**PYTHON PROGRAMMING**

**UNIT-4**

**LISTS, TUPLES, DICTIONARIES**

**Introduction to Lists in Python**

Lists in Python are one of the most commonly used data structures. They are versatile, mutable, and allow the storage of multiple items in a single variable.

### **What is a List?**

A list in Python is an ordered collection of elements, which can be of different types. Lists are defined using square brackets [], and elements inside a list are separated by commas.

**Example:**

# Creating a list

my\_list = [1, 2, 3, 4, 5]

print(my\_list)



**Output:**

[1, 2, 3, 4, 5]



Lists can contain mixed data types:

mixed\_list = [1, "Hello", 3.14, True]

print(mixed\_list)



**Output:**

[1, 'Hello', 3.14, True]



### **Accessing List Elements**

Elements in a list can be accessed using their index. Python uses **zero-based indexing**, meaning the first element is at index 0.

**Example:**

my\_list = [10, 20, 30, 40, 50]

print(my\_list[0]) # Accessing first element

print(my\_list[2]) # Accessing third element

print(my\_list[-1]) # Accessing last element



**Output:**

10

30

50



### **Updating List Elements**

Lists in Python are **mutable**, meaning their elements can be modified.

**Example:**

my\_list = [10, 20, 30, 40, 50]

my\_list[1] = 25 # Updating second element

print(my\_list)



**Output:**

[10, 25, 30, 40, 50]



### **Deleting Elements from a List**

Elements can be removed from a list using del, remove(), or pop().

#### **Using del**

my\_list = [10, 20, 30, 40, 50]

del my\_list[2] # Deletes element at index 2

print(my\_list)



**Output:**

[10, 20, 40, 50]

#### **Using remove()**

my\_list = [10, 20, 30, 40, 50]

my\_list.remove(30) # Removes first occurrence of value 30

print(my\_list)



**Output:**

[10, 20, 40, 50]

#### **Using pop()**

my\_list = [10, 20, 30, 40, 50]

popped\_element = my\_list.pop(1) # Removes element at index 1

print(my\_list)

print("Popped Element:", popped\_element)



**Output:**

[10, 30, 40, 50]

Popped Element: 20



### **Iterating Over Lists**

There are multiple ways to iterate over a list.

#### **Using for loop**

my\_list = [10, 20, 30, 40, 50]

for item in my\_list:

print(item)



**Output:**

10

20

30

40

50

#### **Using while loop**

my\_list = [10, 20, 30, 40, 50]

i = 0

while i < len(my\_list):

print(my\_list[i])

i += 1



**Output:**

10

20

30

40

50



### **Finding List Length and Checking Membership**

#### **Finding Length using len()**

my\_list = [10, 20, 30, 40, 50]

print(len(my\_list)) # Returns the number of elements in the list



**Output:**

5

#### **Checking Membership using in and not in**

my\_list = [10, 20, 30, 40, 50]

print(30 in my\_list) # Returns True if 30 is in the list

print(100 not in my\_list) # Returns True if 100 is not in the list



**Output:**

True

True



**List Operations in Python**

## 1. Indexing, Slicing, and Matrices

### Indexing:

Indexing allows accessing individual elements in a list using their position.

**Example:**

my\_list = [10, 20, 30, 40, 50]

print(my\_list[0]) # Output: 10

print(my\_list[-1]) # Output: 50

### Slicing:

Slicing is used to extract a portion of a list.

**Example:**

my\_list = [10, 20, 30, 40, 50]

print(my\_list[1:4]) # Output: [20, 30, 40]

print(my\_list[:3]) # Output: [10, 20, 30]

print(my\_list[::2]) # Output: [10, 30, 50]

### Matrices:

Lists can be used to create matrices (2D lists).

**Example:**

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

print(matrix[1][2]) # Output: 6



## 2. List Methods

Common list methods include:

### append()

Adds an element to the end of the list.

my\_list = [1, 2, 3]

my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4]

### extend()

Extends a list by appending elements from another iterable.

my\_list = [1, 2, 3]

my\_list.extend([4, 5])

print(my\_list) # Output: [1, 2, 3, 4, 5]

### insert()

Inserts an element at a specific index.

my\_list = [1, 2, 4]

my\_list.insert(2, 3)

print(my\_list) # Output: [1, 2, 3, 4]

### remove()

Removes the first occurrence of a specific element.

my\_list = [1, 2, 3, 2]

my\_list.remove(2)

print(my\_list) # Output: [1, 3, 2]

### pop()

Removes and returns the element at a specified index.

my\_list = [1, 2, 3]

print(my\_list.pop(1)) # Output: 2

print(my\_list) # Output: [1, 3]

### sort()

Sorts the list in ascending order.

my\_list = [3, 1, 2]

my\_list.sort()

print(my\_list) # Output: [1, 2, 3]



## 3. Built-in Functions with Lists

Python provides built-in functions that work with lists.

