Theory Exercise

Accessing List

Q1. Understanding how to create and access elements in a list.

A. A **list** is a **collection of ordered, changeable (mutable)** items.
Lists can hold **different data types** — like numbers, strings, or even other lists.

Example:

```
fruits = ['apple', 'banana', 'mango']
```

🧩 1️⃣ **Creating a List**

You can create a list using **square brackets []** and separating items with commas.

```
fruits = ['apple', 'banana', 'mango']

numbers = [10, 20, 30, 40, 50]

mixed = ['apple', 10, True, 3.14]
```

Output:

```
['apple', 'banana', 'mango']

[10, 20, 30, 40, 50]

['apple', 10, True, 3.14]
```

🧩 2️⃣ **Accessing Elements in a List (Indexing)**

Each element in a list has an **index number**, starting from **0**.

```
fruits = ['apple', 'banana', 'mango']


print(fruits[0])   # First element

print(fruits[1])   # Second element

print(fruits[2])   # Third element
```

Output:

```
apple

banana

mango
```

🧩 3️⃣ Accessing Elements Using Negative Index

Negative indices access elements **from the end** of the list.

```
fruits = ['apple', 'banana', 'mango']


print(fruits[-1])  # Last element
print(fruits[-2])  # Second last element
```

Output:

```
mango
banana
```

🧩 4️⃣ Accessing a Range of Elements (Slicing)

You can use **slicing** to get a subset of the list.

```
numbers = [10, 20, 30, 40, 50]


print(numbers[1:4])   # Elements from index 1 to 3
print(numbers[:3])    # First 3 elements
print(numbers[2:])    # From index 2 to the end
```

Output:

```
[20, 30, 40]
[10, 20, 30]
[30, 40, 50]
```

🧩 5️⃣ Accessing Elements Using a Loop

You can access all elements using a **for loop**.

```
fruits = ['apple', 'banana', 'mango']


for fruit in fruits:
   print(fruit)
```

Output:

```
apple
```

banana

mango

🧩 6️⃣ **Changing Elements in a List**

Lists are **mutable**, meaning you can modify elements.

```
fruits = ['apple', 'banana', 'mango']


fruits[1] = 'orange'  # Change 'banana' to 'orange'
print(fruits)
```

Output:

```
['apple', 'orange', 'mango']
```

Q2. History and evolution of Python.

A.

🔹 1️⃣ **Positive Indexing**

- **Positive indexing** starts from **0** for the **first element**, **1** for the second, and so on.
- Use it to **access elements from the beginning** of the list.

Example:

```
fruits = ['apple', 'banana', 'mango', 'orange']


print(fruits[0])  # First element
print(fruits[1])  # Second element
print(fruits[2])  # Third element
```

Output:

```
apple
banana
mango
```

✅ **Tip:** Index len(fruits)-1 gives the **last element** using positive indexing.

🔹 2️⃣ **Negative Indexing**

- **Negative indexing** starts from **-1** for the **last element**, -2 for the second last, and so on.
- Use it to **access elements from the end** of the list.

Example:

```
fruits = ['apple', 'banana', 'mango', 'orange']


print(fruits[-1])  # Last element
print(fruits[-2])  # Second last element
print(fruits[-3])  # Third last element
```

Output:

```
orange
mango
banana
```

✅ **Tip:** Negative indexing is very useful when you want elements **from the end** without calculating len(list).

🔹 3️⃣ Combining Positive and Negative Indexing

You can mix both with **slicing**:

```
numbers = [10, 20, 30, 40, 50]


print(numbers[1:4])    # Positive indexing (index 1 to 3)
print(numbers[-4:-1])  # Negative indexing (second element to second last)
```

Output:

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

🔹 4️⃣ Quick Reference Table

Index	Element	Description
0	First element	Positive indexing

Index	Element	Description
1	Second element	Positive indexing
-1	Last element	Negative indexing
-2	Second last element	Negative indexing

🔹 5️⃣ **Looping with Indexing**

You can also use **index numbers in loops**:

```
fruits = ['apple', 'banana', 'mango', 'orange']


for i in range(len(fruits)):
   print(f"Index {i}: {fruits[i]}")
```

Output:

```
Index 0: apple
Index 1: banana
Index 2: mango
Index 3: orange
```

Q3. Slicing a list: accessing a range of elements.

A. Slicing means taking out **a portion (sub-list)** from a list using index positions.

👉 General syntax:

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

Part	Meaning
start	Index where slice begins (included)
end	Index where slice ends (excluded)
step	(Optional) How many elements to skip

🧩 **Example 1: Basic Slicing**

```
numbers = [10, 20, 30, 40, 50, 60]
```

```
print(numbers[1:4])   # elements from index 1 to 3
```

Output:

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

✅ Index 1 = 20 → start included
✅ Index 4 = 50 → end excluded

🧩 **Example 2: Omitting Start or End**

```
numbers = [10, 20, 30, 40, 50, 60]

print(numbers[:3])   # from start to index 2
print(numbers[3:])   # from index 3 to end
```

Output:

```
[10, 20, 30]
[40, 50, 60]
```

✅ If **start** is missing → slicing begins from index 0
✅ If **end** is missing → slicing continues till the end

🧩 **Example 3: Using Negative Index**

```
numbers = [10, 20, 30, 40, 50, 60]

print(numbers[-4:-1])  # from 3rd element to 2nd last
```

Output:

```
[30, 40, 50]
```

✅ Negative indices count from the **end** of the list.

🧩 **Example 4: Using Step Value**

```
numbers = [10, 20, 30, 40, 50, 60]

print(numbers[::2])   # every 2nd element
print(numbers[1::2])  # every 2nd element starting from index 1
```

Output:

[10, 30, 50]

[20, 40, 60]

✅ Step value = how many elements to skip each time.

