PYTHON PROGRAMMING: UNIT 2

Prof (Dr) Bishwajeet Kumar Pandey

Department of MCA

GL Bajaj Institute of Technology and Management, Greater Noida, India

UNIT-2 Syllabus

- String Manipulation: Accessing Strings, Basic Operations, String slices, Function and Methods.
- Lists: Introduction, Accessing list, Operations, Working with lists, Function and Methods.
- Tuple: Introduction, accessing tuples, Operations, Working, Functions and Methods.

String Manipulation

- 1. What is a String in Python?

- A string is a sequence of characters enclosed in *single quotes ('), double quotes ("), or triple quotes (''' or """)*.*
- Strings are *immutable, meaning once created, they **cannot be changed*.
- Examples:
- string1 = 'Hello’
- string2 = "World“
- string3 = '''Python is fun'‘’
- print(string1) # Output: Hello
- print(string2) # Output: World
- print(string3) # Output: Python is fun

Accessing Strings

- To access individual characters or parts of a string, Python uses

- indexing
- slicing

Indexing

- Each character in a string has a *position (index).
- Index starts from 0 for the first character.
- Negative indexing is also allowed:
- -1 represents the *last character,
- -2 represents the second-to-last character, and so on.

Example of Positive and Negative Indexing

```
text = “Python”

Positive Indexing

print(text[0])  # Output: P (first character)

print(text[3])  # Output: h (fourth character)

Negative Indexing

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

print(text[-3])  # Output: h (third last character)
```

Slicing in String

Slicing allows you to extract a substring from a string using the syntax:

string[start:end:step]

start → Starting index (inclusive, default = 0)

end → Ending index (exclusive, default = length of string)

step → Interval between characters (default = 1)

Examples of Slicing

```
text = "Programming“

# Basic Slicing

print(text[0:6])   # Output: Progra (from index 0 to 5)

print(text[:6])    # Output: Progra (start is default 0)

print(text[6:])    # Output: mming  (till the end)#

Negative Index Slicing

print(text[-7:-1]) # Output: ramming (from index -7 to -2)

Step Slicing

print(text[0:11:2]) # Output: Pormig (every 2nd character)

print(text[::-1])   # Output: gnimmargorP (reversed string)
```

Key Points about String

- Strings are immutable → you cannot modify characters directly.
- Indexing helps to access *single characters.
- Slicing helps to access *substrings or patterns.
- Negative indices are useful for working from the end of the string.

Practical Example of String

name = "Dr. Bishwajeet Pandey“

Accessing specific characters

- print("First character:", name[0]) # Output: D
- print("Last character:", name[-1]) # Output: y

Extracting a substring

- print("First name:", name[4:14]) # Output: Bishwajeet

Reverse the string

- print("Reversed:", name[::-1]) # Output: yednaP teejawhsiB .rD

Practical Example of String

Access 1st char	text[0]
Access last char	text[-1]
Access Characters 0–4	text[0:5]
Every 2nd char	text[::2]
Reverse string	text[::-1]