### len()

Returns the length of the list.

my\_list = [10, 20, 30]

print(len(my\_list)) # Output: 3

### max()

Returns the maximum element.

print(max([10, 20, 30])) # Output: 30

### min()

Returns the minimum element.

print(min([10, 20, 30])) # Output: 10

### sum()

Returns the sum of elements.

print(sum([10, 20, 30])) # Output: 60

### sorted()

Returns a sorted version of the list without modifying the original.

my\_list = [3, 1, 2]

print(sorted(my\_list)) # Output: [1, 2, 3]



## 4. Nested Lists

A list can contain other lists as elements.

**Example:**

nested\_list = [[1, 2], [3, 4], [5, 6]]

print(nested\_list[1]) # Output: [3, 4]

print(nested\_list[1][0]) # Output: 3



Iterating through nested lists:

for sublist in nested\_list:

for item in sublist:

print(item, end=' ')

# Output: 1 2 3 4 5 6



## 5. List Comprehension

List comprehension provides a concise way to create lists.

### Basic List Comprehension

squares = [x\*\*2 for x in range(5)]

print(squares) # Output: [0, 1, 4, 9, 16]

### List Comprehension with Conditionals

even\_numbers = [x for x in range(10) if x % 2 == 0]

print(even\_numbers) # Output: [0, 2, 4, 6, 8]

### Nested List Comprehension

matrix = [[x for x in range(3)] for y in range(3)]

print(matrix)

# Output: [[0, 1, 2], [0, 1, 2], [0, 1, 2]]



**Introduction to Tuples in Python**

### What are Tuples?

A **tuple** is an ordered, immutable collection of elements in Python. Unlike lists, tuples cannot be modified after creation, making them useful for storing fixed data.

**Key Characteristics of Tuples:**

* Ordered: Elements maintain their position.
* Immutable: Cannot be changed after creation.
* Allows Duplicates: Elements can be repeated.
* Can store multiple data types: Integers, strings, lists, etc.

**Example:**

my\_tuple = (1, 2, 3, 4, 5)

print(my\_tuple) # Output: (1, 2, 3, 4, 5)



### Creating Tuples

Tuples can be created using parentheses () or without any brackets.

#### Creating a Tuple with Multiple Elements:

tuple1 = (10, 20, 30)

print(tuple1) # Output: (10, 20, 30)

#### Creating a Tuple Without Parentheses (Tuple Packing):

tuple2 = "apple", "banana", "cherry"

print(tuple2) # Output: ('apple', 'banana', 'cherry')

#### Creating an Empty Tuple:

empty\_tuple = ()

print(empty\_tuple) # Output: ()

#### Creating a Tuple with a Single Element (Comma Required):

single\_element\_tuple = (5,)

print(single\_element\_tuple) # Output: (5,)



### Accessing Tuple Elements

You can access elements of a tuple using indexing and slicing.

#### Accessing Elements by Index:

tuple3 = ('a', 'b', 'c', 'd')

print(tuple3[1]) # Output: 'b'

#### Accessing Elements Using Negative Indexing:

print(tuple3[-1]) # Output: 'd'

#### Slicing a Tuple:

print(tuple3[1:3]) # Output: ('b', 'c')



### Tuple Packing and Unpacking

#### Packing a Tuple:

Tuple packing means assigning multiple values into a single tuple.

packed\_tuple = 1, 2, 3, "Hello"

print(packed\_tuple) # Output: (1, 2, 3, 'Hello')

#### Unpacking a Tuple:

Unpacking means extracting values from a tuple into separate variables.

(a, b, c, d) = packed\_tuple

print(a) # Output: 1

print(d) # Output: Hello



Using \* for Unpacking:

numbers = (1, 2, 3, 4, 5)

a, \*b, c = numbers

print(a) # Output: 1

print(b) # Output: [2, 3, 4]

print(c) # Output: 5



### Loop through Tuples

You can iterate over a tuple using a for loop.