🧩 **Example 5: Reversing a List with Slicing**

```
numbers = [10, 20, 30, 40, 50, 60]


print(numbers[::-1])  # reverse order
```

Output:

[60, 50, 40, 30, 20, 10]

✅ Step = -1 means move backward through the list.

🧩 **Example 6: Slicing with Variables**

```
start = 1
end = 5
numbers = [10, 20, 30, 40, 50, 60]


print(numbers[start:end])
```

Output:

[20, 30, 40, 50]

List Operations

Q4. Common list operations: concatenation, repetition, membership.

A. 🔹 1️⃣ Concatenation (+)

- Concatenation combines **two or more lists** into a single list.
- Use the + operator.

Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
```

```
combined = list1 + list2
print("Concatenated List:", combined)
```

Output:

```
Concatenated List: [1, 2, 3, 4, 5, 6]
```

✅ Works for lists of any data type:

```
list1 = ['a', 'b']
list2 = ['c', 'd']
print(list1 + list2)  # ['a', 'b', 'c', 'd']
```

🔹 2️⃣ **Repetition (*)**

- Repetition **duplicates the list** a number of times.
- Use the * operator.

Example:

```
numbers = [1, 2, 3]

repeated = numbers * 3
print("Repeated List:", repeated)
```

Output:

```
Repeated List: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

✅ Useful when you need **default values** or repeating patterns.

🔹 3️⃣ **Membership (in / not in)**

- Membership operators **check if an element exists** in a list.
- in → True if element is present
- not in → True if element is **absent**

Example:

```
fruits = ['apple', 'banana', 'mango']

print('banana' in fruits)   # True
print('orange' in fruits)   # False
print('orange' not in fruits)  # True
```

Output:

True

False

True

✅ Can be used in **if statements**:

```
if 'apple' in fruits:
   print("Apple is available!")
```

Output:

Apple is available!

Q5. Understanding list methods like append(), insert(), remove(), pop().

A. 🔹 1️⃣ **append()**

- **Purpose:** Adds an element **at the end** of the list.

Example:

```
fruits = ['apple', 'banana']
fruits.append('mango')
print(fruits)
```

Output:

['apple', 'banana', 'mango']

✅ **Tip:** Only adds **one element at a time**.

🔹 2️⃣ **insert()**

- **Purpose:** Inserts an element at a **specific position** in the list.
- Syntax: list.insert(index, element)

Example:

```
fruits = ['apple', 'banana']
fruits.insert(1, 'orange')  # Insert 'orange' at index 1
print(fruits)
```

Output:

['apple', 'orange', 'banana']

✅ **Tip:** Elements from that index are **shifted to the right**.

🔹 3️⃣ **remove()**

- **Purpose:** Removes the **first occurrence** of a specified element.
- Syntax: list.remove(element)

Example:

```
fruits = ['apple', 'banana', 'mango', 'banana']
fruits.remove('banana')
print(fruits)
```

Output:

```
['apple', 'mango', 'banana']
```

✅ **Tip:** If the element is not found, Python raises a **ValueError**.

🔹 4️⃣ **pop()**

- **Purpose:** Removes an element at a **specific index** (default is **last element**) and **returns it**.
- Syntax: list.pop(index)

Example:

```
fruits = ['apple', 'banana', 'mango']
last_fruit = fruits.pop()  # Removes last element
print("Removed:", last_fruit)
print("Updated list:", fruits)

first_fruit = fruits.pop(0)  # Removes element at index 0
print("Removed:", first_fruit)
print("Updated list:", fruits)
```

Output:

```
Removed: mango
Updated list: ['apple', 'banana']
Removed: apple
Updated list: ['banana']
```

Working with Lists

Q6. Iterating over a list using loops.

A. 🔹 1️⃣ **Using a for loop (most common)**

The for loop lets you access **each element one by one**.

```
fruits = ['apple', 'banana', 'mango']


for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
mango
```

✅ Simple and clean. Use this when you **don't need the index**.

🔹 2️⃣ **Using a for loop with range() (access by index)**

Sometimes you need the **index of elements** while looping.

```
fruits = ['apple', 'banana', 'mango']


for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Output:

```
Index 0: apple
Index 1: banana
Index 2: mango
```

✅ Useful when you want to **modify elements** during iteration.

🔹 3️⃣ **Using a while loop**

A while loop can also be used to iterate over a list **using an index**.

```
fruits = ['apple', 'banana', 'mango']
i = 0
```

```
while i < len(fruits):
    print(fruits[i])
    i += 1
```

Output:

```
apple
banana
mango
```

✅ Gives more control over the loop but slightly longer than a for loop.

🔹 4️⃣ Iterating with enumerate()

enumerate() gives you **both index and element** while looping.

```
fruits = ['apple', 'banana', 'mango']

for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

Output:

```
Index 0: apple
Index 1: banana
Index 2: mango
```

✅ Cleaner than using range(len(list)).

🔹 5️⃣ Iterating in Reverse

You can loop over a list **backwards** using slicing or reversed().

Using reversed():

```
fruits = ['apple', 'banana', 'mango']

for fruit in reversed(fruits):
    print(fruit)
```

Output:

```
mango
```

banana

apple

Using slicing:

```
for fruit in fruits[::-1]:
    print(fruit)
```

Output:

mango

banana

apple

Q7. Sorting and reversing a list using sort(), sorted(), and reverse().

A. 1️⃣ Using sort()

- **Purpose:** Sorts the **list in-place** (modifies the original list).
- Syntax: list.sort()
- Optional parameter: reverse=True for descending order.

Example:

```
numbers = [50, 20, 10, 40, 30]

numbers.sort()  # Ascending order
print("Ascending:", numbers)

numbers.sort(reverse=True)  # Descending order
print("Descending:", numbers)
```

Output:

Ascending: [10, 20, 30, 40, 50]

Descending: [50, 40, 30, 20, 10]

✅ **Tip:** sort() changes the **original list** permanently.