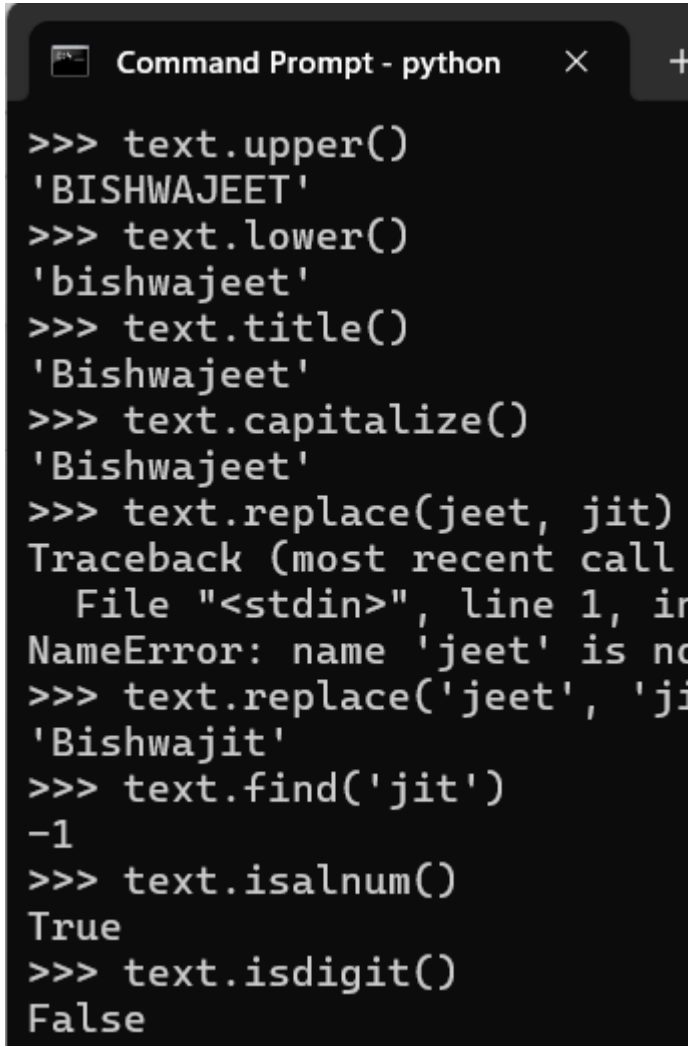
Common String Functions

- len(string) → returns length of the string.
- max(string) → returns the maximum character (based on Unicode value).
- min(string) → returns the minimum character.
- sorted(string) → returns a sorted list of characters.
- str() → converts data into a string.

```
>>> text="Bishwajeet"
>>> len(text)
10
>>> max(text)
'w'
>>> min(text)
'B'
>>> sorted(text)
['B', 'a', 'e', 'e', 'h', 'i', 'j', 's', 't', 'w']
>>> str(5)
'5'
>>>
```

Common String Methods

- string.upper() → converts to uppercase.
- string.lower() → converts to lowercase.
- string.title() → capitalizes first letter of each word.
- string.capitalize() → capitalizes first letter of string.
- string.strip() → removes spaces (leading and trailing).
- string.replace(old, new) → replaces substring.
- string.find(sub) → returns first index of substring, -1 if not found.
- string.count(sub) → counts occurrences of substring.
- string.isalpha() → checks if all characters are alphabets.
- string.isdigit() → checks if all characters are digits.
- string.isalnum() → checks if all characters are alphanumeric.
- string.startswith(sub) / string.endswith(sub) → check beginning/ending.
- string.split() → splits into list of words.
- string.join(iterable) → joins elements with string as separator.



Python Program: Work with 2 Strings & Check Palindrome

```
# Program to check palindrome using two strings
# Input strings
str1 = input("Enter first string: ")
str2 = input("Enter second string: ")
# Combine the two strings
combined = str1 + str2
print("Combined String:", combined)
# Check if palindrome
if combined == combined[::-1]:
print("The combined string is a Palindrome!")
else:
print("The combined string is NOT a Palindrome.")
```

Python Lists

A **list** in Python is an **ordered, mutable collection** of elements.

- It can store **different data types** (int, float, string, etc.).
- Lists are written inside **square brackets** [ ].
- Example: my_list = [10, "hello", 3.5]

Create a list and perform basic operations (Insert, Delete, Update)

- # Create a list
- numbers = [10, 20, 30, 40]
- print("Original List:", numbers)