#### Using a for Loop:

tuple4 = ("apple", "banana", "cherry")

for item in tuple4:

print(item)



**Output:**

apple

banana

cherry

#### Using enumerate():

tuple5 = ('Python', 'Java', 'C++')

for index, value in enumerate(tuple5):

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



**Output:**

Index 0: Python

Index 1: Java

Index 2: C++



**Tuples Operations in Python**

## 1. Concatenation and Repetition

### **Concatenation of Tuples**

Concatenation is the process of joining two or more tuples to create a new tuple. The + operator is used for concatenation.

#### **Example:**

# Concatenating two tuples

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

result = tuple1 + tuple2

print("Concatenated Tuple:", result)

#### **Output:**

Concatenated Tuple: (1, 2, 3, 4, 5, 6)

### **Repetition of Tuples**

Repetition allows us to repeat elements of a tuple multiple times using the \* operator.

#### **Example:**

# Repeating a tuple

tuple1 = ("Hello",) \* 3

print("Repeated Tuple:", tuple1)

#### **Output:**

Repeated Tuple: ('Hello', 'Hello', 'Hello')



## 2. Membership and Iteration

### **Membership Operators (in and not in)**

Membership operators are used to check if an element is present in a tuple.

#### **Example:**

tuple1 = (10, 20, 30, 40)

print(20 in tuple1) # True

print(50 not in tuple1) # True

#### **Output:**

True

True

### **Iteration Through a Tuple**

We can iterate through a tuple using a for loop.

#### **Example:**

tuple1 = ("apple", "banana", "cherry")

for item in tuple1:

print(item)

#### **Output:**

apple

banana

cherry



## 3. Tuple Methods

Tuples have two built-in methods: count() and index().

### **1. count() Method**

The count() method returns the number of times a specified element appears in a tuple.

#### **Example:**

tuple1 = (1, 2, 3, 1, 1, 4, 5)

print("Count of 1:", tuple1.count(1))

#### **Output:**

Count of 1: 3

### **2. index() Method**

The index() method returns the index of the first occurrence of a specified element.

#### **Example:**

tuple1 = (10, 20, 30, 40, 50)

print("Index of 30:", tuple1.index(30))

#### **Output:**

Index of 30: 2



**Introduction to Sets in Python**

A **set** is a built-in data type in Python that is used to store multiple unique elements in an unordered manner. It is commonly used when you need to store distinct values and perform set operations like union, intersection, and difference.

### 1. Define Set

A **set** is a collection that is **unordered, unindexed, and does not allow duplicate values**.

#### Example:

my\_set = {1, 2, 3, 4, 5}

print(my\_set)

#### Output:

{1, 2, 3, 4, 5}



### 2. Create Set

Sets can be created using curly braces {} or by using the set() function.

#### Example 1: Using Curly Braces

fruits = {"apple", "banana", "cherry"}

print(fruits)

#### Example 2: Using set() Function

numbers = set([1, 2, 3, 4, 5])

print(numbers)

#### Output:

{1, 2, 3, 4, 5}



### 3. Using set() Function

The set() function can be used to convert other iterable types into a set.

#### Example:

string\_set = set("hello")

print(string\_set)

#### Output:

{'o', 'l', 'h', 'e'}



(Note: The output order may vary because sets are unordered.)

### 4. Characteristics of a Set

* **Unordered**: The elements have no defined order.
* **Unique Elements**: Duplicates are not allowed.
* **Mutable**: You can add or remove elements.
* **Heterogeneous**: It can contain different data types.

#### Example:

mixed\_set = {1, "Hello", 3.14, True}

print(mixed\_set)

#### Output:

{1, 3.14, 'Hello'}



### 5. Access Set Items

Since sets are unordered, items **cannot** be accessed using an index. However, we can loop through the set.

#### Example:

fruits = {"apple", "banana", "cherry"}

for fruit in fruits:

print(fruit)

#### Output:

banana

apple

cherry



(Note: Order may vary.)

### 6. Adding and Removing Elements

#### Adding Elements

Use the add() method to add a single element and update() to add multiple elements.

my\_set = {1, 2, 3}

my\_set.add(4)

print(my\_set)

my\_set.update([5, 6, 7])

print(my\_set)

#### Output:

{1, 2, 3, 4}

{1, 2, 3, 4, 5, 6, 7}

#### Removing Elements

Use remove() (raises an error if the element is not found) or discard() (does not raise an error if the element is missing).

my\_set.remove(2) # Removes 2 from the set

my\_set.discard(10) # Does nothing as 10 is not present

print(my\_set)

#### Output:

{1, 3, 4, 5, 6, 7}



Use pop() to remove a random element:

removed\_item = my\_set.pop()

print("Removed:", removed\_item)

print(my\_set)



### 7. Looping Through Sets

We can use a for loop to iterate through a set.