🔹 **2️⃣ Using sorted()**

- **Purpose:** Returns a **new sorted list** without changing the original list.
- Syntax: sorted(list)

- Optional: sorted(list, reverse=True) for descending order.

Example:

```
numbers = [50, 20, 10, 40, 30]


sorted_numbers = sorted(numbers)  # Ascending
print("Original list:", numbers)
print("Sorted list:", sorted_numbers)


sorted_numbers_desc = sorted(numbers, reverse=True)  # Descending
print("Sorted Descending:", sorted_numbers_desc)
```

Output:

```
Original list: [50, 20, 10, 40, 30]
Sorted list: [10, 20, 30, 40, 50]
Sorted Descending: [50, 40, 30, 20, 10]
```

✅ **Tip:** Use sorted() when you **don't want to modify** the original list.

🔹 3️⃣ **Using reverse()**

- **Purpose:** Reverses the **elements of the list in-place** (does not sort).
- Syntax: list.reverse()

Example:

```
numbers = [10, 20, 30, 40, 50]


numbers.reverse()
print("Reversed List:", numbers)
```

Output:

```
Reversed List: [50, 40, 30, 20, 10]
```

✅ **Tip:** reverse() just flips the current order; it does **not sort** the list.

🔹 4️⃣ **Summary Table**

Method	Purpose	Modifies Original List?	Example
sort()	Sorts list in ascending/descending order	Yes	[3,1,2].sort() → [1,2,3]
sorted()	Returns new sorted list	No	sorted([3,1,2]) → [1,2,3]
reverse()	Reverses the list order	Yes	[1,2,3].reverse() → [3,2,1]

- **Example Combining All**

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

# Sort in ascending (in-place)
numbers.sort()
print("Sorted:", numbers)

# Reverse the list (in-place)
numbers.reverse()
print("Reversed:", numbers)

# Sorted without modifying original
numbers2 = [5, 3, 4, 1, 2]
print("Original:", numbers2)
print("Sorted New List:", sorted(numbers2))
print("Original After sorted():", numbers2)
```

Output:

```
Sorted: [10, 20, 30, 40]
Reversed: [40, 30, 20, 10]
Original: [5, 3, 4, 1, 2]
Sorted New List: [1, 2, 3, 4, 5]
Original After sorted(): [5, 3, 4, 1, 2]
```

Q8. Basic list manipulations: addition, deletion, updating, and slicing.

A. 🔹 1️⃣ **Addition to a List**

You can add elements using **append()**, **insert()**, or **concatenation (+)**.

a) Using append()

Adds an element at the **end** of the list.

```
fruits = ['apple', 'banana']
fruits.append('mango')
print(fruits)
```

Output:

```
['apple', 'banana', 'mango']
```

b) Using insert()

Inserts an element at a **specific index**.

```
fruits.insert(1, 'orange')  # Insert at index 1
print(fruits)
```

Output:

```
['apple', 'orange', 'banana', 'mango']
```

c) Using + (concatenation)

Combine two lists.

```
fruits = fruits + ['grapes', 'kiwi']
print(fruits)
```

Output:

```
['apple', 'orange', 'banana', 'mango', 'grapes', 'kiwi']
```

🔹 2️⃣ Deletion from a List

You can delete elements using **remove()**, **pop()**, or del.

a) Using remove()

Removes the **first occurrence** of a value.

```
fruits.remove('banana')
print(fruits)
```

Output:

```
['apple', 'orange', 'mango', 'grapes', 'kiwi']
```

b) Using pop()

Removes an element by **index** (default is the last element).

```
fruits.pop()
print(fruits)

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

Output:

```
['apple', 'orange', 'mango', 'grapes']
['apple', 'mango', 'grapes']
```

c) Using del

Deletes an element or a **slice**.

```
del fruits[0]  # Delete first element
print(fruits)

del fruits[0:2]  # Delete a range
print(fruits)
```

Output:

```
['mango', 'grapes']
[]
```

🔹 3️⃣ Updating a List

You can update elements using **index assignment** or **slicing**.

a) Update a single element

```
fruits = ['apple', 'banana', 'mango']
fruits[1] = 'orange'  # Change 'banana' to 'orange'
print(fruits)
```

Output:

```
['apple', 'orange', 'mango']
```

b) Update multiple elements using slicing

```
numbers = [10, 20, 30, 40, 50]
numbers[1:4] = [21, 31, 41]  # Update a slice
print(numbers)
```

Output:

[10, 21, 31, 41, 50]

🔹 4️⃣ **Slicing a List**

Slicing lets you **access a portion of a list**.

```
numbers = [10, 20, 30, 40, 50, 60]


print(numbers[1:4])  # From index 1 to 3
print(numbers[:3])   # First 3 elements
print(numbers[3:])   # From index 3 to end
print(numbers[::2])  # Every second element
print(numbers[::-1]) # Reverse list
```

Output:

```
[20, 30, 40]
[10, 20, 30]
[40, 50, 60]
[10, 30, 50]
[60, 50, 40, 30, 20, 10]
```

Tuple

Q9. Introduction to tuples, immutability.

A. A **tuple** is an **ordered collection of elements**, just like a list.

- **Key differences from a list:**
    1. Tuples are **immutable** (cannot change elements once created).
    2. Tuples are defined using **parentheses ()** instead of square brackets [].

Example:

```
# Creating a tuple
fruits = ('apple', 'banana', 'mango')
print(fruits)
```

Output:

```
('apple', 'banana', 'mango')
```

🔹 2️⃣ **Immutability of Tuples**

- Once a tuple is created, **you cannot modify, add, or remove elements**.

```
fruits = ('apple', 'banana', 'mango')


# Trying to change an element (will raise an error)
# fruits[1] = 'orange'  # TypeError: 'tuple' object does not support item assignment
```

✅ Immutability means **safer data storage** — the contents cannot be accidentally changed.

