UNIT 3

List

A **list** in Python is a built-in sequence data type that can hold an **ordered and mutable collection of items** in a single variable.

Features of Lists

- **Ordered** – Maintains the order of items as they are added.
- **Mutable** – Elements can be changed, added, or removed after creation.
- **Index-based** – Items are accessed by position, starting from index 0.
- **Allows Duplicates** – Same value can appear multiple times.
- **Mixed Data Types** – Can store different types of values (integers, strings, floats, Booleans, even other lists).

Creating List

Lists are created using square brackets **[ ],** with items separated by **commas(,)** inside the list.

Syntax:

```
list_name = [item_1, item_2, item_3……., item_n]
```

Examples

1. Creating number list

```
number_list = [1,2,3,5,6,7,8,9,10]
print(number_list,)
print(type(number_list))
```

Output:

```
[1,2,3,5,6,7,8,9,10]
<class 'list'>
```

2. Different data types in a single list.

```
mixed_list = [1,2.8,3+6j,True,"BCAAI"]
print(mixed_list,type(mixed_list))
```

Output:

```
[1, 2.8, (3+6j), True, 'BCAAI'] <class 'list'>
```

3. Creating nested list

```
nested_list = [1, [2, 3], [4, 5]]
print(nested_list)
print(type(nested_list))
```

Output:

```
[1, [2, 3], [4, 5]]
<class 'list'>
```

Creating an Empty List

An empty list is a list that **does not contain any items**.

Syntax:

```
empty_list = []
```

Example:

```
empty_list = [] # create empty list
print(empty_list)
print(type(empty_list))
```

Output:

```
[]
<class 'list'>
```

Creating a List using list( )

In Python, lists can also be created by using the built-in function **list( ).**
It is commonly used for **type conversion**, that is converting other data types (like tuple, string, or set) into a list.

Syntax:

```
list_name = list(iterable)
```

- **iterable** → any sequence (tuple, string, set, or another list) that will be converted into a list.

Examples:

1. From a tuple (type conversion)

```
numbers = list((1, 2, 3, 4))
print(numbers)
```

Output:

```
[1, 2, 3, 4]
```

2. From a string (type conversion)

```
letters = list("BCAAI")
print(letters)
```

Output:

```
['B', 'C', 'A','A','I']
```

3. From a set (type conversion)

```
myset = {10, 20, 30}
converted = list(myset)
print(converted)
```

Output:

```
[10, 20, 30]
```

4. Empty list using list( )

```
empty = list( )
print(empty)
```

Output:

```
[ ]
```

Basic List Operations in Python

1. Concatenation (+): Combines two or more lists into a single list.

Syntax:

```
new_list = list1 + list2
```

Example:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

2. Repetition (*): Repeats the elements of a list a specified number of times.

Syntax:

```
new_list = list1 * n
```

Example:

```
a = [1, 2]
print(a * 3)
```

Output:

```
[1, 2, 1, 2, 1, 2]
```

3. Membership (in / not in): Checks if an element exists or does not exist in a list.

Syntax:

```
item in list_name
item not in list_name
```

Example:

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)
print("mango" not in fruits)
```

Output:

```
True
True
```

4. Comparison (= =, ! =): Compares two lists for equal (= =) or inequal (! =).

Syntax:

```
list1 == list2
list1 != list2
```

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = [3, 2, 1]
print(a == b)
print(a != c)
```

Output:

```
True
True
```

Indexing and slicing in Lists

Indexing in Lists

- Indexing is process of accessing individual items of a list using their position number (index).
- Indexing starts from 0 for the first element.
- Negative indexing starts from -1 for the last element.

Syntax:

```
list_name[index]
```

Example:

```
fruits = ["apple", "banana", "cherry", "mango"]
print(fruits[0])
print(fruits[1])
print(fruits[2])
```

Output:

```
apple
banana
cherry
```

Positive and Negative Indexing in Lists

1. Positive Indexing

- Positive indexing starts from 0 for the first element of the list.
- The index increases by 1 for each next element.

Syntax:

```
list_name[index]
```

Example:

```
fruits = ["apple", "banana", "cherry", "mango"]
print(fruits[0])  # apple (first element)
print(fruits[2])  # cherry (third element)
```

Index Positions (Positive):

```
fruits = ["apple", "banana", "cherry", "mango"]
           0         1          2         3
```

2. Negative Indexing

- Negative indexing starts from -1 for the last element of the list.
- The index decreases by 1 for each previous element.

Syntax:

```
list_name[-index]
```

Example:

```
fruits = ["apple", "banana", "cherry", "mango"]
print(fruits[-1])  # mango (last element)
print(fruits[-3])  # banana (second element)
```

Index Positions (Negative):

```
fruits = ["apple", "banana", "cherry", "mango"]
           -4        -3         -2        -1
```

Slicing in Lists

Slicing in Python is the process of extracting a portion (subsequence) of a list by specifying a range of index positions.

Syntax:

```
list_name[start : end : step]
```

- `start` → index position where the slice begins (default = 0)
- `end` → index position where the slice ends (**excluded**)
- `step` → interval/jump between elements (default = 1)

Examples:

```
numbers = [10, 20, 30, 40, 50, 60, 70]# 1. Simple slicing
print(numbers[1:4])    # Output: [20, 30, 40]

# 2. Omitting start (defaults to 0)
print(numbers[:3])     # Output: [10, 20, 30]

# 3. Omitting end (till last element)
print(numbers[3:])     # Output: [40, 50, 60, 70]

# 4. Using step
print(numbers[::2])    # Output: [10, 30, 50, 70]

# 5. Negative slicing (reverse list)
print(numbers[::-1])   # Output: [70, 60, 50, 40, 30, 20, 10]
```

WAP to check given word is palindrome or not

```
list = list(input("Enter the word:"))
list_rev = list[ : :-1]
if list == list_rev:
    print("Given word is palindrome")
else:
    print("Given word is not palindrome")
```

Output: case: true
```
Enter the word: madam
Given word is palindrome
```

Output: case: false
```
Enter the word: sir
Given word is not palindrome
```

Built-in Functions Used on List

1. len(): This function returns the number of elements in a list.

Example:

```
fruits = ["apple", "banana", "cherry", "mango"]
print(len(fruits))
```

Output:
```
4
```

2. min(): This function returns the smallest element in a list.

Example:

```
numbers = [5, 8, 2, 10, 3]
print(min(numbers))
```

Output:
```
2
```

3. max(): This function returns the largest element in a list.

Example:

```
numbers = [5, 8, 2, 10, 3]
print(max(numbers))
```

Output:
```
10
```

4. sum(): This function returns the sum of all numeric elements in a list.

Example:

```
numbers = [5, 10, 15, 20]
print(sum(numbers))
```

Output:
```
50
```

5. any(): This function returns **True** if at least one element in the list is **non-zero**. If all elements are **zero,** or **an empty list** it returns **False**.

Example:

```
values = [0, 0, 3, 0]
print(any(values))
```

Output:
```
True
```

6. all(): This function returns **True** if all elements in the list are **non-zero**. If any element is **zero**, or **an empty list**, it returns **False**

Example:

```
values1 = [1, 2, 3, 4]
values2 = [1, 0, 3, 4]
print(all(values1))
print(all(values2))
```

Output:
```
True
False
```

7. sorted(): This function returns a new list with the elements sorted in ascending order.