#### Example:

colors = {"red", "green", "blue"}

for color in colors:

print(color)

#### Output:

red

green

blue



**Sets Operations**  **in Python**

## 1. Mathematical Operations in Sets

Sets in Python support various mathematical operations such as union, intersection, difference, and symmetric difference.

### Union (| or union())

Combines all elements from two sets without duplication.

A = {1, 2, 3, 4}

B = {3, 4, 5, 6}

C = A | B # Using '|' operator

D = A.union(B) # Using union() method

print(C) # Output: {1, 2, 3, 4, 5, 6}

print(D) # Output: {1, 2, 3, 4, 5, 6}

### Intersection (& or intersection())

Finds common elements between two sets.

A = {1, 2, 3, 4}

B = {3, 4, 5, 6}

C = A & B # Using '&' operator

D = A.intersection(B) # Using intersection() method

print(C) # Output: {3, 4}

print(D) # Output: {3, 4}

### Difference (- or difference())

Finds elements present in one set but not in the other.

A = {1, 2, 3, 4}

B = {3, 4, 5, 6}

C = A - B # Using '-' operator

D = A.difference(B) # Using difference() method

print(C) # Output: {1, 2}

print(D) # Output: {1, 2}

### Symmetric Difference (^ or symmetric\_difference())

Finds elements in either set but not in both.

A = {1, 2, 3, 4}

B = {3, 4, 5, 6}

C = A ^ B # Using '^' operator

D = A.symmetric\_difference(B) # Using symmetric\_difference() method

print(C) # Output: {1, 2, 5, 6}

print(D) # Output: {1, 2, 5, 6}



## 2. Checking Membership

Membership checks are done using in and not in.

A = {1, 2, 3, 4}

print(2 in A) # Output: True

print(5 not in A) # Output: True



## 3. Set Comprehension

Python allows set comprehension similar to list comprehension.

squared\_numbers = {x\*\*2 for x in range(1, 6)}

print(squared\_numbers) # Output: {1, 4, 9, 16, 25}



## 4. Nested Sets

Sets do not allow mutable elements, so nested sets cannot be directly used. However, sets of frozensets can be used.

A = {frozenset({1, 2}), frozenset({3, 4})}

print(A) # Output: {frozenset({1, 2}), frozenset({3, 4})}



## 5. Frozen Sets

Frozen sets are immutable sets.

A = frozenset([1, 2, 3, 4])

print(A) # Output: frozenset({1, 2, 3, 4})

print(2 in A) # Output: True



Frozen sets do not support modification operations like add() or remove(). Attempting to do so will raise an error.

A.add(5) # AttributeError: 'frozenset' object has no attribute 'add'



**Introduction to Dictionaries in Python**

A dictionary in Python is an unordered, mutable, and indexed collection of key-value pairs. It is one of the most commonly used data structures for storing and manipulating data efficiently.

### **1. Overview of Dictionary**

A dictionary in Python is a collection that is used to store data values in key-value pairs. Unlike lists, which store elements in an ordered sequence, dictionaries store data in an unordered manner, making it easier to access elements using unique keys.

#### **Example:**

# Creating a dictionary

student = {

"name": "John",

"age": 20,

"course": "Computer Science"

}

print(student)

#### **Output:**

{'name': 'John', 'age': 20, 'course': 'Computer Science'}



### **2. What are Keys and Values?**

A dictionary consists of keys and corresponding values. The keys must be unique and immutable (strings, numbers, or tuples), while the values can be of any data type.

#### **Example:**

demo\_dict = {

"brand": "Toyota",

"model": "Corolla",

"year": 2020

}



In the above dictionary:

* Keys: "brand", "model", "year"
* Values: "Toyota", "Corolla", 2020

### **3. Accessing Items in a Dictionary**

We can access values in a dictionary using their corresponding keys.

#### **Example:**

student = {

"name": "Alice",

"age": 22,

"major": "Mathematics"

}

print(student["name"]) # Accessing value using key

print(student.get("age")) # Using get() method

#### **Output:**

Alice

22



### **4. Changing Items in a Dictionary**

Dictionary values can be changed using their keys.

