Module)14-Python – Collections, functions and Modules

Q.1 Understanding how to create and access elements in a list

Ans: In Python, a **list** is a collection of items that can hold different data types. Here's how you can **create** and **access** elements in a list:

1. Creating a List

You can create a list using square brackets [] and separate elements with commas:

```
# A list of numbers

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

# A list of strings

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

# A mixed list (different data types)

mixed_list = [10, "hello", 3.14, True]
```

2. Accessing Elements in a List

a) Indexing (Access by Position)

Python uses **zero-based indexing**, meaning the first element is at index 0.

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

print(fruits[0])  # Output: apple

print(fruits[1])  # Output: banana

print(fruits[2])  # Output: cherry
```

You can also use **negative indexing** to access elements from the end:

```
print(fruits[-1])  # Output: cherry (last element)

print(fruits[-2])  # Output: banana (second last)
```

b) Slicig (Access a Range of Elements)

You can extract multiple elements using slicing:

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

print(numbers[1:4])   # Output: [20, 30, 40] (index 1 to 3)

print(numbers[:3])    # Output: [10, 20, 30] (start to index 2)

print(numbers[3:])    # Output: [40, 50, 60] (index 3 to end)

print(numbers[-3:])   # Output: [40, 50, 60] (last 3 elements)
```

c) Accessing Elements in a Loop

To access all elements in a list, you can use a loop:

```
for fruit in fruits:

   print(fruit)
```

Q.2 Indexing in lists (positive and negative indexing).

Ans: In Python, **indexing** is used to access elements of a list. Python supports both **positive** and **negative** indexing.

1. Positive Indexing

- Indexing starts from 0 for the first element.
- The second element is at index 1, the third at 2, and so on.

Example:

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

print(fruits[0])  # Output: apple

print(fruits[1])  # Output: banana

print(fruits[2])  # Output: cherry

print(fruits[3])  # Output: date
```

If you try to access an index that is out of range, Python will throw an IndexError.

2. Negative Indexing

- Negative indices start from -1 (last element), -2 (second last), and so on.
- This allows easy access to elements from the end of the list.

Example:

```
print(fruits[-1])  # Output: date  (Last element)

print(fruits[-2])  # Output: cherry (Second last element)

print(fruits[-3])  # Output: banana

print(fruits[-4])  # Output: apple
```

- fruits[-1] refers to "date", which is the last element.
- fruits[-2] refers to "cherry", which is the second last element.

Q.3 Slicing a list: accessing a range of elements.

Ans: In Python, **slicing** is used to extract a portion of a list by specifying a start and end index. It follows the syntax:

```
list[start:end:step]
```

- start → The index where the slice begins (inclusive). Default is 0.
- end → The index where the slice stops (exclusive).
- step → The gap between indices (optional, default is 1).

1. Basic Slicing (Start and End)

```
numbers = [10, 20, 30, 40, 50, 60, 70]

print(numbers[1:4])   # Output: [20, 30, 40] (Index 1 to 3)

print(numbers[:3])    # Output: [10, 20, 30] (Start from 0)

print(numbers[3:])    # Output: [40, 50, 60, 70] (Index 3 to end)
print(numbers[:])    # Output: [10, 20, 30, 40, 50, 60, 70] (Full list)
```

2. Negative Indexing in Slicing

Negative indices can be used to slice from the end.

```
print(numbers[-4:])   # Output: [40, 50, 60, 70] (Last 4 elements)
print(numbers[:-2])   # Output: [10, 20, 30, 40, 50] (Exclude last 2)
print(numbers[-5:-2]) # Output: [30, 40, 50] (From -5 to -3)
```

3. Using Step in Slicing

The step value defines how many elements to skip.

```
print(numbers[::2])   # Output: [10, 30, 50, 70] (Every second element)
print(numbers[1::2])  # Output: [20, 40, 60] (Every second element starting from index 1)
print(numbers[::-1])  # Output: [70, 60, 50, 40, 30, 20, 10] (Reversed list)
print(numbers[::-2])  # Output: [70, 50, 30] (Reversed with step 2)
```

Q.4 Common list operations: concatenation, repetition, membership

Ans: Python provides several operations for working with lists, including **concatenation**, **repetition**, and **membership testing**.

1. Concatenation (+)

Concatenation is used to combine two or more lists into a single list using the + operator.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2
print(result)  # Output: [1, 2, 3, 4, 5, 6]
```

- Both lists remain unchanged; a new list is created.

2. Repetition (*)

Repetition allows repeating a list multiple times using the * operator.

```
list1 = ["A", "B", "C"]
result = list1 * 3
print(result)  # Output: ['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C']
```

- Useful for initializing lists with repeated values.

3. Membership Testing (in and not in)

Checks whether an element exists in a list.

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)   # Output: True
print("grape" in fruits)    # Output: False
```

print("grape" not in fruits) # Output: True

- Returns True if the element is present, otherwise False.

Q.5 Understanding list methods like append(), insert(), remove(), pop().

Ans: Python provides several built-in methods to modify lists. Let's explore four commonly used ones:

1. append() – Add an element to the end

The append() method adds a single element to the **end** of a list.

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

fruits.append("cherry")

print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

- **Modifies the original list.**
- **Can only add one element at a time.**

2. insert() – Add an element at a specific index

The insert(index, element) method inserts an element at a specific position.

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

numbers.insert(2, 30)  # Insert 30 at index 2

print(numbers)  # Output: [10, 20, 30, 40, 50]
```

- **Does not replace elements, shifts them forward.**

3. remove() – Remove the first occurrence of an element

The remove(element) method removes the first occurrence of the specified element.

```
colors = ["red", "blue", "green", "blue"]

colors.remove("blue")  # Removes the first "blue"

print(colors)  # Output: ['red', 'green', 'blue']
```

- **Raises an error if the element is not found.**

4. pop() – Remove an element by index (or last element by default)

The pop(index) method removes an element at the specified index and **returns it**. If no index is given, it removes the last element.

```
animals = ["cat", "dog", "rabbit"]

removed_element = animals.pop(1)  # Removes "dog" at index 1

print(animals)        # Output: ['cat', 'rabbit']

print(removed_element) # Output: 'dog'
```

- **If no index is specified, pop() removes the last element.**
- **Raises an error if the index is out of range.**

Q.6 Iterating over a list using loops.