Example:

```
numbers = [40, 10, 30, 20]
print(numbers)
print(sorted(numbers))
print(sorted(numbers, reverse=True))
```

Output:
```
[40, 10, 30, 20]
[10, 20, 30, 40]
[40, 30, 20, 10]
```

Function	Definition	Example	Output
len()	Returns the number of elements in a list.	`fruits = ["apple","banana","cherry"]` `print(len(fruits))`	3
min()	Returns the smallest element in a list.	`numbers = [5,8,2,10,3]` `print(min(numbers))`	2
max()	Returns the largest element in a list.	`numbers = [5,8,2,10,3]`	10

Function	Definition	Example	Output
		`print(max(numbers))`	
sum()	Returns the sum of all numeric elements in a list.	`numbers = [5,10,15,20]` `print(sum(numbers))`	50
any()	Returns True if at least one element is true/non-zero; otherwise False.	`values = [0,0,3,0]` `print(any(values))`	True
all()	Returns True if all elements are true/non-zero; otherwise False.	`values = [1,2,3,4]` `print(all(values))`	True
sorted()	Returns a new list with elements sorted in ascending order.	`numbers = [40,10,30,20]` `print(sorted(numbers))`	[10, 20, 30, 40]

List Methods

Python provides several built-in **methods** that can be used to manipulate lists. Unlike functions, **methods are called on the list object**.

1. append(): this method is used to add a single element to the end of the list.

Syntax:

```
list_name.append(element)
```

Example:

```
numbers = [10, 20, 30]
numbers.append(40)
print(numbers)
```

Output:

```
[10, 20, 30, 40]
```

2. insert(): this method is used to inserts an element at a specific index.

Syntax:

```
list_name.insert(index, element)
```

Example:

```
numbers = [10, 20, 30]
numbers.insert(1, 15)
print(numbers)
```

Output:

```
[10, 15, 20, 30]
```

3. extend(): this method is used to add multiple elements (from another list or iterable) to the end of the list.

Syntax:

```
list_name.extend(iterable)
```

Example:

```
numbers = [10, 20]
numbers.extend([30, 40])
print(numbers)
```

Output:

```
[10, 20, 30, 40]
```

4. remove(): this method is used removes the first occurrence of the specified element.

Syntax:

```
list_name.remove(element)
```

Example:

```
numbers = [10, 20, 30, 20]
numbers.remove(20)
print(numbers)
```

Output:

```
[10, 30, 20]
```

5. pop():this method removes and returns the element at a specified index. If no index is given, removes the last element.

Syntax:

```
list_name.pop(index)
```

Example:

```
numbers = [10, 20, 30, 40]
print(numbers.pop())    # Removes last element
```

Output:

```
40
20
[10, 30]
```

```
print(numbers.pop(1))  # Removes element at index 1
print(numbers)
```

6. clear(): this method is used to removes all elements from the list.
Syntax:

```
list_name.clear()
```

Example:

```
numbers = [10, 20, 30]
numbers.clear()
print(numbers)
```

Output:
```
[ ]
```

7. index(): this method is used returns the index of the first occurrence of the specified element.
Syntax:

```
list_name.index(element)
```

Example:

```
numbers = [10, 20, 30, 20]
print(numbers.index(20))
```

Output:
```
1
```

8. count(): this method is used to returns the number of times a specified element appears in the list.
Syntax:

```
list_name.count(element)
```

Example:

```
numbers = [10, 20, 30, 20]
print(numbers.count(20))
```

Output:
```
2
```

9. sort(): this method is used to sorts the list in ascending order by default. Can sort in descending order using reverse=True.
Syntax:

```
list_name.sort(reverse=False)
```

Example:

```
numbers = [40, 10, 30, 20]
numbers.sort()
print(numbers)  # Ascending
numbers.sort(reverse=True)
print(numbers)  # Descending
```

Output:
```
[10, 20, 30, 40]
[40, 30, 20, 10]
```

10. reverse(): this method is used to reverses the elements of the list in place.
Syntax:

```
list_name.reverse()
```

Example:

```
numbers = [10, 20, 30, 40]
numbers.reverse()
print(numbers)
```

Output:
```
[40, 30, 20, 10]
```

Updating and Deleting List items by indexing and slicing

Updating List items by Indexing

- means changing the value of an existing element in a list by referring to its index position.

Syntax:

```
list_name[index] = new_value
```

Example

```
numbers = [10, 20, 30, 40, 50]
print("Before update:", numbers)
numbers[1] = 25
print("After update:", numbers)
```

Output:
```
Before update: [10, 20, 30, 40, 50]
After update: [10, 25, 30, 40, 50]
```

Updating List items by Slicing

- means modifying multiple elements at once by assigning new values to a slice of list **elements.**

Syntax:
```
list_name[start:end] = [new_values]
```

Example:

```
numbers = [10, 20, 30, 40, 50]
print("Before update:", numbers)
numbers[1:4] = [21, 31, 41]
print("After update:", numbers)
```

Output:
```
Before update: [10, 20, 30, 40, 50]
After update: [10, 21, 31, 41,50]
```

Deleting List items by Indexing

- Means removing the value of an existing element in a list by referring to its index position.
- Using del statement, we can remove the items in list by referring index position.

Syntax:
```
del list_name[index]
```

Example

```
fruits = ['apple', 'banana', 'cherry', 'mango']
print("Before deletion:", fruits)

del fruits[1]
print("After deletion:", fruits)
```

Output:
```
Before deletion: ['apple', 'banana', 'cherry', 'mango']
After deletion: ['apple', 'cherry', 'mango']
```

Deleting List items by Slicing

- Means delete multiple items at once using slicing.

Syntax:
```
del list_name[start:end]
```

Example

```
colors = ['red', 'green', 'blue', 'yellow', 'black', 'white']
print("Before deletion:", colors)

del colors[1:4]
print("After deletion:", colors)
```

Output:

```
Before deletion: ['red', 'green', 'blue', 'yellow', 'black', 'white']
After deletion: ['red', 'black', 'white']
```

Populating and Traversing List in Python

1. Populating a List

- Populating a list means filling the list with elements or data.

Example:

```
numbers = []
# Populating the list with values
for i in range(1, 6):
    numbers.append(i)
print("Populated List:", numbers)
```

Output:
```
Populated List: [1, 2, 3, 4, 5]
```

2. Traversing a List

- Traversing a list means accessing or visiting each element of the list one by one in sequence.
- Purpose: To read, display, or perform operations on all elements of the list.

Example:

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

Output:
```
apple
banana
cherry
```

Implementation of stacks and queues using list:

Stack Implementation using List

Stack: A **stack** is a linear data structure that follows **LIFO (Last In First Out)** principle – the insertion and deletion of element take place at same end called **TOP**

Operations:

- **push()** → add element to the top of the stack
- **pop()** → remove element from the top of the stack
- **Display()** → Display the elements

Stack Implementation program:

```
stack = []
n=int(input("Enter the Lenth of stack: "))
def push():
    if len(stack) == n:
        print("Stack is full! Cannot push.")
    else:
        element = input("Enter element to push: ")
        stack.append(element)
        print(f"{element} pushed onto the stack.")
def pop():
    if len(stack) == 0:
            print("Stack is empty! Cannot pop.")
    else:
        popped = stack.pop()
        print(f"{popped} is popped from stack.")
def display():
    if len(stack) == 0:
            print("Stack is empty!")
    else:
        print("Stack elements (top to bottom):")
        for item in stack[::-1]:
            print(item)
while True:
    print("\nChoose an operation:")
    print("1. Push")
    print("2. Pop")
    print("3. Display Stack")
    print("4. Exit")
    choice = int(input("Enter your choice: "))
    if choice == 1:
```

```
        push()
    elif choice == 2:
        pop()
    elif choice == 3:
        display()
    elif choice == 4:
        print("Exiting stack operations.")
        stack.clear()
        break
    else:
        print("Invalid choice! Please enter a number between 1 and 5.")
```

Queue Implementation using List

Queue: A **queue** is a linear data structure that follows **FIFO (First In First Out)** principle – insertion takes place at one end (rear) and deletion takes place at the other end (front).