#### **Example:**

student = {

"name": "Bob",

"age": 21,

"major": "Physics"

}

student["age"] = 22 # Changing the age

student.update({"major": "Computer Science"}) # Using update() method

print(student)

#### **Output:**

{'name': 'Bob', 'age': 22, 'major': 'Computer Science'}



### **5. Adding Items in a Dictionary**

We can add new key-value pairs to a dictionary.

#### **Example:**

student = {

"name": "Emma",

"age": 20

}

student["course"] = "Data Science" # Adding a new key-value pair

student.update({"grade": "A"}) # Using update() method

print(student)

#### **Output:**

{'name': 'Emma', 'age': 20, 'course': 'Data Science', 'grade': 'A'}



**Dictionaries Methods in Python**

## Removing Items in a Dictionary

To remove items from a dictionary, Python provides several methods:

### 1. Using pop()

Removes a specified key and returns its value.

my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

age = my\_dict.pop('age')

print(my\_dict) # {'name': 'Alice', 'city': 'New York'}

print(age) # 25

### 2. Using popitem()

Removes and returns the last inserted key-value pair.

my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

item = my\_dict.popitem()

print(my\_dict) # {'name': 'Alice', 'age': 25'}

print(item) # ('city', 'New York')

### 3. Using del

Deletes a key-value pair or the entire dictionary.

my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

del my\_dict['age']

print(my\_dict) # {'name': 'Alice', 'city': 'New York'}

### 4. Using clear()

Removes all elements from the dictionary.

my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

my\_dict.clear()

print(my\_dict) # {}



## Loop through Dictionary

### 1. Iterating Through Keys

my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

for key in my\_dict:

print(key)



**Output:**

name

age

city

### 2. Iterating Through Values

for value in my\_dict.values():

print(value)

### 3. Iterating Through Key-Value Pairs

for key, value in my\_dict.items():

print(key, "->", value)



**Output:**

name -> Alice

age -> 25

city -> New York



## Copying Dictionaries

### 1. Using copy()

original = {'a': 1, 'b': 2, 'c': 3}

copy\_dict = original.copy()

print(copy\_dict) # {'a': 1, 'b': 2, 'c': 3}

### 2. Using dict() Constructor

original = {'a': 1, 'b': 2, 'c': 3}

copy\_dict = dict(original)

print(copy\_dict) # {'a': 1, 'b': 2, 'c': 3}



## Dictionary Methods

### 1. keys() - Returns all keys

my\_dict = {'name': 'Alice', 'age': 25}

print(my\_dict.keys()) # dict\_keys(['name', 'age'])

### 2. values() - Returns all values

print(my\_dict.values()) # dict\_values(['Alice', 25])

### 3. items() - Returns key-value pairs

print(my\_dict.items()) # dict\_items([('name', 'Alice'), ('age', 25)])

### 4. get() - Retrieves value safely

print(my\_dict.get('age')) # 25

print(my\_dict.get('salary', 'Not Found')) # Not Found

### 5. update() - Merges dictionaries

d1 = {'a': 1, 'b': 2}

d2 = {'b': 3, 'c': 4}

d1.update(d2)

print(d1) # {'a': 1, 'b': 3, 'c': 4}



## Additional Methods

### 1. setdefault() - Returns a value and inserts a key if not found

my\_dict = {'name': 'Alice'}

print(my\_dict.setdefault('age', 25)) # 25

print(my\_dict) # {'name': 'Alice', 'age': 25}

### 2. fromkeys() - Creates dictionary from keys

keys = ['a', 'b', 'c']

values = 0

dict\_from\_keys = dict.fromkeys(keys, values)

print(dict\_from\_keys) # {'a': 0, 'b': 0, 'c': 0}



## Nested Dictionaries

A nested dictionary contains dictionaries inside a dictionary.

students = {

'student1': {'name': 'Alice', 'age': 25},

'student2': {'name': 'Bob', 'age': 23}

}

### Accessing Nested Dictionary Elements

print(students['student1']['name']) # Alice

### Modifying Nested Dictionary

students['student1']['age'] = 26

print(students)

### Adding New Dictionary Inside a Dictionary

students['student3'] = {'name': 'Charlie', 'age': 22}

print(students)

### Looping Through Nested Dictionary

for key, value in students.items():

print(key, value)



**Output:**

student1 {'name': 'Alice', 'age': 26}

student2 {'name': 'Bob', 'age': 23}

student3 {'name': 'Charlie', 'age': 22}



**END OF UNIT - 4**