🔹 3️⃣ **Creating Tuples**

a) With multiple elements

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

b) Single element tuple

- **Important:** Include a **comma** for a single-element tuple.

```
single = (10,)
print(single)
```

Without the comma: single = (10) → it's just an integer, **not a tuple**.

c) Tuple without parentheses

```
fruits = 'apple', 'banana', 'mango'
print(fruits)
```

✅ Python automatically recognizes it as a tuple.

🔹 4️⃣ **Accessing Elements in a Tuple**

- Use **indexing** (positive and negative) and **slicing** — just like lists.

```
fruits = ('apple', 'banana', 'mango', 'orange')


print(fruits[0])   # 'apple'
print(fruits[-1])  # 'orange'
print(fruits[1:3]) # ('banana', 'mango')
```

🔹 5️⃣ Tuple Operations (No Modification)

Even though tuples are immutable, you can:

1. **Concatenate** tuples using +
2. **Repeat** tuples using *
3. **Check membership** using in

```
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
```

🔹 6️⃣ Why Use Tuples?

- Faster than lists because they are immutable.
- Useful for **data that should not change** (like coordinates, dates, or configuration values).
- Can be used as **keys in dictionaries** (lists cannot).

Q10. Creating and accessing elements in a tuple.

A. 1️⃣ Creating a Tuple

a) Tuple with multiple elements

```
fruits = ('apple', 'banana', 'mango')
print(fruits)
```

Output:

```
('apple', 'banana', 'mango')
```

b) Tuple without parentheses (tuple packing)

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

Output:

```
(10, 20, 30, 40)
```

c) Single element tuple

Important: Use a trailing comma, otherwise it is not considered a tuple.

```
single = (10,)
print(single)
print(type(single))  # <class 'tuple'>
```

Without the comma:

```
not_a_tuple = (10)
print(type(not_a_tuple))  # <class 'int'>
```

🔹 2️⃣ Accessing Elements in a Tuple

Tuples are **ordered**, so you can access elements by **indexing and slicing**, just like lists.

a) Positive Indexing

```
fruits = ('apple', 'banana', 'mango', 'orange')
print(fruits[0])  # First element
print(fruits[2])  # Third element
```

Output:

```
apple
mango
```

b) Negative Indexing

```
print(fruits[-1])  # Last element
print(fruits[-2])  # Second last element
```

Output:

```
orange
mango
```

c) Slicing a Tuple

```
print(fruits[1:3])   # Elements from index 1 to 2
print(fruits[:2])    # First two elements
print(fruits[2:])    # From index 2 to end
```

Output:

```
('banana', 'mango')
('apple', 'banana')
('mango', 'orange')
```

🔹 3️⃣ **Iterating Over a Tuple**

You can loop through a tuple using a for loop:

```
for fruit in fruits:
   print(fruit)
```

Output:

```
apple
banana
mango
orange
```

Q11. Basic operations with tuples: concatenation, repetition, membership.

A. 1️⃣ Concatenation (+)

- **Purpose:** Combine two or more tuples into a **new tuple**.
- **Important:** Tuples are immutable, so concatenation **creates a new tuple**.

Example:

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)


t3 = t1 + t2
print(t3)
```

Output:

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

✅ Works for **tuples of any data type**:

```
t1 = ('a', 'b')
t2 = ('c', 'd')
print(t1 + t2)  # ('a', 'b', 'c', 'd')
```

🔹 2️⃣ **Repetition (*)**

- **Purpose:** Repeat a tuple multiple times to create a **new tuple**.
- Syntax: tuple * n (n = number of repetitions)

Example:

```
t = (1, 2, 3)
t_repeated = t * 3
print(t_repeated)
```

Output:

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

✅ Useful when you need a **fixed pattern repeated**.

🔹 3️⃣ **Membership (in / not in)**

- **Purpose:** Check whether an element **exists in a tuple**.
- in → True if element exists
- not in → True if element does **not exist**

Example:

```
fruits = ('apple', 'banana', 'mango')

print('banana' in fruits)     # True
print('orange' in fruits)     # False
print('orange' not in fruits) # True
```

Output:

```
True
False
True
```

✅ Can be used in **conditional statements**:

```
if 'apple' in fruits:
   print("Apple is available!")
```

Output:

```
Apple is available!
```

Accessing Tuples

Q12. Accessing tuple elements using positive and negative indexing.

A. 1️⃣ Positive Indexing

- **Positive indexing** starts from **0** for the first element, **1** for the second, and so on.

- Use it to **access elements from the beginning** of the tuple.

Example:

```
fruits = ('apple', 'banana', 'mango', 'orange')


print(fruits[0])  # First element
print(fruits[2])  # Third element
```

Output:

```
apple
mango
```

✅ **Tip:** Index len(fruits)-1 gives the **last element** using positive indexing.

🔹 2️⃣ **Negative Indexing**

- **Negative indexing** starts from **-1** for the **last element**, -2 for the second last, and so on.
- Use it to **access elements from the end** of the tuple.

Example:

```
fruits = ('apple', 'banana', 'mango', 'orange')


print(fruits[-1])  # Last element
print(fruits[-2])  # Second last element
```

Output:

```
orange
mango
```

✅ Very useful when you want elements **from the end** without calculating the length.

🔹 3️⃣ **Slicing with Indexing**

- You can also use positive and negative indices to **slice a tuple**.

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


print(numbers[1:4])    # Positive indexing: index 1 to 3
print(numbers[-5:-2])  # Negative indexing: same slice
```

Output:

(20, 30, 40)

(20, 30, 40)

🔹 4️⃣ **Quick Reference Table**

Index	Element	Description
0	First element	Positive indexing
1	Second element	Positive indexing
-1	Last element	Negative indexing
-2	Second last element	Negative indexing

🔹 5️⃣ **Iterating Using Indexes**

```
fruits = ('apple', 'banana', 'mango', 'orange')


for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Output:

Index 0: apple

Index 1: banana

Index 2: mango

Index 3: orange

✅ Combine with **negative indexing** if needed:

```
for i in range(-1, -len(fruits)-1, -1):
    print(fruits[i])
```

Output (reverse):

orange

mango

banana

apple

Q13. Slicing a tuple to access ranges of elements.

A. Slicing lets you extract a **subset of elements** from a tuple.

- Syntax:

  tuple_name[start:end:step]

Part	Meaning
start	Index where slice begins (included)
end	Index where slice ends (excluded)
step	Optional, number of elements to skip

🔹 2️⃣ **Basic Slicing**

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


print(numbers[1:4])  # Elements from index 1 to 3
print(numbers[:3])   # First 3 elements
print(numbers[3:])   # From index 3 to end
```

Output:

```
(20, 30, 40)
(10, 20, 30)
(40, 50, 60)
```

✅ Remember: **start is included, end is excluded**.