Operations:

- **enqueue()** → add element to the rear of the queue
- **dequeue()** → remove element from the front of the queue

Queue implementation program:

```
queue = []
n=int(input("Enter the lenth of queue: "))
def enqueue():
    if len(queue) == n:
        print("Queue is full! Cannot enqueue.")
    else:
        element = input("Enter element to enqueue: ")
        queue.append(element)
        print(f"{element} is enqueued to the queue.")
def dequeue():
    if len(queue) == 0:
            print("Queue is empty! Cannot dequeue.")
    else:
        removed = queue.pop(0)
        print(f"{removed} is dequeued from queue.")
def display():
    if len(queue) == 0:
            print("queue is empty!")
    else:
        print("Queue elements are :")
        for item in queue:
            print(item,end = " ")
while True:
    print("\nChoose an operation:")
    print("1. Enqueue")
    print("2. Dequeue")
    print("3. Display Queue")
    print("4. Exit")
    choice = int(input("Enter your choice: "))
    if choice == 1:
        enqueue()
    elif choice == 2:
        dequeue()
    elif choice == 3:
        display()
    elif choice == 4:
        print("Exiting queue operations.")
        queue.clear()
        break
    else:
        print("Invalid choice! Please enter a number between 1 and 4.")
```

Nested List

A **nested list** is a list that contains one or more lists as its elements. It is often used to represent data in tabular or matrix form.

Syntax:

```
nested_list = [[element1, element2, ...],
               [element3, element4, ...],
               ...]
```

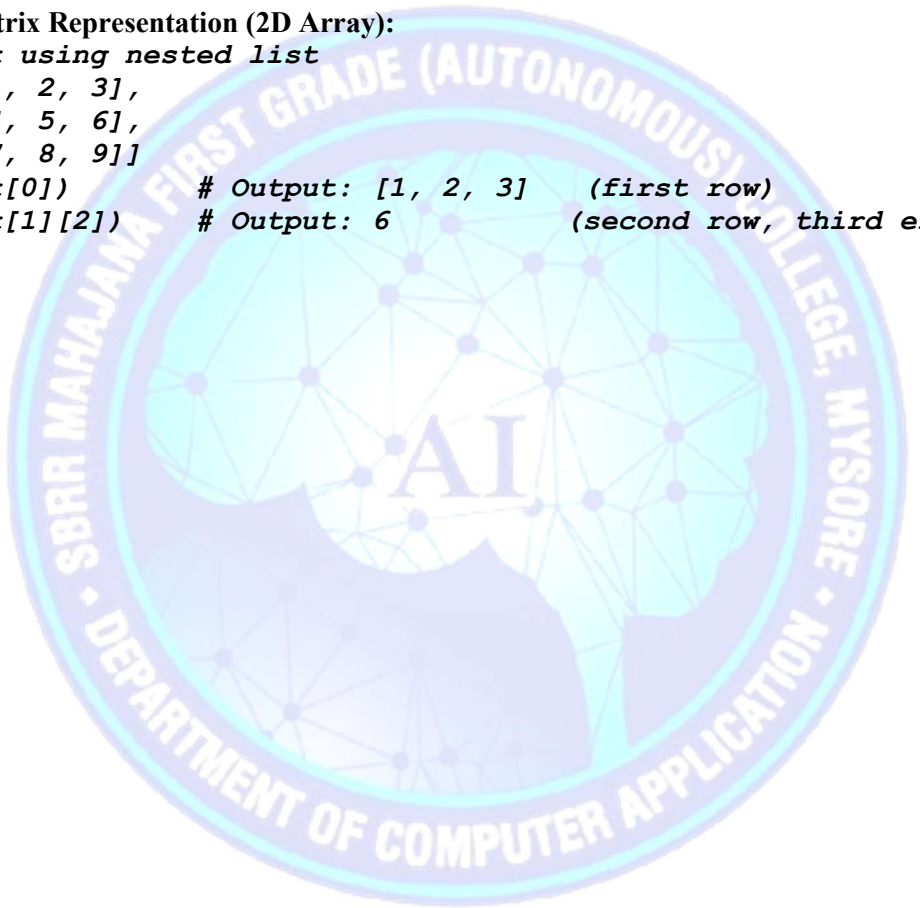
Example 1 – Student Marks:

```
student_marks = [["Alice", 85, 90, 95],
                 ["Bob", 75, 80, 82],
                 ["Charlie", 92, 88, 91]]

print(student_marks[0])       # Output: ['Alice', 85, 90, 95]
print(student_marks[1][1])    # Output: 75
```

Example 2 – Matrix Representation (2D Array):

```
# 3x3 matrix using nested list
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
print(matrix[0])      # Output: [1, 2, 3]   (first row)
print(matrix[1][2])   # Output: 6           (second row, third element)
```



Dictionary

A **dictionary** in Python is a built-in **mapping data type** and an **unordered collection** that stores data in the form of **key–value pairs**, where each **key** is **unique and immutable**, and the **values are mutable**.

Features of Dictionary

- **Unordered (before 3.7)**, **Ordered (from 3.7+)** – Maintains insertion order from Python 3.7 onward.
- **Keys are unique** – No two keys in a dictionary can have the same name.
- **Keys are immutable** – Keys can be **strings**, **numbers**, or **tuples**, but not **lists** or other mutable types.
- **Values are mutable** – The values of a dictionary can be **changed or updated** after creation.
- **Supports mixed data types** – Both **keys and values** can be of **different data types**.

Creating Dictionary

Dictionaries are created using **curly braces { }**, with each **key–value pair separated by a colon (:)**, and pairs separated by commas (,).

Syntax:

```
dict_name = {key_1: value_1, key_2: value_2, ..., key_n: value_n}
```

Examples

1. Creating a simple dictionary

```
student = {"name": "Ajay", "age": 22, "course": "BCAAI"}
print(student)
print(type(student))
```

Output:

```
{'name': 'Ajay', 'age': 22, 'course': 'BCAAI'}
<class 'dict'>
```

2. Mixed data types in a dictionary

```
info = {"id": 101, "name": "Ajay", "marks": [85, 90, 88], "passed": True}
print(info)
```

Output:

```
{'id': 101, 'name': 'Alice', 'marks': [85, 90, 88], 'passed': True}
```

3. Creating a nested dictionary

```
student = {
    "name": "Bob",
    "details": {"age": 21, "course": "BCAAI"},
    "marks": {"Python": 90, "Maths": 88}
}
print(student)
```

Output:

```
{'name': 'Bob', 'details': {'age': 21, 'course': 'BCAAI'}, 'marks': {'Python':
90, 'Maths': 88}}
```

Creating an Empty Dictionary

An empty dictionary does not contain any key–value pairs.

Example

```
empty_dict = {}
print(empty_dict)
print(type(empty_dict))
```

Output:

```
{}
<class 'dict'>
```

Creating Dictionary using dict() Constructor

- Dictionaries can also be created using the **built-in function dict()**.
- It can convert sequences (like lists or tuples) of key–value pairs into a dictionary.

Syntax:

```
dict_name = dict(iterable)
```

- **iterable** → Any sequence containing pairs of items (key, value).

Examples:

1. From a list of tuples

```
student = dict([("name", "Pooja"), ("age", 19), ("course", "BCAAI")])
print(student)
```

Output:

```
{'name': 'Pooja', 'age': 19, 'course': 'BCAAI'}
```

2. From keyword arguments

```
info = dict(name="Ravi", age=22, course="BCAAI")
print(info)
```

Output:

```
{'name': 'Ravi', 'age': 22, 'course': 'BCAAI'}
```

3. Creating an empty dictionary using dict()

```
empty = dict()
print(empty)
```

Output:

```
{}
```

Operations on Dictionary

1. Accessing Items

Dictionary items are accessed using their **keys** inside square brackets [ ] .