Ans: In Python, you can iterate over a list using different types of loops. Here are the most common ways:

1. Using a for Loop (Simple Iteration)

The easiest way to iterate over a list is using a for loop.

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

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

Output:

```
apple
banana
cherry
```

- This method is simple and efficient.

2. Using range() and len() (Index-Based Iteration)

If you need access to the index while looping:

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

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

Output:

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

- This is useful when modifying elements inside the loop.

3. Using enumerate() (Best for Index and Value)

enumerate() provides both **index** and **value** in a loop.

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

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

Output:

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

- This is cleaner than range(len(list)).

4. Using a while Loop

A while loop can iterate until a condition is met.

```
numbers = [10, 20, 30, 40]
i = 0

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

Output:

```
10
20
30
40
```

- Useful when looping with a **dynamic condition**.

5. Iterating in Reverse (reversed())

To loop over a list in **reverse order**:

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

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

Output:

```
cherry
banana
apple
```

6. Iterating with List Comprehension

List comprehensions allow **compact iteration**:

```
numbers = [1, 2, 3, 4]
squared = [x**2 for x in numbers]

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

- Best for **transforming** lists efficiently.

Q.7 Sorting and reversing a list using sort(), sorted(), and reverse().

Ans: Python provides several built-in methods to sort and reverse lists efficiently.

Let's explore sort(), sorted(), and reverse().

1. Sorting a List Using sort() (Modifies Original List)

The sort() method sorts a list **in place**, meaning it modifies the original list.

```
numbers = [5, 2, 9, 1, 5, 6]

numbers.sort()

print(numbers)  # Output: [1, 2, 5, 5, 6, 9]
```

- **Modifies the original list.**
- **Default is ascending order (small → large).**

Sorting in Descending Order

Use reverse=True to sort in **descending order**:

```
numbers.sort(reverse=True)

print(numbers)  # Output: [9, 6, 5, 5, 2, 1]
```

2. Sorting a List Using sorted() (Returns a New List)

The sorted() function returns a **new sorted list** without modifying the original.

```
numbers = [5, 2, 9, 1, 5, 6]

sorted_numbers = sorted(numbers)

print(sorted_numbers)  # Output: [1, 2, 5, 5, 6, 9]

print(numbers)  # Original list remains unchanged
```

- **Returns a new sorted list.**
- **Does NOT modify the original list.**

Descending Order Sorting with sorted()

```
sorted_numbers_desc = sorted(numbers, reverse=True)

print(sorted_numbers_desc)  # Output: [9, 6, 5, 5, 2, 1]
```

3. Reversing a List Using reverse() (Modifies Original List)

The reverse() method **reverses** the order of elements **in place**.

```
numbers = [5, 2, 9, 1, 5, 6]

numbers.reverse()


print(numbers)  # Output: [6, 5, 1, 9, 2, 5]
```

- **Does NOT sort—just reverses the current order.**
- **Modifies the original list.**

4. Reversing a List Using [::-1] (Creates a New List)

```
numbers = [5, 2, 9, 1, 5, 6]

reversed_numbers = numbers[::-1]
```

```
print(reversed_numbers)  # Output: [6, 5, 1, 9, 2, 5]
```

- **Creates a new reversed list.**
- **Original list remains unchanged.**

Q.8 Basic list manipulations: addition, deletion, updating, and slicing.

Ans: Python allows various operations to **add, delete, update, and slice** lists efficiently.

1. Adding Elements to a List

You can add elements using append(), insert(), and extend().

A. Using append() (Adds to End)

```
fruits = ["apple", "banana"]
fruits.append("cherry")


print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

B. Using insert() (Adds at Specific Index)

```
fruits.insert(1, "mango")  # Insert "mango" at index 1
print(fruits)  # Output: ['apple', 'mango', 'banana', 'cherry']
```

C. Using extend() (Merge Two Lists)

```
fruits.extend(["grape", "orange"])
print(fruits)  # Output: ['apple', 'mango', 'banana', 'cherry', 'grape', 'orange']
```

2. Deleting Elements from a List

You can remove elements using remove(), pop(), and del.

A. Using remove() (Removes First Occurrence)

```
numbers = [10, 20, 30, 20, 40]
numbers.remove(20)  # Removes first 20


print(numbers)  # Output: [10, 30, 20, 40]
```

B. Using pop() (Removes by Index and Returns Element)

```
removed_element = numbers.pop(2)  # Removes element at index 2
print(removed_element)  # Output: 20
print(numbers)  # Output: [10, 30, 40]
```

C. Using del (Delete by Index or Entire List)

```
del numbers[1]  # Deletes element at index 1
print(numbers)  # Output: [10, 40]
```

```
del numbers  # Deletes entire list
```

3. Updating Elements in a List

Lists allow updating elements by directly assigning new values.