🔹 3️⃣ **Using Negative Indices**

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


print(numbers[-5:-2])  # From 2nd element to 3rd last
print(numbers[-3:])    # Last 3 elements
```

Output:

```
(20, 30, 40)
(40, 50, 60)
```

✅ Negative indices count from the **end of the tuple**.

🔹 4️⃣ **Using Step Value**

- Step defines **how many elements to skip**.

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


print(numbers[::2])   # Every second element
print(numbers[1::2])  # Every second element starting from index 1
```

Output:

```
(10, 30, 50)
(20, 40, 60)
```

🔹 5️⃣ **Reversing a Tuple Using Slicing**

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


print(numbers[::-1])  # Reverse the tuple
```

Output:

```
(60, 50, 40, 30, 20, 10)
```

✅ Step = -1 means **move backward** through the tuple.

Dictionaries

Q14. Introduction to dictionaries: key-value pairs.

A. 1️⃣ Dictionary

- A **dictionary** is a **collection of key-value pairs** in Python.
- Each **key** is unique and maps to a **value**.
- Dictionaries are **unordered** (Python ≥3.7 maintains insertion order).
- Syntax uses **curly braces {}**.

Example:

```
# Creating a dictionary
student = {
```

```
    'name': 'Alice',
    'age': 20,
    'course': 'Python'
}

print(student)
```

Output:

```
{'name': 'Alice', 'age': 20, 'course': 'Python'}
```

🔹 2️⃣ **Key-Value Pairs**

- **Key:** Used to **identify** a value. Must be **immutable** (string, number, tuple).
- **Value:** Can be **any data type** (string, number, list, tuple, etc.).

Example:

```
student = {
    'name': 'Alice',      # key: 'name', value: 'Alice'
    'age': 20,            # key: 'age', value: 20
    'grades': [85, 90, 95] # key: 'grades', value: [85, 90, 95]
}
```

✅ Keys must be **unique**, but values can **repeat**:

```
info = {'x': 10, 'y': 10}
```

🔹 3️⃣ **Accessing Values**

- Use **square brackets [key]** or the **get() method**.

```
student = {'name': 'Alice', 'age': 20}

print(student['name'])     # Alice
print(student.get('age'))  # 20
```

✅ Difference:

- [key] → gives an **error** if key doesn't exist.

- get(key) → returns **None** (or a default value) if key doesn't exist.

```
print(student.get('course', 'Not Found'))  # Not Found
```

🔹 4️⃣ **Adding & Updating Key-Value Pairs**

```
student = {'name': 'Alice', 'age': 20}

# Adding a new key
student['course'] = 'Python'

# Updating an existing key
student['age'] = 21

print(student)
```

Output:

```
{'name': 'Alice', 'age': 21, 'course': 'Python'}
```

🔹 5️⃣ **Deleting Key-Value Pairs**

```
student = {'name': 'Alice', 'age': 21, 'course': 'Python'}

# Remove a key
del student['course']

# Remove last inserted key
student.popitem()

print(student)
```

Q15. Accessing, adding, updating, and deleting dictionary elements.

A. 🧩 1. Accessing Dictionary Elements

You can access values in a dictionary using their **keys**.

```
# Example dictionary
student = {"name": "Zaid", "age": 20, "course": "Python"}

# Accessing values using keys
print(student["name"])   # Output: Zaid
print(student["age"])    # Output: 20

# Using get() method (safer way)
print(student.get("course"))     # Output: Python
print(student.get("grade", "Not Found"))  # Output: Not Found (default value)
```

➕ 2. Adding Elements

You can add a new key–value pair by simply assigning it.

```
student["grade"] = "A"   # Adding a new key-value pair
print(student)
# Output: {'name': 'Zaid', 'age': 20, 'course': 'Python', 'grade': 'A'}
```

🔄 3. Updating Elements

To update an existing key's value:

```
student["age"] = 21   # Updating existing key
print(student)
# Output: {'name': 'Zaid', 'age': 21, 'course': 'Python', 'grade': 'A'}

# You can also update multiple items using update()
student.update({"course": "Data Science", "grade": "A+"})
print(student)
# Output: {'name': 'Zaid', 'age': 21, 'course': 'Data Science', 'grade': 'A+'}
```

❌ 4. Deleting Elements

You can delete elements in multiple ways:

```
# Using del keyword
del student["grade"]
print(student)
# Output: {'name': 'Zaid', 'age': 21, 'course': 'Data Science'}

# Using pop() — removes and returns the value
age = student.pop("age")
print(age)      # Output: 21
print(student)   # Output: {'name': 'Zaid', 'course': 'Data Science'}

# Using popitem() — removes the last inserted item (Python 3.7+)
student["city"] = "Mumbai"
student["year"] = 2025
student.popitem()
print(student)
# Output: {'name': 'Zaid', 'course': 'Data Science', 'city': 'Mumbai'}

# Using clear() — removes all items
student.clear()
print(student)
# Output: {}
```

Q16. Dictionary methods like keys(), values(), and items().

A. 🧩 Example Dictionary

```
student = {
    "name": "Zaid",
    "age": 20,
    "course": "Python",
    "grade": "A"
}
```

🔑 1. keys() Method

The **keys()** method returns a *view object* containing all the **keys** in the dictionary.