Syntax:

```
dict_name[key]
```

Example

```
student = {"name": "John", "age": 20, "course": "BCAAI"}
print(student["name"])
print(student["course"])
```

2. Adding Elements

New key–value pairs can be added to a dictionary by assigning a value to a new key name using the assignment operator =.

Syntax:

```
dictionary_name[new_key] = value
```

Example:

```
student = {"name": "Raju", "age": 20, "course": "BCAAI"}
student["marks"] = 89
print(student)
```

Output:

```
{'name': 'Raju', 'age': 20, 'course': 'BCAAI', 'marks': 89}
```

3. Updating Items

Existing dictionary values can be **modified (updated)** by assigning a new value to an **existing key** using the assignment operator = .

Syntax:

```
dictionary_name[existing_key] = new_value
```

Example:

```
student = {"name": "Raju", "age": 23, "course": "BCAAI", "marks": 89}
student["age"] = 21
print(student)
```

Output:

```
{'name': 'Raju', 'age': 21, 'course': 'BCAAI', 'marks': 89}
```

4. Membership Operator

Checks if a key exists or does not exist in a dictionary.

Syntax:

```
key in dictionary_name
key not in dictionary_name
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print('name' in student)
print('marks' in student)
print('course' not in student)
print('age' not in student)
```

Output:

```
True
False
False
False
```

5. Equality Operator (= =)

Two dictionaries are **equal** if they have:

- The **same keys**, and
- The **same corresponding values**, regardless of the order of key–value pairs.

Example:

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 2, 'a': 1, 'c': 3}
print(dict1 == dict2)
```

Output:

```
True
```

6. Not Equal Operator (! =)

Returns **True** if the dictionaries differ in **any** key or value.

Example:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 1, 'b': 3}

print(dict1 != dict2)
```

Output:

```
True
```

Note: Because the value of 'b' is different.

Built-in functions on Dictionary

1. len(): Returns the **number of key–value pairs** present in the dictionary.

Syntax:

```
len(dictionary)
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print(len(student))
```

Output:
```
3
```

2. all( ): Returns **True** if **all keys** in the dictionary are **True** (non-zero / non-empty).
Returns **False** if any key is 0, False, or ''.

Syntax:

```
all(dictionary)
```

Example:

```
d1 = {1: 'A', 2: 'B', 3: 'C'}
d2 = {0: 'A', 2: 'B'}
print(all(d1))
print(all(d2))
```

Output:
```
True
False
```

3. any( ): Returns **True** if **any key** in the dictionary is **True** (non-zero / non-empty).
Returns **False** if **all keys** are 0, False, or empty.

Syntax:

```
any(dictionary)
```

Example:

```
d1 = {0: 'A', 1: 'B', 0: 'C'}
d2 = {0: 'A', 0: 'B'}
print(any(d1))
print(any(d2))
```

Output:
```
True
False
```

4. sorted( ): Returns a **sorted list of dictionary keys** in **ascending order**.

Syntax:

```
sorted(dictionary)
```

Example:

```
student = {'b': 20, 'a': 10, 'c': 30}
print(sorted(student))
```

Output:
```
['a', 'b', 'c']
```

Note: sorted( ) always returns a **new sorted list of keys**, no matter what type of iterable (string, tuple, etc.) give it.
If keys are **tuples**: like (2, 3) and (1, 4).
Tuples are compared like:

- First elements are compared → 2 vs 1
- Since 1 < 2, (1, 4) comes **before** (2, 3).

Dictionary Methods

1. keys( ) : Returns a view object containing all the **keys** of the dictionary.

Syntax:

```
dictionary.keys()
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print(student.keys())
```

Output:
```
dict_keys(['name', 'age', 'course'])
```

2. values( ) : Returns a view object containing all the **values** of the dictionary.

Syntax:

```
dictionary.values()
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print(student.values())
```

Output:
```
dict_values(['Raju', 22, 'BCAAI'])
```

3. items() : Returns a view object containing all **key–value pairs** as tuples.

Syntax:

```
dictionary.items()
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print(student.items())
```

Output:
```
dict_items([('name', 'Raju'), ('age', 22), ('course', 'BCAAI')])
```

4. get() : Returns the **value** of the specified key.
If the key is not found, returns the **default value** instead of an error.

Syntax:

```
dictionary.get(key, default_value)
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print(student.get('name'))
print(student.get('marks'))
```

Output:
```
Raju
None
```

Note 1: student.get('marks') → The key 'marks' does **not** exist in the dictionary.

- The get() method does **not raise an error** (unlike student['marks'] which would cause a KeyError).
- Instead, it returns **None** by default.

Note 2: **we can also provide a default value:**

```
print(student.get('marks', 0))
```

Output:
```
0
```

5. update( ): Updates the dictionary with **key–value pairs** from another dictionary.

Syntax:

```
dictionary.update(other_dictionary)
```

Example 1:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
student.update({'marks': 88, 'age': 23})
print(student)
```

Output:
```
{'name': 'Raju', 'age': 23, 'course': 'BCAAI', 'marks': 88}
```

Note: When you pass an existing key **(like 'age' or 'name')**, its value gets replaced with the new one.

Example 2:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
student.update({'age': 23, 'name': 'Ajay'})
print(student)
```

Output:

```
{'name': 'Ajay', 'age': 23, 'course': 'BCAAI'}
```

6. pop(): Removes the item with the specified key and returns its value.

Syntax:

```
dictionary.pop(key)
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
student.pop('age')
print(student)
```

Output:

```
{'name': 'Raju',
'course': 'BCAAI'}
```

7. popitem(): Removes and returns the **last inserted key–value pair** as a tuple.

Syntax:

```
dictionary.popitem()
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
student.popitem()
print(student)
```

Output:

```
{'name': 'Raju', 'age': 22}
```

8. clear(): Removes **all items** from the dictionary.

Syntax:

```
dictionary.clear()
```

Example:

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
student.clear()
print(student)
```

Output:

```
{}
```

9. setdefault() : Returns the **value of a key** if it exists;
if not, inserts the key with a **default value** and returns that value.

Syntax:

```
dictionary.setdefault(key, default_value)
```

Example:

```
student = {'name': 'Raju', 'age': 22}
student.setdefault('course', 'BCAAI')
print(student)
```

Output:

```
{'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
```

11. fromkeys(): Creates a **new dictionary** from a **sequence of keys** and assigns them a **common value**.
If the value is not given, all keys get the default value **None**.

Syntax:

```
dictionary = dict.fromkeys(sequence, value)
```

- **sequence** → list, tuple, or set of keys
- **value** → (optional) value assigned to each key

Example 1: With a Given Value

```
keys = ['name', 'age', 'course']
student = dict.fromkeys(keys, 'N/A')
print(student)
```

Output:
```
{'name': 'N/A', 'age': 'N/A', 'course': 'N/A'}
```

Example 2: Without a Given Value

```
keys = ['name', 'age', 'course']
student = dict.fromkeys(keys)
print(student)
```

Output:
```
{'name': None, 'age': None, 'course': None}
```

Populating and Traversing in Dictionary
1. Populating: Populating means adding key–value pairs to a dictionary — either while creating it or after creation.

Example 1: Creating (Static Population)

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
print(student)
```

Output:
```
{'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
```

Example 2: Dynamic Population

```
student = {}
student['name'] = 'Raju'
student['age'] = 22
student['course'] = 'BCAAI'
print(student)
```

Output:
```
{'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
```

2. Traversing a Dictionary: Traversing means **accessing each key and value** in a dictionary — usually with a for loop.

Example 1: Traversing Keys

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
for key in student:
    print(key)
```

Output:
```
name
age
course
```

Example 2: Traversing Values

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
for value in student.values():
    print(value)
```

Output:
```
Raju
22
BCAAI
```

Example 3: Traversing Key–Value Pairs

```
student = {'name': 'Raju', 'age': 22, 'course': 'BCAAI'}
for key, value in student.items():
    print(key, ":", value)
```

Output:

```
name : Raju
age : 22
course : BCAAI
```

Tuples

A **tuple** in Python is a **built-in sequence data type** that can hold an **ordered and immutable collection of items** in a single variable.