```
colors = ["red", "blue", "green"]
colors[1] = "yellow"  # Change "blue" to "yellow"


print(colors)  # Output: ['red', 'yellow', 'green']
```

- **Can also update multiple elements at once:**

```
colors[0:2] = ["purple", "black"]
print(colors)  # Output: ['purple', 'black', 'green']
```

4. Slicing a List (Extracting Portions)

Slicing allows accessing a range of elements.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]


print(numbers[2:5])   # Output: [3, 4, 5] (Index 2 to 4)
print(numbers[:3])    # Output: [1, 2, 3] (First 3 elements)
print(numbers[4:])    # Output: [5, 6, 7, 8] (From index 4 to end)
print(numbers[::2])   # Output: [1, 3, 5, 7] (Every 2nd element)
print(numbers[::-1])  # Output: [8, 7, 6, 5, 4, 3, 2, 1] (Reversed list)
```

Q.9 Introduction to tuples, immutability.

Ans: A **tuple** is an ordered, immutable collection of elements, similar to a list but with **fixed values**.

1. Creating Tuples

Tuples are defined using **parentheses ()** or simply by separating values with commas.

```
# Creating tuples
tuple1 = (1, 2, 3)
tuple2 = "apple", "banana", "cherry"  # Without parentheses
tuple3 = (10,)  # Single-element tuple (comma is necessary)
print(tuple1)  # Output: (1, 2, 3)
print(tuple2)  # Output: ('apple', 'banana', 'cherry')
print(tuple3)  # Output: (10,)
```

- **Without a comma (tuple3 = (10)) Python treats it as an integer!**

2. Accessing Tuple Elements

Tuples support **indexing and slicing** just like lists.

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

print(fruits[0])   # Output: apple

print(fruits[-1])  # Output: cherry

print(fruits[1:])  # Output: ('banana', 'cherry')
```

3. Immutability of Tuples

Tuples **cannot be modified** after creation. Any attempt to change a value will raise an error.

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

numbers[1] = 50  # ❌ TypeError: 'tuple' object does not support item assignment
```

Why Use Tuples?

✅ **Faster than lists** (better performance).
✅ **Immutable** (prevents accidental modification).
✅ **Can be used as dictionary keys** (lists cannot).

4. Tuple Packing & Unpacking

You can **pack** multiple values into a tuple and **unpack** them into variables.

```
# Packing

person = ("John", 25, "USA")


# Unpacking

name, age, country = person

print(name)    # Output: John

print(age)     # Output: 25

print(country) # Output: USA
```

5. Tuple Methods

Since tuples are immutable, they have only **two built-in methods**:

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

print(numbers.count(10))  # Output: 3 (Counts occurrences of 10)

print(numbers.index(30))  # Output: 3 (Finds first occurrence of 30)
```

Q.10 Creating and accessing elements in a tuple.

Ans: Tuples in Python are **ordered, immutable sequences** that store multiple values. Let's explore how to create and access elements in a tuple.

1. Creating a Tuple

A. Using Parentheses ()

Tuples are defined by enclosing elements in **parentheses**.

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

print(fruits)  # Output: ('apple', 'banana', 'cherry')
```

B. Without Parentheses (Tuple Packing)

Tuples can also be created without parentheses—just separating values with commas.

```
colors = "red", "green", "blue"

print(colors)  # Output: ('red', 'green', 'blue')
```

C. Single-Element Tuple (Comma Required!)

A **single element tuple** must have a trailing comma, otherwise Python treats it as a different type.

```
single_tuple = (5,)  # ✅ Correct

not_a_tuple = (5)   # ❌ Incorrect, this is an integer


print(type(single_tuple))  # Output: <class 'tuple'>

print(type(not_a_tuple))   # Output: <class 'int'>
```

2. Accessing Elements in a Tuple

A. Using Indexing

Tuples support **zero-based indexing**, just like lists.

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


print(numbers[0])   # Output: 10

print(numbers[2])   # Output: 30

print(numbers[-1])  # Output: 40 (Negative Indexing)
```

🔹 **Negative indexing** allows access from the end (-1 is the last element).

B. Using Slicing (:)

Tuple slicing allows extracting a **subset** of elements.

```
letters = ("a", "b", "c", "d", "e")


print(letters[1:4])  # Output: ('b', 'c', 'd')

print(letters[:3])   # Output: ('a', 'b', 'c')

print(letters[::2])  # Output: ('a', 'c', 'e')  (Every 2nd element)

print(letters[::-1]) # Output: ('e', 'd', 'c', 'b', 'a') (Reversed)
```

Q.11 Basic operations with tuples: concatenation, repetition, membership.

Ans: Tuples support several operations like **concatenation, repetition, and membership checking**. Let’s explore them with examples.

1. Tuple Concatenation (+)

Tuples can be **joined** using the + operator, which creates a **new tuple**.

```
tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)


result = tuple1 + tuple2

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

- **Note:** Since tuples are **immutable**, concatenation does **not modify the original tuples**; it returns a new one.

2. Tuple Repetition (*)

You can **repeat** a tuple multiple times using the * operator.

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

repeated = numbers * 3


print(repeated)  # Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

- **Useful for initializing tuples with repeated values.**

3. Membership Check (in, not in)

You can check if an element **exists** in a tuple using in or not in.

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


print("banana" in fruits)   # Output: True

print("grape" not in fruits)  # Output: True
```

- **Fast lookup** since tuples are indexed.

Q.12 Accessing tuple elements using positive and negative indexing.

Ans: Tuples in Python are **ordered** and allow element access using **indexing**. Python supports both **positive** and **negative** indexing.

1. Accessing Elements with Positive Indexing (0, 1, 2, ...)

Positive indexing starts from **0** and moves forward.

```
fruits = ("apple", "banana", "cherry", "date")


print(fruits[0])  # Output: apple

print(fruits[2])  # Output: cherry

print(fruits[3])  # Output: date
```

- 🔹 **Index starts at 0** → fruits[0] returns "apple".
- 🔹 **Last element index** → fruits[3] returns "date".

2. Accessing Elements with Negative Indexing (-1, -2, -3, ...)

Negative indexing allows accessing elements **from the end**.