```
print(student.keys())
# Output: dict_keys(['name', 'age', 'course', 'grade'])


# You can loop through keys
for key in student.keys():
    print(key)
```

Output:

```
name
age
course
grade
```

✅ **Tip:** You can convert it to a list if needed.

```
key_list = list(student.keys())
print(key_list)
```

Output: ['name', 'age', 'course', 'grade']

💎 2. values() Method

The **values()** method returns a *view object* containing all the **values** in the dictionary.

```
print(student.values())
```

Output: dict_values(['Zaid', 20, 'Python', 'A'])

```
# Loop through values
for value in student.values():
    print(value)
```

Output:

```
Zaid
20
Python
A
```

✅ **Tip:** Convert to list if you want indexing or sorting.

```
value_list = list(student.values())
print(value_list)
```

Output: ['Zaid', 20, 'Python', 'A']

🧠 **3. items() Method**

The **items()** method returns all **key–value pairs** as tuples inside a *view object*.

print(student.items())

Output: dict_items([('name', 'Zaid'), ('age', 20), ('course', 'Python'), ('grade', 'A')])

You can use it to loop through both **keys and values** easily:

```
for key, value in student.items():
   print(key, ":", value)
```

Output:

```
name : Zaid
age : 20
course : Python
grade : A
```

Working with Dictionaries

Q17. Iterating over a dictionary using loops.

A. We'll use this example dictionary throughout 👇

```
student = {
   "name": "Zaid",
   "age": 20,
   "course": "Python",
   "grade": "A"
}
```

🔁 **1. Looping Through Keys (default)**

When you loop directly over a dictionary, it automatically iterates over its **keys**.

```
for key in student:
```

```
    print(key)
```

✅ **Output:**

```
name
age
course
grade
```

Or more clearly:

```
for key in student.keys():
    print(key)
```

(same output)

🧩 2. Looping Through Values

If you want only the **values**, use the .values() method.

```
for value in student.values():
    print(value)
```

✅ **Output:**

```
Zaid
20
Python
A
```

🧠 3. Looping Through Key–Value Pairs

To get both keys and values at the same time, use the .items() method.

```
for key, value in student.items():
    print(key, ":", value)
```

✅ **Output:**

```
name : Zaid
age : 20
course : Python
grade : A
```

⚡ **Bonus Examples**

➕ **Example: Display formatted information**

```
for key, value in student.items():
    print(f"The student's {key} is {value}")
```

✅ Output:

```
The student's name is Zaid
The student's age is 20
The student's course is Python
The student's grade is A
```

Q21. Merging two lists into a dictionary using loops or zip().

A. 🧩 Example Lists

```
keys = ["name", "age", "course"]
values = ["Zaid", 20, "Python"]
```

🌀 **1. Using a Loop**

We can loop through both lists using their indexes and create a dictionary step-by-step.

```
result = {}
for i in range(len(keys)):
    result[keys[i]] = values[i]


print(result)
```

✅ **Output:**

```
{'name': 'Zaid', 'age': 20, 'course': 'Python'}
```

⚡ **2. Using the zip() Function (Easier Way)**

zip() combines two (or more) sequences element by element — perfect for this task!

```
result = dict(zip(keys, values))
print(result)
```

✅ **Output:**

{'name': 'Zaid', 'age': 20, 'course': 'Python'}

🧠 **How zip() Works**

print(list(zip(keys, values)))

Output: [('name', 'Zaid'), ('age', 20), ('course', 'Python')]

Then, dict() converts these pairs into a dictionary.

⚙️ **3. Handling Unequal Lengths**

If the lists are not the same length, zip() stops at the shortest one.

```
keys = ["name", "age", "course", "grade"]
values = ["Zaid", 20, "Python"]

result = dict(zip(keys, values))
print(result)
```

✅ **Output:**

{'name': 'Zaid', 'age': 20, 'course': 'Python'}

(The extra key "grade" is ignored.)

If you want to handle missing values, you can use itertools.zip_longest():

```
from itertools import zip_longest

result = dict(zip_longest(keys, values, fillvalue="N/A"))
print(result)
```

✅ **Output:**

{'name': 'Zaid', 'age': 20, 'course': 'Python', 'grade': 'N/A'}

Q22. Counting occurrences of characters in a string using dictionaries.

A. 🧩 Example String

text = "programming"

🔁 1. Using a Loop and Dictionary

We'll create an empty dictionary and count how many times each character appears.

```
text = "programming"
count = {}  # empty dictionary

for char in text:
    if char in count:
        count[char] += 1  # increment count if char exists
    else:
        count[char] = 1   # initialize count if char not in dict

print(count)
```

✅ **Output:**

```
{'p': 1, 'r': 2, 'o': 1, 'g': 2, 'a': 1, 'm': 2, 'i': 1, 'n': 1}
```

⚡ 2. Using the get() Method (Cleaner Way)

Instead of an if–else, we can use get() to simplify:

```
text = "programming"
count = {}

for char in text:
    count[char] = count.get(char, 0) + 1

print(count)
```

✅ **Same Output**

```
{'p': 1, 'r': 2, 'o': 1, 'g': 2, 'a': 1, 'm': 2, 'i': 1, 'n': 1}
```

🧠 3. Ignoring Case or Spaces

You can preprocess the string before counting:

```
text = "Hello World".replace(" ", "").lower()
count = {}
```

```
for char in text:
    count[char] = count.get(char, 0) + 1


print(count)
```

✅ **Output:**

```
{'h': 1, 'e': 1, 'l': 3, 'o': 2, 'w': 1, 'r': 1, 'd': 1}
```

⚙️ **4. Using collections.Counter (Shortcut Way)**

Python provides a built-in tool just for this purpose.