Features of Tuples

- **Ordered** – Maintains the order of items as they are added.
- **Immutable** – Elements cannot be changed, added, or removed after creation.
- **Index-based** – Items are accessed by their index, starting from index 0.
- **Allows Duplicates** – Same value can appear multiple times in a tuple.
- **Mixed Data Types** – Can store different types of values (integers, strings, floats, Booleans, lists, even other tuples).

Creating Tuples

Tuples are created using **parentheses ( )**, with items separated by **commas (,)** inside the tuple.

Syntax:

```
tuple_name = (item_1, item_2, item_3, ……, item_n)
```

Examples

1. Creating a Number Tuple

```
number = (1, 2, 3, 5, 6, 7, 8, 9, 10)
print(number)
print(type(number))
```

Output:

```
(1, 2, 3, 5, 6, 7, 8, 9, 10)
<class 'tuple'>
```

2. Different Data Types in a Single Tuple

```
mixed_tuple = (1, 2.8, 3+6j, True, "BCAAI")
print(mixed_tuple, type(mixed_tuple))
```

Output:

(1, 2.8, (3+6j), True, 'BCAAI') <class 'tuple'>

3. Creating a Nested Tuple

```
nested_tuple = (1, (2, 3), (4, 5))
print(nested_tuple)
print(type(nested_tuple))
```

Output:

(1, (2, 3), (4, 5))
<class 'tuple'>

Creating an Empty Tuple

A tuple that does not contain any items is called an **empty tuple**.

Syntax:

```
empty_tuple = ()
```

Example:

```
empty_tuple = ()
print(empty_tuple)
print(type(empty_tuple))
```

Output:

```
()
<class 'tuple'>
```

Note: Creating Single-Value Tuple

- A **single-value tuple** is a tuple that contains **only one element**.
- To create a single-value tuple, **a comma (,) must be added after the element**. Without the comma, Python will treat it as a normal value in parentheses.

Syntax:

```
single_tuple = (value,)
```

Example

```
single_int = (5,)
print(single_int)
print(type(single_int))
```

Output:

```
(5,)
<class 'tuple'>
```

Creating a Tuple using tuple()

In Python, tuples can also be created using the built-in function tuple().
It is commonly used for **type conversion**, i.e., converting other data types (like list, string, or set) into a tuple.

Syntax:

```
tuple_name = tuple(iterable)
```

- iterable → any sequence (list, string, set, or another tuple) that will be converted into a tuple.

Examples:

1. From a list (type conversion)

```
numbers = tuple([1, 2, 3, 4])
print(numbers)
```

Output:
`(1, 2, 3, 4)`

2. From a string (type conversion)

```
letters = tuple("BCAAI")
print(letters)
```

Output:
`('B', 'C', 'A', 'A', 'I')`

3. From a set (type conversion)

```
myset = {10, 20, 30}
converted = tuple(myset)
print(converted)
```

Output:
`(10, 20, 30)`

4. Creating an empty tuple

```
empty = tuple()
print(empty)
```

Output:
`( )`

Basic Tuple Operations in Python

Tuples support similar operations to lists, with the exception that they are **immutable** (cannot be changed).

1. Concatenation (+): Combines two or more tuples into a single tuple.

Syntax:

```
new_tuple = tuple1 + tuple2
```

Example:

```
a = (1, 2, 3)
b = (4, 5, 6)
c = a + b
print(c)
```

Output:
`(1, 2, 3, 4, 5, 6)`

2. Repetition (*): Repeats the elements of a tuple a specified number of times.

Syntax:

```
new_tuple = tuple1 * n
```

Example:

```
a = (1, 2)
print(a * 3)
```

Output:
`(1, 2, 1, 2, 1, 2)`

3. Membership (in / not in): Checks if an element exists or does not exist in a tuple.

Syntax:

```
item in tuple_name
item not in tuple_name
```

Example:

```
fruits = ("apple", "banana",
"cherry")
print("banana" in fruits)
print("mango" not in fruits)
```

Output:

```
True
True
```

4. Comparison (= =, ! =): Compares two tuples for equality (==) or inequality (!=).

Syntax:

```
tuple1 == tuple2
tuple1 != tuple2
```

Example:

```
a = (1, 2, 3)
b = (1, 2, 3)
c = (3, 2, 1)
print(a == b)
print(a != c)
```

Output:

```
True
True
```

Indexing and Slicing in Tuples

Indexing in Tuples

- **Indexing** is the process of accessing individual items of a tuple using their position number (called *index*).
- Indexing starts from **0** for the first element (positive indexing).
- **Negative indexing** starts from **-1** for the last element and moves backward.

Syntax:

```
tuple_name[index]
```

Example:

```
fruits = ("apple", "banana", "cherry", "mango")
print(fruits[0])
print(fruits[1])
print(fruits[2])
```

Output:

```
apple
banana
cherry
```

Positive and Negative Indexing in Tuples

1. Positive Indexing

- Starts from **0** for the first element.
- Increases by 1 for each next element.

Syntax:

```
tuple_name[index]
```

Example:

```
fruits = ("apple", "banana", "cherry", "mango")
print(fruits[0])
print(fruits[2])
```

Index Positions (Positive):

```
fruits = ("apple", "banana", "cherry", "mango")
           0         1          2          3
```

2. Negative Indexing

- Starts from **-1** for the last element.
- Decreases by 1 as we move backward.

Syntax:

```
tuple_name[-index]
```

Example:

```
fruits = ("apple", "banana", "cherry", "mango")
print(fruits[-1])
print(fruits[-3])
```

Index Positions (Negative):

```
fruits = ("apple", "banana", "cherry", "mango")
          -4         -3          -2          -1
```

Slicing in Tuples

Slicing is the process of extracting a portion (subsequence) of a tuple by specifying a range of index positions.

Syntax:

```
tuple_name[start : end : step]
```

- **start** → index where the slice begins (default = 0)
- **end** → index where the slice ends (excluded)
- **step** → interval/jump between elements (default = 1)

Example:

```
numbers = (10, 20, 30, 40, 50, 60, 70)

# 1. Simple slicing
print(numbers[1:4])
# Output: (20, 30, 40)

# 2. Omitting start (defaults to 0)
print(numbers[:3])
# Output: (10, 20, 30)

# 3. Omitting end (till last element)
print(numbers[3:])
# Output: (40, 50, 60, 70)

# 4. Using step
print(numbers[::2])
# Output: (10, 30, 50, 70)

# 5. Negative slicing (reverse tuple)
print(numbers[::-1])
# Output: (70, 60, 50, 40, 30, 20, 10)
```

Built-in Functions in Tuples

1. len(): This function returns the number of elements in a tuple.

Example:

```
fruits = ("apple", "banana", "cherry", "mango")
print(len(fruits))
```

Output:
4

2. min(): This function returns the smallest element in a tuple.

Example:

```
numbers = (5, 8, 2, 10, 3)
print(min(numbers))
```

Output:
2

3. max(): This function returns the largest element in a tuple.

Example:

```
numbers = (5, 8, 2, 10, 3)
print(max(numbers))
```

Output:
10

4. sum(): This function returns the sum of all numeric elements in a tuple.

Example:

```
numbers = (5, 10, 15, 20)
print(sum(numbers))
```

Output:
50

5. any(): This function returns **True** if at least one element in the tuple is **non-zero**. If all elements are **zero,** or **an empty tuple** it returns **False**.