```
print(fruits[-1])  # Output: date

print(fruits[-2])  # Output: cherry

print(fruits[-4])  # Output: apple
```

- 🔹 **-1 is the last element**, -2 is the second-last, and so on.

3. Handling Index Errors

Trying to access an index out of range **raises an error**.

```
print(fruits[4])   # ❌ IndexError: tuple index out of range

print(fruits[-5])  # ❌ IndexError: tuple index out of range
```

Key Takeaways

✅ **Positive Indexing:** Starts from 0 and moves forward.
✅ **Negative Indexing:** Starts from -1 (last element) and moves backward.
✅ **Avoid Index Errors:** Ensure the index is within the tuple length.

Q.13 Slicing a tuple to access ranges of elements.

Ans: Tuple **slicing** allows extracting a **subset** of elements using the syntax:

```
tuple[start:end:step]
```

- **start** → Index where the slice begins (inclusive).
- **end** → Index where the slice stops (**exclusive**).
- **step** → Defines the interval between elements (**default is 1**).

1. Basic Slicing (start:end)

Extracts elements from **start index to (end - 1)**.

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


print(numbers[1:4])  # Output: (20, 30, 40)
```

- 🔹 Starts at index **1** (20), stops **before index 4** (50).

2. Omitting start or end

- 🔹 If **start is omitted**, slicing begins from the first element.
- 🔹 If **end is omitted**, slicing continues to the last element.

```
print(numbers[:3])   # Output: (10, 20, 30)   → First 3 elements

print(numbers[2:])   # Output: (30, 40, 50, 60) → From index 2 to end
```

3. Using a Step (start:end:step)

The step defines **how many elements to skip**.

print(numbers[::2]) # Output: (10, 30, 50) → Every 2nd element

print(numbers[1:5:2]) # Output: (20, 40) → Skips 1 element each time

4. Negative Slicing (Reversing a Tuple)

Negative indices allow slicing **backward**.

print(numbers[::-1]) # Output: (60, 50, 40, 30, 20, 10) → Reversed tuple

print(numbers[::-2]) # Output: (60, 40, 20) → Every 2nd element in reverse

Key Takeaways

✅ **tuple[start:end]** extracts a slice from **start to end-1**.
✅ **Omitting start or end** includes all elements before/after the given index.
✅ **Using step** skips elements (::2 → every second element).
✅ **Negative slicing ([::-1])** reverses a tuple.

Q.14 Introduction to dictionaries: key-value pairs.

Ans: A **dictionary** in Python is a collection of **key-value pairs**. Unlike lists or tuples, dictionaries store data in an **unordered, mutable, and indexed** way, allowing fast lookups and modifications.

1. Creating a Dictionary

Dictionaries are defined using **curly braces {}** with key-value pairs separated by colons :.

```
# Creating a dictionary

student = {

   "name": "Alice",

   "age": 22,

   "course": "Computer Science"

}

print(student)

# Output: {'name': 'Alice', 'age': 22, 'course': 'Computer Science'}
```

- **Keys must be unique and immutable** (strings, numbers, or tuples).
- **Values can be of any data type** (strings, numbers, lists, other dictionaries).

2. Accessing Dictionary Elements

A. Using Keys (dict[key])

print(student["name"]) # Output: Alice

print(student["age"])   # Output: 22

- **Accessing a non-existing key** causes a KeyError.

B. Using .get() Method (Safer Way)

```
print(student.get("course"))    # Output: Computer Science

print(student.get("gender", "Not specified"))  # Output: Not specified
```

- Returns None or a **default value** if the key is missing.

3. Dictionary Characteristics

✅ **Unordered** → Elements have no fixed order (Python 3.6+ maintains insertion order).
✅ **Mutable** → You can modify values.
✅ **Fast Lookups** → Uses a **hash table** for quick access.

Q.15 Accessing, adding, updating, and deleting dictionary elements

Ans: **Accessing, Adding, Updating, and Deleting Dictionary Elements in Python** Dictionaries in Python store **key-value pairs**, allowing efficient data retrieval and modification. Let’s explore how to **access, add, update, and delete elements** in a dictionary.

1. Accessing Dictionary Elements

You can retrieve values using **keys**.

A. Using dict[key] (Direct Access)

```
student = {"name": "Alice", "age": 22, "course": "Computer Science"}


print(student["name"])  # Output: Alice

print(student["age"])   # Output: 22
```

- **Raises KeyError if the key is missing.**

B. Using get() (Safer Approach)

```
print(student.get("course"))     # Output: Computer Science

print(student.get("gender", "Not Available"))  # Output: Not Available
```

- **Returns None or a default value if the key does not exist.**

2. Adding Elements to a Dictionary

New key-value pairs can be added dynamically.

```
student["gender"] = "Female"

student["GPA"] = 3.8

print(student)

# Output: {'name': 'Alice', 'age': 22, 'course': 'Computer Science', 'gender': 'Female', 'GPA': 3.8}
```

- If the key **does not exist**, a new entry is created.

