# ***Module 7) Python – Collections, functions and Modules***

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

***Ans:***

#### 1. Creating a List

A list in Python is a collection of elements (items) stored in a single variable. Lists can hold **different data types** — numbers, strings, booleans, or even other lists.

**Syntax:**

```
list_name = [item1, item2, item3, ...]
```

**Examples:**

```
# List of integers
numbers = [10, 20, 30, 40]


# List of strings
fruits = ["apple", "banana", "cherry"]


# Mixed data types
mixed = [25, "hello", True, 3.14]


# Empty list
empty_list = []
```

---

**2. Accessing Elements in a List**

Python lists are **indexed**, meaning each element has a position number.

**Indexing**

- **First element** has index 0
- **Last element** has index -1 (negative indexing starts from the end)

**Example:**

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


print(fruits[0])   # apple  (first element)
print(fruits[1])   # banana (second element)
print(fruits[-1])  # cherry (last element)
```

---

**3. Accessing a Range of Elements (Slicing)**

You can get a portion of a list using **slicing**:

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


print(fruits[1:4])  # ['banana', 'cherry', 'mango']
print(fruits[:3])   # ['apple', 'banana', 'cherry']
print(fruits[2:])   # ['cherry', 'mango', 'grape']
```

---

**4. Accessing with a Loop**

You can loop through a list to access each element:

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

---

**5. Changing Elements**

Lists are **mutable** — you can change elements after creation:

```
fruits[1] = "blueberry"
print(fruits)  # ['apple', 'blueberry', 'cherry', 'mango', 'grape']
```

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

***Ans:***

**1. Positive Indexing**

Positive indexing starts **from the beginning** of the list.

- The first element is at **index 0**
- The second element is at **index 1**
- And so on...

**Example:**

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

print(fruits[0])  # apple   (1st element)
print(fruits[1])  # banana  (2nd element)
print(fruits[3])  # mango   (4th element)
```

**Index Map:**

```
Index:  0      1      2      3
Item:  apple   banana  cherry  mango
```

---

**2. Negative Indexing**

Negative indexing starts **from the end** of the list.

- The last element is at **index -1**
- The second last element is at **index -2**
- And so on...

**Example:**

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


print(fruits[-1])  # mango   (last element)
print(fruits[-2])  # cherry  (second last)
print(fruits[-4])  # apple   (first element)
```

**Negative Index Map:**

```
Index:  -4     -3     -2     -1
Item:  apple   banana  cherry  mango
```

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

***Ans:***

**1. Basic Syntax**

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

- **start** → index where slicing begins (**inclusive**)
- **end** → index where slicing stops (**exclusive**)
- Elements are returned from start **up to** but **not including** end.

---

**2. Example**

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

print(fruits[1:4])  # ['banana', 'cherry', 'mango']

📌 Here:

- Start at index **1** → "banana"
- End before index **4** → "grape" is not included.

---

**3. Omitting Start or End**

- **No start** → starts from **0**
- **No end** → goes to the **last element**

print(fruits[:3])  # ['apple', 'banana', 'cherry']

print(fruits[2:])  # ['cherry', 'mango', 'grape']

---

**4. Using Step**

You can add a **step** to skip elements:

list_name[start:end:step]

**Example:**

print(fruits[0:5:2])  # ['apple', 'cherry', 'grape']

- Takes every **2nd** element between index 0 and 4.

---

**5. Negative Index with Slicing**

Negative indexes also work in slicing:

print(fruits[-4:-1])  # ['banana', 'cherry', 'mango']

---

**6. Reverse Slicing**

You can reverse a list using slicing:

```
print(fruits[::-1])   # ['grape', 'mango', 'cherry', 'banana', 'apple']
```

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

***Ans:***

**1. Concatenation (+)**

You can **join two or more lists** using the + operator.
It creates a **new list** without changing the originals.