Example:

```
values = (0, 0, 3, 0)
print(any(values))
```

Output:
```
True
```

6. all(): This function returns **True** if all elements in the tuple are **non-zero**. If any element is **zero**, or **an empty tuple**, it returns **False**

Example:

```
values1 = (1, 2, 3, 4)
values2 = (1, 0, 3, 4)
print(all(values1))
print(all(values2))
```

Output:
```
True
False
```

7. sorted(): This function returns a new tuple with the elements sorted in ascending order.

Example:

```
numbers = (40, 10, 30, 20)
print(numbers)
print(sorted(numbers))
print(sorted(numbers, reverse=True))
```

Output:
```
[40, 10, 30, 20]
[10, 20, 30, 40]
[40, 30, 20, 10]
```

Tuple Methods in Python

1. count(): Returns the number of times a specified value appears in the tuple.

Syntax:

```
tuple_name.count(value)
```

Example:

```
numbers = (10, 20, 30, 20, 40, 20)
print(numbers.count(20))
```

Output:
```
3
```

2. index(): Returns the index (position) of the first occurrence of the specified value in the tuple.
If the value is not found, it raises a **ValueError**.

Syntax:

```
tuple_name.index(value)
```

Example:

```
fruits = ('apple', 'banana', 'cherry', 'banana', 'mango')
print(fruits.index('banana'))
```

Output:
```
1
```

Tuple Packing and Unpacking

Tuple Packing

Tuple packing is the process of **combining multiple values into a single tuple variable**.
Python automatically packs the given values into a tuple.

Syntax:

```
tuple_name = value1, value2, value3….value_n
```

Example:

```
t = 31, 21, 45, 89.5
print(t)
```

Output:

```
(31, 21, 45, 89.5)
```

Tuple Unpacking

Tuple unpacking is the process of **extracting individual elements from a tuple and assigning them to separate variables**.

The number of variables must match the number of elements in the tuple.

Syntax:

```
var1, var2, var3, ...var_n = tuple_name
```

Example:

```
t = 31, 21, 40
marks1, marks2, marks3 = t
```

Populating and Traversing Tuples

1. Populating a Tuple

- Since **tuples are immutable**, you **cannot add or remove elements** after creation.
- Tuples are populated by directly assigning values when creating them.

Example:

```
fruits = ("apple", "banana", "cherry", "mango")
print(fruits)
```

Note:

- To "add" elements, you need to **create a new tuple** by concatenation.

```
fruits = fruits + ("orange", "grapes")
print(fruits)
```

2. Traversing a Tuple

- Traversing a Tuple means accessing or visiting each element of the tuple one by one in sequence.
- Purpose: To read, display, or perform operations on all elements.

Example:

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

Output:

```
apple
banana
cherry
```

Sets

A **set** in Python is a **built-in data type** that can hold an **unordered and mutable collection of unique items** in a single variable.

Characteristics of Sets

1. **Unordered:**
   The elements in a set do not maintain any specific order.
   When elements are stored in a set, their arrangement is arbitrary and may change each time the set is accessed or displayed.
2. **Unindexed:**
   Sets do not support indexing or slicing operations because they are unordered.
   Therefore, individual elements cannot be accessed using an index position like in lists or tuples.
3. **Mutable:**
   A set is a mutable, meaning its content can be changed after creation.
4. **Unique Elements:**
   Sets automatically eliminate duplicate values.
   Each element in a set must be unique; if duplicates are added, only one instance is retained.
5. **Mixed Data Types:** A set can store elements of different data types, such as integers, strings, floats, Booleans, or tuples.

Note: Integers, strings, floats, Booleans, and tuples are **immutable**. A set can only store **immutable elements**; mutable objects like lists and dictionaries **cannot** be stored in a set.

Creating Sets

Sets in Python are created using **curly braces { }** or the built-in function set(), with elements separated by **commas (,)** inside the set.

Syntax:

```
set_name = {item_1, item_2, item_3, ..., item_n}
```

Examples

1. Creating a Number Set

```
numbers = {1, 2, 3, 5, 6, 7, 8, 9, 10}
print(numbers)
print(type(numbers))
```

Output:

```
{1, 2, 3, 5, 6, 7, 8, 9, 10}
<class 'set'>
```

2. Different Data Types in a Single Set

```
mixed_set = {1, 2.8, True, "BCAAI", (3, 4)}
print(mixed_set)
print(type(mixed_set))
```

Output:

```
{1, 2.8, (3, 4), 'BCAAI'}
<class 'set'>
```

Note:

- Nested sets are **not allowed** to contain mutable elements like lists or dictionaries.
- Nesting is only possible with **immutable elements**, such as tuples.

Creating a Set using set()

Python provides the built-in function set() to create sets, especially from **iterable objects** like lists, strings, or tuples. This is commonly used for **type conversion**.

Syntax:

```
set_name = set(iterable)
```

- iterable → list, string, tuple, or another set

Examples:

1. From a List (type conversion)

```
numbers = set([1, 2, 3, 4])
print(numbers)
```

Output:

```
{1, 2, 3, 4}
```

2. From a String (type conversion)

```
letters = set("BCAAI")
print(letters)
```

Output:
```
{'C', 'B', 'I', 'A'}
```

3. From a Tuple (type conversion)

```
mytuple = (10, 20, 30)
converted = set(mytuple)
print(converted)
```

Output:
```
{10, 20, 30}
```

Creating an Empty Set

A set that does not contain any elements is called an empty set.

Syntax:

```
empty_set = set()
```

Example:

```
empty_set = set()
print(empty_set)
print(type(empty_set))
```

Output:
```
set()
<class 'set'>
```

Note: `{}` creates an empty **dictionary**, not a set.

Basic Set Operations in Python

1. Union (|): Combines all unique elements from two or more sets into a single set.

Syntax:
```
new_set = set_1 | set_2
```

Example:

```
A = {1, 2, 3}
B = {3, 4, 5}
C = A | B
print(C)
```

Output:
```
{1, 2, 3, 4, 5}
```

2. Intersection (&): Returns only the elements that are common to both sets.

Syntax:
```
new_set = set_1 & set_2
```

Example:

```
A = {1, 2, 3}
B = {3, 4, 5}
C = A & B
print(C)
```

Output:
```
{3}
```

3. Difference (-): Returns elements present in the first set but not in the second.

Syntax:
```
new_set = set1 - set2
```

Example:

```
A = {1, 2, 3}
B = {3, 4, 5}
C = A - B
print(C)
```

Output:
```
{1, 2}
```

4. Symmetric Difference (^): Returns elements present in either set, but not in both.

Syntax:

```
new_set = set1 ^ set2
```

Example:

```
A = {1, 2, 3}
B = {3, 4, 5}
C = A ^ B
print(C)
```

Output:
```
{1, 2, 4, 5}
```

5. Membership (in / not in): Checks whether an element exists or does not exist in a set.

Syntax:

```
item in set_name
item not in set_name
```

Example:

```
fruits = {"apple", "banana", "cherry"}
print("banana" in fruits)
print("mango" not in fruits)
```

Output:
True
True

6. Comparison (== / !=): Compares two sets for equality or inequality.

Syntax:

```
set1 == set2
set1 != set2
```

Example:

```
A = {1, 2, 3}
B = {1, 2, 3}
C = {3, 2, 1}
print(A == B)
print(A != C)
```

Output:
```
True
False
```

Built-in Functions on Sets

1. len( ): This function returns the total number of elements present in the set.

Example:

```
A = {1, 2, 3}
print(len(A))
```

Output:
```
3
```

2. max( ): This function returns the largest element in the set.

Example:

```
A = {1, 2, 3}
print(max(A))
```

Output:
```
3
```

3. min( ): This function returns the smallest element in the set.