3. Updating Dictionary Elements

A. Updating a Single Key

```
student["age"] = 23  # Changing age from 22 to 23

print(student["age"])  # Output: 23
```

B. Using .update() to Modify Multiple Keys

```
student.update({"age": 24, "course": "Data Science"})

print(student)

# Output: {'name': 'Alice', 'age': 24, 'course': 'Data Science', 'gender': 'Female', 'GPA': 3.8}
```

◆ **Efficient when updating multiple values at once.**

4. Deleting Dictionary Elements

A. Using del Statement

```
del student["GPA"]

print(student)

# Output: {'name': 'Alice', 'age': 24, 'course': 'Data Science', 'gender': 'Female'}
```

◆ **Raises KeyError if the key does not exist.**

B. Using .pop() Method

```
age = student.pop("age")

print(age)  # Output: 24

print(student)

# Output: {'name': 'Alice', 'course': 'Data Science', 'gender': 'Female'}
```

◆ **Removes and returns the value of the key.**

C. Using .popitem() (Removes the Last Inserted Item)

```
student.popitem()

print(student)

# Output: {'name': 'Alice', 'course': 'Data Science'}
```

◆ **Useful when handling Python 3.7+ where dictionaries maintain insertion order.**

D. Using .clear() (Removes All Elements)

```
student.clear()

print(student)  # Output: {}
```

◆ **Completely empties the dictionary.**

Key Takeaways

✅ **Access elements** using dict[key] or get().

✅ **Add new key-value pairs** dynamically.

✅ **Update values** with direct assignment or .update().

✅ **Delete elements** using del, .pop(), .popitem(), or .clear().

Q.16 Dictionary methods like keys(), values(), and items().

Ans: Python provides built-in methods to access different parts of a dictionary efficiently.

1. Accessing Keys with keys()

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

```
student = {"name": "Alice", "age": 22, "course": "Computer Science"}


print(student.keys())

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


# Converting keys to a list

keys_list = list(student.keys())

print(keys_list)

# Output: ['name', 'age', 'course']
```

- **Useful when checking if a key exists** (if "name" in student.keys():).

2. Accessing Values with values()

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

```
print(student.values())

# Output: dict_values(['Alice', 22, 'Computer Science'])


# Converting values to a list

values_list = list(student.values())

print(values_list)

# Output: ['Alice', 22, 'Computer Science']
```

- **Great for searching specific values** (if 22 in student.values():).

3. Accessing Key-Value Pairs with items()

The .items() method returns **key-value pairs** as tuples.

```
print(student.items())

# Output: dict_items([('name', 'Alice'), ('age', 22), ('course', 'Computer Science')])


# Converting to a list of tuples

items_list = list(student.items())

print(items_list)

# Output: [('name', 'Alice'), ('age', 22), ('course', 'Computer Science')]
```

- ◆ **Useful for looping through key-value pairs**.

4. Using These Methods in a Loop

A. Iterating Over Keys

```
for key in student.keys():
    print(key)
# Output:
# name
# age
# course
```

B. Iterating Over Values

```
for value in student.values():
    print(value)
# Output:
# Alice
# 22
# Computer Science
```

C. Iterating Over Key-Value Pairs

```
for key, value in student.items():
    print(f"{key}: {value}")
# Output:
# name: Alice
# age: 22
# course: Computer Science
```

Key Takeaways

✅ **keys()** → Returns all keys.

✅ **values()** → Returns all values.

✅ **items()** → Returns key-value pairs as tuples.

✅ **Efficient for looping & checking dictionary contents.**

Q.17 Iterating over a dictionary using loops

Ans: **Dictionaries store key-value pairs, and you can iterate through them using loops to access keys, values, or both.**

1. Iterating Over Keys (for key in dict)

You can loop through the dictionary keys directly.

```
student = {"name": "Alice", "age": 22, "course": "Computer Science"}
```

```
for key in student:
    print(key)
# Output:
# name
# age
# course
```

- Equivalent to using .keys()

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

2. Iterating Over Values (for value in dict.values())

To iterate over values instead of keys:

```
for value in student.values():
    print(value)
# Output:
# Alice
# 22
# Computer Science
```

3. Iterating Over Key-Value Pairs (for key, value in dict.items())

To get both keys and values, use .items().

```
for key, value in student.items():
    print(f"{key}: {value}")
# Output:
# name: Alice
# age: 22
# course: Computer Science
```

4. Using enumerate() to Get Indexes

You can get the index of dictionary items using enumerate().

```
for index, (key, value) in enumerate(student.items()):
    print(f"{index}: {key} → {value}")
# Output:
# 0: name → Alice
# 1: age → 22
# 2: course → Computer Science
```

5. Iterating Over a Nested Dictionary

If a dictionary contains another dictionary, loop through it like this:

```
students = {
    "Alice": {"age": 22, "course": "CS"},
    "Bob": {"age": 24, "course": "Math"}
}
for name, details in students.items():
    print(f"Student: {name}")
    for key, value in details.items():
        print(f"  {key}: {value}")
# Output:
# Student: Alice
#   age: 22
#   course: CS
# Student: Bob
#   age: 24
#   course: Math
```

Key Takeaways

✅ **Looping through keys → for key in dict: or dict.keys()**
✅ **Looping through values → dict.values()**
✅ **Looping through key-value pairs → dict.items()**
✅ **Using enumerate() for indexes**
✅ **Handling nested dictionaries with nested loops**

Q.18 Merging two lists into a dictionary using loops or zip().

Ans: Merging two lists into a dictionary is a common task where **one list acts as keys** and **the other as values**.

1. Using zip() (Most Efficient Way)

The zip() function pairs elements from both lists and converts them into a dictionary using dict().

```
 keys = ["name", "age", "course"]
values = ["Alice", 22, "Computer Science"]

# Merging with zip()
student_dict = dict(zip(keys, values))

print(student_dict)
# Output: {'name': 'Alice', 'age': 22, 'course': 'Computer Science'}
```

- **zip() stops at the shortest list** (if they are of different lengths).

2. Using a Loop (for Loop)

If you prefer a **manual approach**, use a for loop.

```
keys = ["name", "age", "course"]
values = ["Alice", 22, "Computer Science"]

student_dict = {}

for i in range(len(keys)):
    student_dict[keys[i]] = values[i]

print(student_dict)
# Output: {'name': 'Alice', 'age': 22, 'course': 'Computer Science'}
```

- **Make sure both lists have the same length** to avoid IndexError.