```
from collections import Counter


text = "programming"
count = Counter(text)
print(count)
```

✅ **Output:**

```
Counter({'r': 2, 'g': 2, 'm': 2, 'p': 1, 'o': 1, 'a': 1, 'i': 1, 'n': 1})
```

Functions

Q23. Defining functions in Python.

A. 🧩 1. Function

A **function** is a block of reusable code that performs a specific task.
It helps make your program organized, readable, and avoids repetition.

🧠 **2. Function Syntax**

```
def function_name(parameters):
    """Optional docstring explaining the function."""
    # function body
    statement(s)
    return value
```

Example:

```
def greet():
    print("Hello, welcome to Python!")
```

Calling the function:

```
greet()
```

✅ **Output:**

```
Hello, welcome to Python!
```

⚙️ **3. Function with Parameters**

You can pass data to a function using **parameters**.

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

Calling:

```
greet_user("Zaid")
```

✅ **Output:**

```
Hello, Zaid
```

🔁 **4. Function with Return Value**

Functions can **return** results using the return statement.

```
def add(a, b):
    return a + b


result = add(5, 3)
print("Sum is:", result)
```

✅ **Output:**

```
Sum is: 8
```

🧱 **5. Function with Default Parameter**

You can give default values to parameters.

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

Calling:

```
greet()        # uses default value
greet("Zaid")  # overrides default
```

✅ **Output:**

```
Hello, User
Hello, Zaid
```

🧠 **6. Function with Multiple Parameters**

```
def student_info(name, age, course):
    print(f"Name: {name}, Age: {age}, Course: {course}")


student_info("Zaid", 20, "Python")
```

✅ **Output:**

```
Name: Zaid, Age: 20, Course: Python
```

💡 **7. Returning Multiple Values**

You can return multiple values (as a tuple).

```
def calculate(a, b):
    add = a + b
    sub = a - b
    return add, sub


result = calculate(10, 5)
print(result)        # Output: (15, 5)
```

Or unpack them:

```
sum_result, diff_result = calculate(10, 5)
print(sum_result)     # 15
print(diff_result)    # 5
```

Q24. Different types of functions: with/without parameters, with/without return values.

A. 🧩 1. Function Without Parameters and Without Return Value

- It **does not take any input** (no parameters).

- It **does not return anything** — only performs an action (like printing).

```
def greet():
    print("Hello! Welcome to Python.")
```

Calling:

```
greet()
```

✅ **Output:**

```
Hello! Welcome to Python.
```

🟢 **Use When:** You just want to display or perform an action, not calculate or return data.

🧠 **2. Function With Parameters and Without Return Value**

- It **accepts input(s)** through parameters.
- It **does not return** any value, only performs an operation (like printing results).

```
def greet_user(name):
    print("Hello,", name, "Welcome to Python!")
```

Calling:

```
greet_user("Zaid")
```

✅ **Output:**

```
Hello, Zaid Welcome to Python!
```

🟢 **Use When:** You need to perform an operation using inputs but don’t need to return data.

⚙️ **3. Function Without Parameters but With Return Value**

- It **does not take input**.
- It **returns** a value to the caller.

```
def get_pi():
    return 3.14159
```

Calling:

```
pi_value = get_pi()
print("Value of pi:", pi_value)
```

✅ **Output:**

```
Value of pi: 3.14159
```

🟢 **Use When:** You don't need input but want to send back a result.

🔁 **4. Function With Parameters and With Return Value**

- It **accepts inputs**.
- It **returns** a value (result of computation).

```
def add(a, b):
    return a + b
```

Calling:

```
result = add(10, 5)
print("Sum:", result)
```

✅ **Output:**

```
Sum: 15
```

🟢 **Use When:** You want to process input and return the computed output.

Q25. Anonymous functions (lambda functions).

A. 🧩 1. What is a Lambda Function?

A **lambda function** is a **small, one-line anonymous function** — meaning it doesn't have a name like normal functions defined with def.

It's used for **short, simple operations**, usually where a function is needed only once.

⚙️ **2. Syntax**

```
lambda arguments: expression
```

✅ **Note:**

- lambda keyword defines the function.
- It can take **any number of arguments**, but **only one expression**.
- It automatically **returns** the value of that expression.

🧠 **3. Example 1: Simple Lambda Function**

```
# Normal function
def add(a, b):
    return a + b
```

```
# Same using lambda
add_lambda = lambda a, b: a + b

print(add_lambda(5, 3))
```

✅ **Output:**

```
8
```

💡 4. Example 2: Lambda with One Argument

```
square = lambda x: x * x
print(square(4))
```

✅ **Output:**

```
16
```

🔁 5. Example 3: Using Lambda Inside Another Function

You can define a lambda **inside** a normal function.

```
def apply_operation(a, b, operation):
    return operation(a, b)

result = apply_operation(10, 5, lambda x, y: x - y)
print(result)
```

✅ **Output:**

```
5
```

🧱 6. Example 4: Using Lambda with Built-in Functions

Lambda functions are commonly used with **map()**, **filter()**, and **reduce()**.

➕ Using map() – apply operation to each element

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x * x, numbers))
print(squares)
```

✅ Output:

[1, 4, 9, 16, 25]

🔍 **Using filter() – filter elements based on condition**

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

even = list(filter(lambda x: x % 2 == 0, numbers))

print(even)

✅ Output:

[2, 4, 6]

➗ **Using reduce() – reduce list to single value**

from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x * y, numbers)

print(product)

✅ Output:

24

Modules

Q25. Introduction to Python modules and importing modules.

A. 🧩 1. What is a Module in Python?

A **module** in Python is simply a **file containing Python code** — functions, variables, or classes — that you can **reuse** in other programs.

✅ **Purpose:**

- To organize code into smaller, manageable parts
- To avoid repetition (reuse code easily)
- To improve readability and maintainability

📘 2. Types of Modules

Type	Description	Example
Built-in modules	Already included in Python	math, random, datetime, os, sys
User-defined modules	Created by you	my_module.py
External modules	Installed via pip	numpy, pandas, requests

🧠 3. Using Built-in Modules

You can use the import statement to load a module.