Example:

```
A = {1, 2, 3}
print(min(A))
```

Output:
```
1
```

4. sum(): This function returns the sum of all numeric elements in the set.

Example:

```
A = {1, 2, 3}
print(sum(A))
```

Output:
```
6
```

5. sorted(): This function returns a sorted list of elements from the set in ascending order.

Example:
```
B = {3, 1, 2}
print(sorted(B))
```

Output:
```
[1, 2, 3]
```

Note: sorted ( ) give list of sorted elements.

6. any(): This function returns True if at least one element in the set evaluates to True; otherwise returns False.

Example:
```
C = {0, 0, 1}
print(any(C))
```

Output:
```
True
```

7. all(): this function returns **True** if all elements in the set evaluate to True; otherwise returns False.

Example:
```
D = {1, 2, 3}
print(all(D))
```

Output:
```
True
```

Set Methods in Python

Python provides several built-in methods to manipulate sets. Unlike functions, methods are called on the set object.

1. add( ) : This Method is used to add a single element to the set. If the element already exists, it does nothing.

Syntax:
```
set_name.add(element)
```

Example:
```
A = {1, 2, 3}
A.add(4)
print(A)
```

Output:
{1, 2, 3, 4}

2. update( ) : This method adds multiple elements from another iterable (list, set, tuple, etc.) to the set.

Syntax:
```
set_name.update(iterable)
```

Example:
```
A = {1, 2, 3}
A.update([4, 5, 6])
print(A)
```

Output:
```
{1, 2, 3, 4, 5, 6}
```

3. remove( ): This method removes the specified element from the set. Raises a **KeyError** if the element does not exist.

Syntax:
```
set_name.remove(element)
```

Example:
```
A = {1, 2, 3, 4}
A.remove(3)
print(A)
```

Output:
```
{1, 2, 4}
```

4. discard( ): This method removes the specified element from the set. Does **not** raise an error if the element is not present.

Syntax:
```
set_name.discard(element)
```

Example:

```
A = {1, 2, 3, 4}
A.discard(5)
print(A)
```

Output:
```
{1, 2, 3, 4}
```

5. pop( ): This method removes and returns an arbitrary element from the set. Raises KeyError if the set is empty.

Syntax:
```
set_name.pop( )
```

Example:
```
A = {1, 2, 3, 4}
elem = A.pop()
print(elem)
print(A)
```

Output:
```
4
{1, 2, 3, 4}
```

6. clear( ): This method removes all elements from the set, resulting in an empty set.

Syntax:
```
set_name.clear()
```

Example:
```
A = {1, 2, 3}
A.clear()
print(A)
```

Output:
```
set()
```

7. union( ): This method returns a new set containing all unique elements from the original set and another set.

Syntax:
```
set1.union(set2)
```

Example:
```
A = {1, 2, 3}
B = {3, 4, 5}
C = A.union(B)
print(C)
```

Output:
```
{1, 2, 3, 4, 5}
```

8. intersection( ) : This method returns a new set containing elements common to both sets.

Syntax:
```
set1.intersection(set2)
```

Example:
```
A = {1, 2, 3}
B = {2, 3, 4}
C = A.intersection(B)
print(C)
```

Output:
```
{2, 3}
```

9. difference( ): This method returns a new set containing elements present in the first set but not in the second.

Syntax:
```
set1.difference(set2)
```

Example:
```
A = {1, 2, 3}
B = {2, 3, 4}
C = A.difference(B)
print(C)
```

Output:
```
{1}
```

10. symmetric_difference(): This method returns a new set containing elements present in either of the sets but not in both.

Syntax:
```
set1.symmetric_difference(set2)
```

Example:
```
A = {1, 2, 3}
B = {2, 3, 4}
C = A.symmetric_difference(B)
print(C)
```

Output
```
{1, 4}
```

11. issubset(): This method returns True if all elements of the set are present in another set.

Syntax:
```
set1.issubset(set2)
```

Example:
```
A = {1, 2}
B = {1, 2, 3}
print(A.issubset(B))
```

Output:
```
True
```

12. issuperset( ) : This method returns True if the set contains all elements of another set.

Syntax:
```
set1.issuperset(set2)
```

Example:
```
A = {1, 2, 3}
B = {1, 2}
print(A.issuperset(B))
```

Output:
```
True
```

13. isdisjoint(): This method returns **True** if the set has no elements in common with another set.

Syntax:
```
set1.isdisjoint(set2)
```

Example:
```
A = {1, 2}
B = {3, 4}
print(A.isdisjoint(B))
```

Output:
```
True
```

Populating and Traversing a Set in Python

1. Populating a Set

- Populating a set means **adding elements** to the set.
- Since sets are unordered, **duplicates are automatically ignored**.
- Elements can be added **one by one** using the add() method.

Example:
```
numbers = set()
for i in range(1, 6):
    numbers.add(i)
print("Populated Set:", numbers)
```

Output:
```
Populated Set: {1, 2, 3, 4, 5}
```

2. Traversing a Set

- Traversing a set means **accessing each element** in the set one by one.
- Since sets are unordered, elements may not appear in the order they were added.

Example:

```
fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
```

Output:
apple
banana
cherry

Note: Sets are unordered collections, so the traversal order may differ each time the program runs.

File Handling in Python

File

A **file** in Python is a **placeholder or named location in computer memory** that is used to **store data permanently**.

- Unlike **variables**, which store data **temporarily** while a program is running, **files** store data **permanently** so that it can be accessed or used even after the program ends.

Files Types in Python

Python supports mainly **two types of files** based on how data is stored and accessed:

1. Text File

A **text file** is a file that stores data in **plain text form**, that is, as **a sequence of readable characters** (letters, digits, and symbols).
Each line in a text file end with a **newline character (\n)**.
These files can be easily opened and read using text editors such as **Notepad**, **VS Code**, or **Sublime Text**.

Characteristics of Text Files

1. Data is stored as a **sequence of characters**.
2. Lines are separated by the **newline (\n)** character.
3. Easily readable and editable by humans.
4. Best suited for storing **structured text data** like documents, program code, or tabular information.

Common Text Files

1. **Web Standards Files**
   - **Purpose:** Used to create and design **web pages**, including structure, styling, and client-side scripting.
   - **Example Extensions:** .html, .css, .js
   - **Example:** index.html, style.css, script.js

2. **Source Code Files**
   - **Purpose:** Store **program source code** written in programming languages for compiling or interpreting.
   - **Example Extensions:** .py, .java, .c, .cpp
   - **Example:** program.py, main.java, code.c

3. **Document Files**
   - **Purpose:** Store **readable information** like notes, reports, or instructions that can be accessed and edited by users.
   - **Example Extensions:** .txt, .md
   - **Example:** notes.txt, readme.md

4. **Tabular Data Files**
   - **Purpose:** Store **structured data** in rows and columns for easy data analysis, storage, and exchange between applications.
   - **Example Extensions:** .csv (comma-separated), .tsv (tab-separated)
   - **Example:** students.csv, sales_data.tsv

2. Binary File

A **binary file** is a file that stores data in **binary format (0s and 1s)** instead of human-readable text.
Unlike text files, **binary files are not readable** by humans directly and are interpreted by programs or special software.
Binary files are used to store **non-text data** such as **images, audio, video, executables, and other multimedia files**.

Characteristics of Binary Files

1. Data is stored as **binary digits (0s and 1s)**.
2. Cannot be opened and read directly using normal text editors.
3. Requires **binary modes** to read or write.
4. Can store **large and complex data** like images, videos, and compiled programs.
5. More **efficient** for storage and retrieval of non-text data than converting it to text.