3. Using Dictionary Comprehension

A more **concise** and **Pythonic** way:

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

print(student_dict)
# Output: {'name': 'Alice', 'age': 22, 'course': 'Computer Science'}
```

- **Similar to the loop but written in one line.**

4. Handling Unequal List Lengths Gracefully

If the lists are **unequal**, zip_longest() from itertools can fill missing values.

```
from itertools import zip_longest

keys = ["name", "age"]
values = ["Alice"]  # Shorter list
student_dict = dict(zip_longest(keys, values, fillvalue="N/A"))
print(student_dict)
# Output: {'name': 'Alice', 'age': 'N/A'}
```

- **fillvalue="N/A"** ensures missing values are handled

Key Takeaways

✅ **zip()** → Best and simplest way.

✅ **Looping (for)** → Manual but clear.

✅ **Dictionary Comprehension** → Concise and Pythonic.

✅ **zip_longest()** → Handles different-length lists.

Q.19 Counting occurrences of characters in a string using dictionaries.

Ans: You can count how many times each character appears in a string using dictionaries in multiple ways.

Let’s explore different approaches.

1. Using a for Loop (Basic Approach)

```
text = "hello"

char_count = {}

for char in text:

    if char in char_count:

        char_count[char] += 1

    else:

        char_count[char] = 1

print(char_count)

# Output: {'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

- 🔹 **Checks if a character exists** in the dictionary and increments the count.
- 🔹 **Adds new characters dynamically** as they appear in the string.

2. Using get() Method (Cleaner Approach)

The .get() method helps avoid the explicit if check.

```
text = "hello"

char_count = {}

for char in text:

    char_count[char] = char_count.get(char, 0) + 1

print(char_count)

# Output: {'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

- 🔹 **get(char, 0)** initializes the count to 0 if the character is not found.

3. Using collections.Counter (Best for Large Strings)

The Counter class from the collections module makes counting very efficient.

```
from collections import Counter

text = "hello"

char_count = Counter(text)

print(char_count)

# Output: Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
```

- **Fastest method** as it's optimized for counting operations.
- **Returns a Counter dictionary-like object**.

4. Using Dictionary Comprehension

If you want a one-liner solution:

```
text = "hello"

char_count = {char: text.count(char) for char in set(text)}

print(char_count)

# Output: {'o': 1, 'l': 2, 'h': 1, 'e': 1}
```

- **Uses set(text)** to iterate over unique characters.
- **Not the most efficient** (text.count(char) runs separately for each character).

Key Takeaways

✅ **Basic for loop** → Works well for small cases.

✅ **Using get()** → Cleaner and avoids conditionals.

✅ **Counter()** → Best for performance.

✅ **Dictionary comprehension** → One-liner but inefficient for large texts.

Q.20 Defining functions in Python

Ans: A function in Python is a reusable block of code that performs a specific task. Functions help **organize code, reduce redundancy, and improve readability**.

1. Defining a Function (def keyword)

A function is defined using the def keyword, followed by a **function name**, parentheses ( ), and a **colon :**.

```
def greet():

   print("Hello, welcome to Python!")
```

- The function **does nothing until called**.

Calling the Function

python

CopyEdit

```
greet()

# Output: Hello, welcome to Python!
```

- A function must be **called** to execute its code.

2. Function with Parameters

You can pass **arguments** (inputs) to a function.

```
def greet(name):

   print(f"Hello, {name}!")

greet("Alice")
```

```
# Output: Hello, Alice!
```

- **Parameters (name) are placeholders** for values passed during function calls.

3. Function with Multiple Parameters

```
def add(a, b):
    return a + b
result = add(5, 3)
print(result)  # Output: 8
```

- **return statement** sends back a value from the function.

4. Default Parameters

If no value is provided, a default is used.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
greet()         # Output: Hello, Guest!
greet("Bob")     # Output: Hello, Bob!
```

5. Keyword Arguments (key=value format)

You can specify arguments by name.

```
def introduce(name, age):
    print(f"My name is {name} and I am {age} years old.")
introduce(age=25, name="Alice")
# Output: My name is Alice and I am 25 years old.
```

- **Order doesn’t matter** when using keyword arguments.

6. Variable-Length Arguments (*args and **kwargs)

A. *args (Multiple Positional Arguments)

Allows passing multiple values as a tuple.

```
def total(*numbers):
    return sum(numbers)
print(total(1, 2, 3, 4))  # Output: 10
```

B. **kwargs (Multiple Keyword Arguments)

Stores extra arguments in a dictionary.

```
def display_info(**details):
    for key, value in details.items():
        print(f"{key}: {value}")
display_info(name="Alice", age=22, city="NY")
```

```
    # Output:

    # name: Alice

    # age: 22

    # city: NY
```

7. Lambda (Anonymous) Functions

A **one-line function** using lambda.

```
square = lambda x: x * x

print(square(5))  # Output: 25
```

🔹 **Useful for short, simple operations**.

Key Takeaways

✅ **Use def** to define functions.
✅ **Use return** to send values back.
✅ **Default parameters** prevent errors.
✅ **Use *args for multiple positional arguments**.
✅ **Use **kwargs for multiple keyword arguments**.
✅ **Lambda functions** are concise and anonymous.

Q.21 Different types of functions: with/without parameters, with/without return values.

Ans: Functions in Python can be classified based on **parameters (input)** and **return values (output)**. Let's go through all possible types.

1. Function Without Parameters & Without Return Value

✅ **Takes no arguments**
✅ **Does not return anything** (just prints something)

```
def greet():

   print("Hello, welcome to Python!")

greet()

# Output: Hello, welcome to Python!
```

🔹 **Use case:** When you want a function to just execute a task without needing input.