Example:

```
import math


print(math.sqrt(16))    # Square root
print(math.pi)          # Pi constant
```

✅ **Output:**

```
4.0
3.141592653589793
```

⚙️ 4. Importing Specific Functions

You don't always need to import the whole module — you can import specific functions.

```
from math import sqrt, pi


print(sqrt(25))
print(pi)
```

✅ **Output:**

```
5.0
3.141592653589793
```

🖍️ 5. Using an Alias (Nickname)

You can use an **alias** to shorten module names using as.

```
import math as m
```

```
print(m.pow(2, 3))  # 2 raised to 3 = 8
```

✅ **Output:**

```
8.0
```

🧱 6. Creating Your Own Module (User-defined)

You can make your own .py file and import it like any other module.

Step 1️⃣: Create a file my_module.py

```
# my_module.py
def greet(name):
    print(f"Hello, {name}!")

def add(a, b):
    return a + b
```

Step 2️⃣: Use it in another file

```
import my_module

my_module.greet("Zaid")
result = my_module.add(5, 10)
print("Sum:", result)
```

✅ **Output:**

```
Hello, Zaid!
Sum: 15
```

📦 7. Importing External Modules

You can install external modules using pip (Python's package installer):

```
pip install requests
```

Then use it:

```
import requests

response = requests.get("https://api.github.com")
```

print(response.status_code)

✅ **Output:**

200

Q26. Standard library modules: math, random.

A. Perfect 👏 — let's now learn about two of the most commonly used **Python standard library modules**:
✅ math (for mathematical operations)
✅ random (for generating random numbers).

🧮 1. The math Module

The **math module** provides mathematical constants and functions.

👉 **Importing the module**

import math

🔹 **Commonly Used math Functions**

Function	Description	Example	Output
math.sqrt(x)	Square root	math.sqrt(16)	4.0
math.pow(x, y)	x raised to power y	math.pow(2, 3)	8.0
math.floor(x)	Round **down** to nearest integer	math.floor(3.8)	3
math.ceil(x)	Round **up** to nearest integer	math.ceil(3.2)	4
math.pi	Constant π	math.pi	3.141592653589793
math.e	Constant e	math.e	2.718281828459045
math.factorial(x)	Factorial of x	math.factorial(5)	120
math.sin(x)	Sine (x in radians)	math.sin(math.pi/2)	1.0
math.cos(x)	Cosine (x in radians)	math.cos(0)	1.0
math.log(x)	Natural log (base e)	math.log(10)	2.302585...

🔸 **Example: Using math module**

import math

```
r = 5
area = math.pi * math.pow(r, 2)
print("Area of circle:", area)
```

✅ **Output:**

```
Area of circle: 78.53981633974483
```

🎲 2. The random Module

The **random module** is used to generate random numbers and choices.

👉 **Importing the module**

```
import random
```

🔹 **Commonly Used random Functions**

Function	Description	Example	Output
random.random()	Random float between 0.0 and 1.0	random.random()	0.3745...
random.randint(a, b)	Random integer between a and b (inclusive)	random.randint(1, 10)	7
random.randrange(start, stop, step)	Random integer from range	random.randrange(0, 10, 2)	4
random.choice(sequence)	Randomly choose one item	random.choice(['red','blue','green'])	'green'
random.shuffle(list)	Randomly shuffle list	random.shuffle(my_list)	(list rearranged)
random.uniform(a, b)	Random float between a and b	random.uniform(1, 5)	3.67...

🔸 **Example: Using random module**

```
import random

colors = ["red", "green", "blue", "yellow"]
print("Random color:", random.choice(colors))
```

```
print("Random integer (1–6):", random.randint(1, 6))


nums = [1, 2, 3, 4, 5]
random.shuffle(nums)
print("Shuffled list:", nums)
```

✅ **Possible Output:**

```
Random color: blue
Random integer (1–6): 4
Shuffled list: [5, 1, 3, 2, 4]
```

Q27. Creating custom modules.

A. 🧩 1. What is a Custom Module?

A **custom module** is simply a **Python file (.py) containing functions, variables, or classes** that you create yourself.
You can **reuse it in other Python programs** by importing it.

✅ **Purpose:**

- Organize code
- Reuse functions across multiple programs
- Keep projects clean and maintainable

🧠 2. Steps to Create a Custom Module

Step 1: Create a Python file

Create a file named my_module.py and add some functions or variables.

```
# my_module.py


def greet(name):
    print(f"Hello, {name}!")


def add(a, b):
```

```
    return a + b

pi = 3.14159
```

Step 2: Import the custom module in another file

Create another Python file (for example, main.py) in the **same directory**.

```
# main.py
import my_module

my_module.greet("Zaid")
result = my_module.add(5, 10)
print("Sum:", result)
print("Value of pi:", my_module.pi)
```

✅ **Output:**

```
Hello, Zaid!
Sum: 15
Value of pi: 3.14159
```

Step 3: Import Specific Items

You can also import specific functions or variables:

```
from my_module import greet, pi

greet("Zaid")
print("Value of pi:", pi)
```

✅ **Output:**

```
Hello, Zaid!
Value of pi: 3.14159
```

Step 4: Use Alias

You can give your module an alias for convenience:

```
import my_module as mm
```

```
mm.greet("Zaid")

print(mm.add(7, 3))
```

✅ **Output:**

```
Hello, Zaid!

10
```

⚙️ **3. Important Notes**

1. **Directory**: The .py file (module) must be in the same directory or in a folder recognized by Python.
2. **Naming**: Module names should be **valid identifiers** (letters, numbers, underscores) and **cannot start with a number**.
3. **Reloading Modules**: If you modify a module while your program is running, use importlib.reload() to reload it.

```
import importlib

import my_module


importlib.reload(my_module)
```