Common Binary Files

1. **Image Files**
   - **Purpose:** Store image data.
   - **Example Extensions:** .jpg, .png, .gif, .bmp
   - **Example File:** photo.jpg
2. **Audio Files**
   - **Purpose:** Store sound and music data.
   - **Example Extensions:** .mp3, .wav, .aac
   - **Example File:** song.mp3
3. **Video Files**
   - **Purpose:** Store video data such as movies or clips.
   - **Example Extensions:** .mp4, .avi
   - **Example File:** movie.mp4
4. **Executable Files**
   - **Purpose:** Store programs or software that can be executed.
   - **Example Extensions:** .exe, .bin
   - **Example File:** setup.exe
5. **Other Data Files**
   - **Purpose:** Store structured, custom, or program-specific binary data.
   - **Example Extensions:** .bin, .dat
   - **Example File:** data.bin

File Handling

File handling refers to the process of performing operations on a file, such as **creating, opening, reading, writing and closing** it through a programming interface.
It involves managing the data flow between the **program** and the **file system** on the storage device, ensuring that data is handled safely and efficiently.

Access modes of files

Mode	Operation / Meaning	File Type	File Must Exist?	Description / Action
r	Read	Text	Yes	Opens file for reading. Error if file does not exist.
w	Write	Text	No	Opens file for writing. Creates new file or **overwrites** existing file.
a	Append	Text	No	Opens file to add data at the end. Creates new file if it doesn't exist.
x	Exclusive Create	Text	No	Creates a new file. Error if file already exists.
r+	Read & Write	Text	Yes	Opens file for both reading and writing. File must exist.
w+	Write & Read	Text	No	Opens file for reading and writing. Creates new or **overwrites** existing file.
a+	Append & Read	Text	No	Opens file for reading and appending. Creates new file if it doesn't exist.
rb	Read Binary	Binary	Yes	Opens binary file for reading.
wb	Write Binary	Binary	No	Opens binary file for writing. Creates new or **overwrites** existing file.
ab	Append Binary	Binary	No	Opens binary file to append data. Creates new file if it doesn't exist.
rb+	Read & Write Binary	Binary	Yes	Opens binary file for both reading and writing. File must exist.
wb+	Write & Read Binary	Binary	No	Opens binary file for reading and writing. Creates new or **overwrites** existing file.
ab+	Append & Read Binary	Binary	No	Opens binary file for reading and appending. Creates new file if it doesn't exist.

Operations on Files

Create/Open Operation
To create or open a file, we can use **open( )** function, which requires file-path and mode as arguments:

Syntax:

```
file_handler = open('file_name', 'mode')
```

- **file_handler**: **variable** that stores the reference to the opened file.
- **file_name:** name (or path) of the file to be opened.
- **mode:** mode in which you want to open the file (read, write, append, etc.).

Note: *If you don't specify the mode, Python uses* ***'r'*** *(read mode) by default.*

Modes for **creating files**:
- ***'w'*** → Write mode (creates a new file or overwrites existing file)
- ***'a'*** → Append mode (creates file if it doesn't exist, else adds data at the end)
- ***'x'*** → Exclusive creation (creates new file, error if file exists)

Example – Creating a File

```
file = open("newfile.txt", "w")
file.write("Hello, Python!")
file.close()
```

Read Operation
To read data from a file, we use **file methods like read(), readline(),** and **readlines().** The file must be opened in **read mode** ('r') or **read/write mode** ('r+').

Syntax

```
file_handler.read()
```
- **file_handler** → Variable storing the reference to the opened file.

Access modes
- ***'r'*** - Read mode (text)
- ***'r+'*** - Read and write mode
- ***'rb'*** - Read mode (binary)

Common reading methods:
- ***.read()*** – Reads the **entire file** or specified number of characters.
- ***.readline()*** – Reads **one line** at a time.
- ***.readlines()*** – Reads **all lines** and returns them as a **list**.

Example – Reading Entire File

```
file = open("newfile.txt", "r")
content = file.read()
print(content)
file.close()
```

Example – Reading One Line

```
file = open("newfile.txt", "r")
line = file.readline()
print(line)
file.close()
```

Example – Reading All Lines

```
file = open("newfile.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

Write Operation

To write data to a file, we use the **write()** or **writelines()** methods. The file must be opened in **write mode ('w')**, **write/read mode ('w+')**, or **append mode ('a')**.

Syntax

```
file_handler.write(string)
file_handler.writelines(list_of_strings)
```

- **file_handler** → Variable storing the reference to the opened file.
- **string** → Text to be written to the file.
- **list_of_strings** → List of multiple strings to be written.

Notes

- `'w'-` Write mode (creates new or overwrites existing file)
- `'w+'-` Write and read mode
- `'a'-` Append mode (adds data at the end)
- `'a+'-` Append and read mode
- `'wb'-` Write mode for binary files
- `'wb+'-` Write and read mode for Binary files
- `'ab'-` Append mode for binary files
- `'ab+'-` Append and read mode for binary files
-

Example – Writing to a File

```
file = open("newfile.txt", "w")
file.write("This is a Python file handling example.")
file.close()
```

Example – Appending to a File

```
file = open("newfile.txt", "a")
file.write("\nThis line is appended.")
file.close()
```

Close Operation

After performing file operations, it is **important to close the file**. This ensures that **all data is saved** and system **resources are freed**.

Syntax

```
file_handler.close()
```

- **file_handler** → Variable storing the reference to the opened file.

Notes

- Always close a file after reading, writing, or appending.
- Alternatively, use **with open()** to automatically close the file after the block.

Example – Closing a File

```
file = open("newfile.txt", "r")
content = file.read()
file.close()
```

Opening a File with `with` Statement

The **`with` statement** in Python is used to **open a file and ensure it is automatically closed** after the block of code is executed. This avoids the need to explicitly call `.close()`.

Syntax

```
with open('file_name', 'mode') as file_handler:
  # statements
```

- **file_name** - Name or path of the file.
- **mode** - Access mode (`'r'`, `'w'`, `'a'`, etc.).
- **file_handler** - Variable referencing the opened file.

Notes

- The file is **automatically closed** when the block ends.
- Supports all file operations: **read**, **write**, **append**, and **binary** modes.
- Safer than manually opening and closing files.

File Names and Paths

A **file name** identifies a file, and a **path** specifies the location of a file on the computer. Python uses these to locate files when performing file operations like **open(), read(), or write()**.

File Name Rules

- Can include letters, numbers, underscores (_), and dot (.).
- Cannot include special characters like \ /: * ? " < > | on Windows.
- Should end with a **file extension** (e.g., .txt, .jpg, .py) to indicate the file type.

File Paths

A **file path** describes the **directory location** of a file on the system.

1. Absolute Path

- Specifies the **complete location** of the file from the root directory.
- Works regardless of the current working directory.

Example:

```
file = open("C:/Users/Documents/newfile.txt", "r")
```

2. Relative Path

- Specifies the location of the file **relative to the current working directory**.
- Convenient for files in the same project or folder.

Example:

```
file = open("data/newfile.txt", "r")  # File inside 'data' folder
```

Format Operators in File Handling

Format operators in Python are used to **insert or format data** into strings before writing them to a file. They are useful when you want to **write structured data** in a readable format.

Common Format Operators

Operator	Description	Example
`%s`	String	"Name: %s" % name
`%d`	Integer	"Age: %d" % age
`%f`	Float	"Price: %f" % price

Note: % operator works with a **single value** or a **tuple of values**.

Writing Formatted Data to a File

```
name = "Amith"
age = 24
salary = 45000.0

file = open("info.txt", "w")
file.write("Name: %s\n" % name)
file.write("Age: %d\n" % age)
file.write("Salary: %f\n" % salary)
file.close()
```

Content of info.txt after writing:

```
Name: Amith
Age: 24
Salary: 45000.0
```