2. Function With Parameters & Without Return Value

✅ **Takes arguments as input**
✅ **Does not return anything** (just prints or modifies something)

```
def greet(name):

   print(f"Hello, {name}!")

greet("Alice")

# Output: Hello, Alice!
```

🔹 **Use case:** When you need to provide input but don’t need to return a result.

3. Function Without Parameters & With Return Value

✅ **Takes no arguments**
✅ **Returns a value instead of printing**

```
def get_message():

   return "Hello, welcome to Python!"

msg = get_message()

print(msg)

# Output: Hello, welcome to Python!
```

🔹 **Use case:** When a function needs to generate and return a value.

4. Function With Parameters & With Return Value

✅ **Takes input arguments**
✅ **Returns a computed value**

```
def add(a, b):

   return a + b

result = add(5, 3)

print(result)

# Output: 8
```

🔹 **Use case:** When a function performs a calculation and returns the result.

Q.22 Anonymous functions (lambda functions).

Ans: A **lambda function** is a small **anonymous function** (a function without a name). It can have multiple arguments but **only one expression**.

1. Syntax of Lambda Function

```
lambda arguments: expression
```

🔹 **No def keyword** is needed.
🔹 **Returns the result automatically** (no return statement required).

2. Basic Example

A lambda function to **add 10 to a number**:

```
add_ten = lambda x: x + 10

print(add_ten(5))  # Output: 15
```

🔹 Equivalent to:

```
def add_ten(x):

   return x + 10
```

3. Lambda with Multiple Arguments

A lambda function to **add two numbers**:

```
add = lambda a, b: a + b

print(add(3, 7))  # Output: 10
```

- Equivalent to:

```
def add(a, b):

   return a + b
```

4. Lambda with if-else (Conditional Expression)

A lambda function to check if a number is **even or odd**:

```
even_or_odd = lambda x: "Even" if x % 2 == 0 else "Odd"

print(even_or_odd(4))  # Output: Even

print(even_or_odd(7))  # Output: Odd
```

5. Using Lambda Inside Higher-Order Functions

A. Using map()

Applies a function to each element in an iterable.

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

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

print(squared)

# Output: [1, 4, 9, 16, 25]
```

B. Using filter()

Filters elements based on a condition.

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

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

print(even_numbers)

# Output: [2, 4, 6]
```

C. Using sorted() with Lambda

Sort a list of tuples based on the second element.

```
students = [("Alice", 85), ("Bob", 90), ("Charlie", 80)]

sorted_students = sorted(students, key=lambda x: x[1])

print(sorted_students)

# Output: [('Charlie', 80), ('Alice', 85), ('Bob', 90)]
```

6. Lambda Inside a Function

A function returning a lambda function.

```
def multiplier(n):

   return lambda x: x * n
```

```
double = multiplier(2)

triple = multiplier(3)

print(double(5))  # Output: 10

print(triple(5))  # Output: 15
```

Key Takeaways

✅ **Lambda functions are anonymous (no name).**

✅ **Used for short, single-expression functions.**

✅ **Often used in map(), filter(), sorted(), etc.**

✅ **Cannot contain multiple statements or assignments.**

Q.23 Introduction to Python modules and importing modules.

Ans: **A module in Python is a file containing Python code (functions, variables, and classes) that can be used in other programs. Modules help in code reusability, organization, and maintenance.**

1. What is a Module?

A module is simply a .py file containing Python code.

Example: A module named math_operations.py

```
# math_operations.py

def add(a, b):

   return a + b

def multiply(a, b):

   return a * b
```

This module can now be imported and used in another script.

2. Importing Modules

Python provides multiple ways to import modules.

A. Importing the Entire Module

```
import math_operations

print(math_operations.add(3, 5))      # Output: 8

print(math_operations.multiply(3, 5))  # Output: 15
```

◆ You must use module_name.function_name() when calling functions.

B. Importing Specific Functions (from module import function)

```
from math_operations import add

print(add(3, 5))  # Output: 8
```

◆ You don't need to prefix the function with the module name.

◆ This imports only the add function, not multiply.

C. Importing Everything (from module import *)

```
from math_operations import *
print(add(3, 5))       # Output: 8
print(multiply(3, 5))  # Output: 15
```

- Imports all functions and variables in the module.
- Not recommended for large modules (can cause conflicts).

D. Importing with an Alias (import module as alias)

```
import math_operations as mo
print(mo.add(3, 5))  # Output: 8
```

- Saves typing effort by using a short alias.

3. Python's Built-in Modules

Python comes with many pre-installed (built-in) modules.

A. Using the math Module

```
import math
print(math.sqrt(25))  # Output: 5.0
print(math.pi)       # Output: 3.141592653589793
```

B. Using the random Module

```
import random
print(random.randint(1, 10))  # Random number between 1 and 10
print(random.choice(["apple", "banana", "cherry"]))  # Random choice
```

C. Using the datetime Module

```
import datetime
now = datetime.datetime.now()
print(now)  # Output: Current date and time
```

4. Creating and Using Your Own Module

Step 1: Create a module (math_operations.py)

```
# math_operations.py
def subtract(a, b):
   return a - b
```

Step 2: Use the module in another script

```
import math_operations
print(math_operations.subtract(10, 5))  # Output: 5
```

5. Checking Available Functions in a Module (dir())

You can use dir() to list all functions in a module.

```
import math

print(dir(math))
```

- ◆ This prints a list of all available functions in the math module.

6. Finding Module Documentation (help())

```
import math

help(math)
```

- ◆ This shows detailed documentation about the module.

7. Installing External Modules (pip install)

Python allows you to install third-party modules using pip.