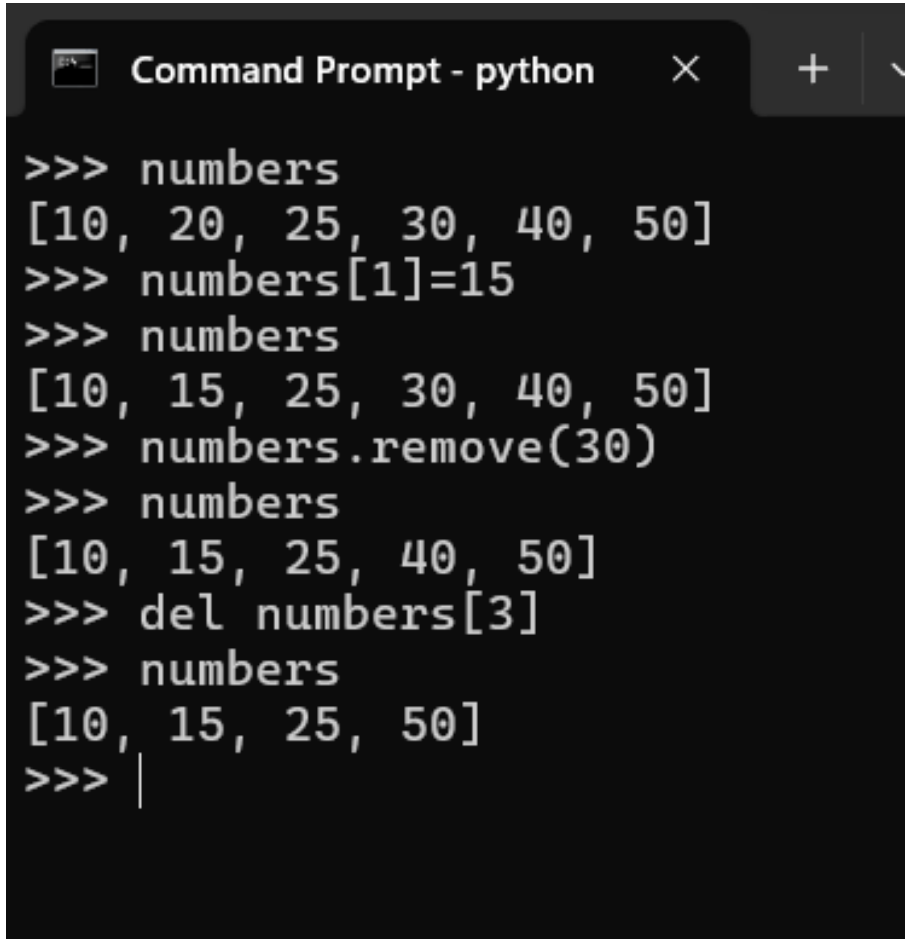
- # Insert (append and insert at index)
- numbers.append(50) # add at end
- numbers.insert(2, 25) # insert 25 at index 2
- print("After Insertion:", numbers)

```
>>> numbers=[10,20,30,40]
>>> print("Original List: ",numbers)
Original List:  [10, 20, 30, 40]
>>> numbers.append(50)
>>> print("Original List: ",numbers)
Original List:  [10, 20, 30, 40, 50]
>>> numbers.append(2,25)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list.append() takes exactly one argument (2 given)
>>> numbers.insert(2,25)
>>> print("Original List: ",numbers)
Original List:  [10, 20, 25, 30, 40, 50]
>>> print("Original List: ",numbers[0])
Original List:  10
>>> print("Original List: ",numbers[-1])
Original List:  50
>>> print("Original List: ",numbers[-2])
Original List:  40
>>>
```

Create a list and perform basic operations (Insert, Delete, Update)

- # Update (change value at index)
- numbers[1] = 15
- print("After Update:", numbers)

