explanation:

* @my\_decorator is **syntactic sugar** for:  
  say\_hello = my\_decorator(say\_hello)
* wrapper() adds behavior **before and after** the original function.