Example: Installing numpy

```
pip install numpy
```

Then, import and use it:

```
import numpy as np

arr = np.array([1, 2, 3])

print(arr)
```

Key Takeaways

✅ Modules help organize and reuse code.
✅ Use import to access built-in and custom modules.
✅ Built-in modules like math, random, and datetime are useful.
✅ pip allows installing external modules.

Q.24 Standard library modules: math, random

Ans: Python provides built-in **standard library modules** that contain useful functions. Two important modules are:

1. **math** → Provides mathematical functions.
2. **random** → Generates random numbers.

1. math Module (Mathematical Functions)

The math module provides advanced math operations like square roots, trigonometry, logarithms, and constants.

Importing math

```
import math
```

Common math Functions

Function	**Description**	**Example**
math.sqrt(x)	Square root of x	math.sqrt(25) → 5.0
math.pow(x, y)	x raised to power y	math.pow(2, 3) → 8.0

Function	Description	Example
math.factorial(x)	Factorial of x	math.factorial(5) → 120
math.floor(x)	Rounds down to nearest integer	math.floor(3.7) → 3
math.ceil(x)	Rounds up to nearest integer	math.ceil(3.2) → 4
math.fabs(x)	Absolute value	math.fabs(-10) → 10.0
math.log(x, base)	Logarithm of x (default base e)	math.log(10, 10) → 1.0
math.pi	Constant π (pi)	math.pi → 3.1415926535
math.e	Constant e (Euler's number)	math.e → 2.7182818284

Example: Using math Functions

```
import math

print(math.sqrt(16))   # Output: 4.0

print(math.factorial(5))  # Output: 120

print(math.ceil(3.2))  # Output: 4

print(math.pi)  # Output: 3.141592653589793
```

2. random Module (Random Number Generation)

The random module is used to **generate random numbers** for simulations, games, and security purposes.

Importing random

```
import random
```

Common random Functions

Function	Description	Example
random.randint(a, b)	Random integer between a and b (inclusive)	random.randint(1, 10)
random.random()	Random float between 0.0 and 1.0	random.random()
random.uniform(a, b)	Random float between a and b	random.uniform(1, 5)
random.choice(seq)	Random item from a list, tuple, or string	random.choice(["red", "blue", "green"])
random.shuffle(seq)	Shuffles the elements of a list **in place**	random.shuffle(my_list)
random.sample(seq, k)	Picks k random elements from a sequence	random.sample(range(1, 10), 3)

Example: Using random Functions

```
import random

print(random.randint(1, 10))  # Random number between 1 and 10

print(random.random())  # Random float between 0.0 and 1.0

print(random.choice(["apple", "banana", "cherry"]))  # Random choice from list
```

```
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)  # Shuffle list
print(numbers)
print(random.sample(range(1, 50), 5))  # Pick 5 random numbers from 1 to 49
```

Key Takeaways

✅ **math** provides **mathematical operations** (sqrt, pow, log, pi, factorial).
✅ **random** helps generate **random numbers, shuffle lists, and pick random choices**.
✅ **Both are part of Python's standard library**, so no installation is needed.

Q.25 Creating custom modules.

Ans: A **module** is a Python file (.py) that contains functions, classes, and variables. Creating custom modules allows you to **reuse code** and keep your project organized.

1. Creating a Custom Module

Let's create a module named **math_operations.py**.

Step 1: Define the Module (math_operations.py)

```
# math_operations.py
def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
def multiply(a, b):
    return a * b
def divide(a, b):
    if b == 0:
        return "Cannot divide by zero!"
    return a / b
```

🔹 This module contains basic mathematical functions.

2. Importing and Using a Custom Module

Now, create another Python script (e.g., **main.py**) to use the module.

Step 2: Import the Module in Another File (main.py)

```
import math_operations
print(math_operations.add(10, 5))      # Output: 15
print(math_operations.subtract(10, 5))  # Output: 5
print(math_operations.multiply(10, 5))  # Output: 50
print(math_operations.divide(10, 5))    # Output: 2.0
```

- **import module_name** is used to access functions from the module.
- Use **module_name.function_name()** to call functions.

3. Importing Specific Functions

You can import only the required functions instead of the entire module.

```
from math_operations import add, multiply

print(add(4, 2))       # Output: 6

print(multiply(4, 2))   # Output: 8
```

- Now, you don’t need to use **math_operations.** before calling functions.

4. Importing with an Alias

```
import math_operations as mo

print(mo.add(7, 3))  # Output: 10

print(mo.divide(8, 2))  # Output: 4.0
```

- **mo** is a shorter alias for math_operations, making the code cleaner.

5. Using if __name__ == "__main__" in a Module

If you want a module to run some code **only when executed directly**, use:

```
# math_operations.py

def add(a, b):

   return a + b

if __name__ == "__main__":

   print(add(2, 3))  # Output: 5 (Only runs when executed directly)
```

- This prevents the code from running when **imported** into another file.

6. Finding Module Location

To check where Python is loading a module from:

```
import math_operations

print(math_operations.__file__)  # Prints module location
```

7. Storing Modules in a Folder (Package Creation)

If you want to organize multiple modules in a folder:

Step 1: Create a folder, e.g., my_package/
Step 2: Add an **__init__.py** file (empty or with initialization code).
Step 3: Add your modules (math_operations.py, string_operations.py, etc.).

Importing from the package:

```
from my_package import math_operations

print(math_operations.add(3, 2))  # Output: 5
```

Key Takeaways

✅ A module is a Python file containing functions, variables, or classes.

✅ Import modules using import module_name.

✅ Use from module import function to import specific functions.

✅ Use import module as alias to create shorter names.

✅ Organize multiple modules into packages using folders and __init__.py.