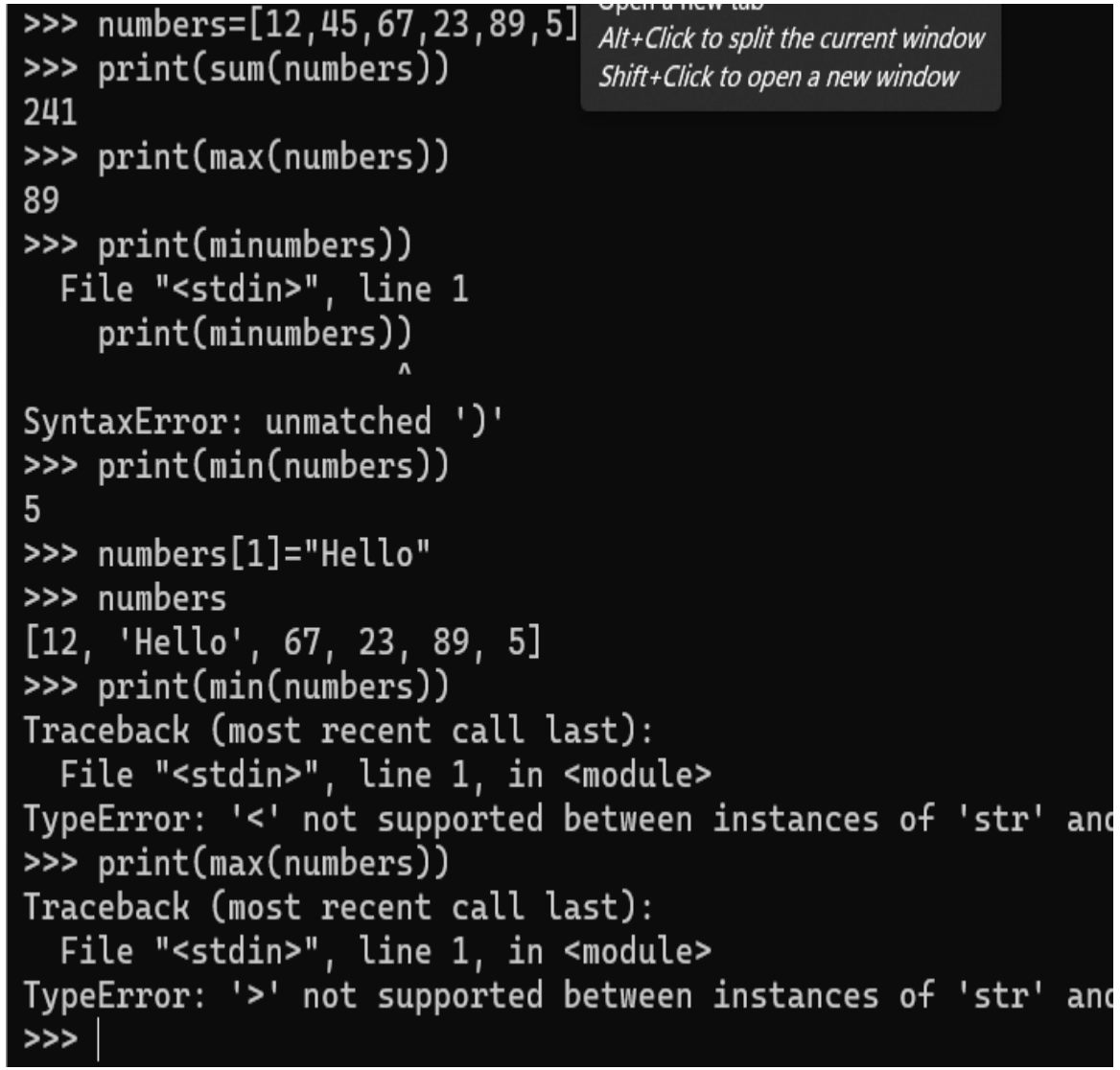
- # Delete (by value and by index)
- numbers.remove(30) # remove element 30
- del numbers[3] # delete element at index 3
- print("After Deletion:", numbers)



Find the sum, maximum, and minimum of list elements

- numbers = [12, 45, 67, 23, 89, 5]

- print("List:", numbers)
- print("Sum:", sum(numbers))
- print("Maximum:", max(numbers))
- print("Minimum:", min(numbers))



Sort a list in ascending and descending order

```
numbers = [34, 12, 56, 78, 1, 90]
print("Original List:", numbers)
print("Ascending Order:", sorted(numbers))
print("Descending Order:", sorted(numbers, reverse=True))
# In-place sort
numbers.sort()
print("Sorted Ascending (in-place):", numbers)
numbers.sort(reverse=True)
print("Sorted Descending (in-place):", numbers)
```

Concepts Used:

sorted(list) → returns a new sorted list.

.sort() → sorts the list **in place**.

Count even and odd numbers in a list

```
numbers = [10, 23, 45, 66, 72, 89, 90]
even_count = 0
odd_count = 0
for num in numbers:
if num % 2 == 0:
even_count += 1
else:
odd_count += 1
print("List:", numbers)
print("Even Numbers:", even_count)
print("Odd Numbers:", odd_count)
```

Linear Search using a list

```
numbers = [12, 34, 56, 78, 90, 23]
key = int(input("Enter number to search: "))
found = False
for i in range(len(numbers)):
if numbers[i] == key:
print(f"Element {key} found at index {i}")
found = True
break
if not found:
print("Element not found in the list")
```

Merge two lists

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
merged = list1 + list2
print("Merged List:", merged)
# Alternative using extend()
list1.extend(list2)
print("List1 after extending:", list1)
```

Concepts Used:

+ operator concatenates lists.

.extend() adds elements of another list.

Iteration using a list with a for loop

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

print("Iterating over list elements:")

for fruit in fruits:

print(fruit)
```

Concepts Used:

for loop iterates directly over list elements.

Lists are iterable in Python.

Tuples in Python

A **tuple** is an ordered, immutable collection of elements in Python.

Defined using **parentheses** ( ).

Tuples can store **heterogeneous data** (integers, strings, lists, etc.).

Since tuples are **immutable**, once created, their elements **cannot be changed**.

Tuples in Python

```
my_tuple = (10, "apple", 3.14, True)
```

```
print(my_tuple)
```

Accessing Tuples

Access elements using **indexing** (starts at 0).

Can use **negative indexing** (from end).

Supports **slicing** to access a range.

```
t = (1, 2, 3, 4, 5)
```

```
print(t[0])     # First element → 1
```

```
print(t[-1])    # Last element → 5
```

```
print(t[1:4])   # Slice → (2, 3, 4)
```

Operations on Tuples

Concatenation (+) → join tuples.

Repetition (*) → repeat tuple elements.

Membership (in, not in) → check if element exists.

Iteration → use loops.

Operations on Tuples

```
t1 = (1, 2, 3)
```

```
t2 = (4, 5)
```

```
print(t1 + t2)      # (1, 2, 3, 4, 5)
```

```
print(t1 * 2)       # (1, 2, 3, 1, 2, 3)
```

```
print(2 in t1)      # True
```

Working with Tuples

- Tuples can be **nested** (tuple inside tuple).
- Used for **multiple assignment** (packing & unpacking).

```
# Packing
student = ("Alice", 21, "CS")

# Unpacking
name, age, course = student
print(name, age, course)  # Alice 21 CS

# Nested tuple
nested = (1, (2, 3), (4, 5))
print(nested[1][1])  # 3
```

Functions for Tuples

- len(tuple) → length
- max(tuple) → maximum element
- min(tuple) → minimum element
- sum(tuple) → sum of numeric elements
- sorted(tuple) → returns a sorted list

t = (10, 20, 5, 15)

- print(len(t)) # 4
- print(max(t)) # 20
- print(min(t)) # 5
- print(sum(t)) # 50
- print(sorted(t)) # [5, 10, 15, 20]

Tuple Methods

Unlike lists, tuples have very few methods since they are immutable.

- .count(value) → returns number of occurrences.

- .index(value) → returns first index of value.

- t = (1, 2, 3, 2, 4, 2)

- print(t.count(2)) # 3
- print(t.index(3)) # 2

List vs Tuple in Python

Feature	**List**	**Tuple**
Definition	Ordered, **mutable** collection	Ordered, **immutable** collection
Syntax	Square brackets `[ ]`	Parentheses `( )`
Mutability	Can be updated (insert, delete, modify elements)	Cannot be changed after creation
Methods Available	Many (`append()`, `extend()`, `pop()`, `remove()`, `sort()`, `reverse()` etc.)	Very few (`count()`, `index()`)
Performance	Slower (due to mutability and overhead of dynamic operations)	Faster (lightweight, fixed structure)
Iteration Speed	Comparatively slower	Faster
Use Case	When data **needs modification** (e.g., dynamic data, CRUD operations)	When data should remain **constant** (e.g., fixed records, keys in dictionaries)
Memory Usage	Uses **more memory**	Uses **less memory**
Hashable (can be dictionary key)?	❌ No (because mutable)	✅ Yes (if all elements are immutable)