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

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

💡 You can also concatenate more than two lists:

```
all_items = list1 + list2 + ["extra"]
print(all_items)  # [1, 2, 3, 4, 5, 6, 'extra']
```

---

**2. Repetition (*)**

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

```
fruits = ["apple", "banana"]
print(fruits * 3)
# ['apple', 'banana', 'apple', 'banana', 'apple', 'banana']
```

📌 This is useful for creating repeated patterns or initializing lists.

---

**3. Membership (in, not in)**

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

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


print("apple" in fruits)     # True
print("mango" not in fruits) # True
```

💡 Membership checks are **case-sensitive**:

```
print("Apple" in fruits)  # False (capital A is different)
```

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

***Ans:***

**1. append()**

- **Adds** an element **to the end** of the list.
- Modifies the original list.

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

---

**2. insert()**

- **Inserts** an element **at a specific index**.
- Shifts the other elements to the right.

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

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

---

**3. remove()**

- **Removes the first occurrence** of a specific element.
- If the element is not found → ValueError.

```
fruits = ["apple", "banana", "cherry", "banana"]
fruits.remove("banana")  # removes the first 'banana'
print(fruits)
# ['apple', 'cherry', 'banana']
```

---

**4. pop()**

- Removes an element **by index** and returns it.
- If no index is given → removes and returns the **last element**.

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

last_item = fruits.pop()   # removes last element
print(last_item)          # cherry
print(fruits)             # ['apple', 'banana']

first_item = fruits.pop(0) # removes at index 0
print(first_item)         # apple
print(fruits)             # ['banana']
```

**Quick Comparison**

| Method | Action | Returns value? | Requires index? |
|---|---|---|---|
| append() | Adds element at the end | No | No |
| insert() | Adds element at a given position | No | Yes |
| remove() | Removes first occurrence of given value | No | No (needs value) |
| pop() | Removes element at a given index (default last) | Yes | Optional |

***6.Iterating over a list using loops.***

***Ans:***

**1. Using a for loop (Direct Access)**

This is the most common and simplest way.

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


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

**Output:**

```
apple
banana
```

cherry

📌 Here, fruit takes the value of each element in the list.

---

**2. Using a for loop with range() and indexing**

If you need the **index** as well as the value:

```
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

---

**3. Using enumerate() (Best way to get index + value)**

enumerate() gives both index and element in a clean way.

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

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

---

**4. Using a while loop**

Sometimes you might iterate with a counter.

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

```
i = 0


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

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

***Ans:***

**1. sort()**

- **Sorts the list in place** (changes the original list).
- By default → **ascending order**.
- Has an optional reverse=True parameter for descending order.

```
numbers = [4, 2, 9, 1]


numbers.sort()
print(numbers)  # [1, 2, 4, 9]


numbers.sort(reverse=True)
print(numbers)  # [9, 4, 2, 1]
```

---

**2. sorted()**

- **Returns a new sorted list** without changing the original.
- Also has reverse=True option.

```
numbers = [4, 2, 9, 1]
```

```
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # [1, 2, 4, 9]
print(numbers)         # [4, 2, 9, 1] (unchanged)

desc_sorted = sorted(numbers, reverse=True)
print(desc_sorted)     # [9, 4, 2, 1]
```

---

**3. reverse()**

- Reverses the **order of elements** in place (does not sort, just flips).
- Works on any order the list currently has.

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()
print(fruits)  # ['cherry', 'banana', 'apple']
```

---

**Quick Comparison Table**

| Method | Changes Original? | Sorts? | Returns New List? | Extra Options |
|---|---|---|---|---|
| sort() | ✅ Yes | ✅ Yes | ❌ No | reverse=True |
| sorted() | ❌ No | ✅ Yes | ✅ Yes | reverse=True |

| Method | Changes Original? | Sorts? | Returns New List? | Extra Options |
|---|---|---|---|---|
| reverse() | ✅ Yes | ❌ No (only reverses) | ❌ No | None |

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

***Ans:***

**1. Addition (Adding Elements)**

We can add elements in several ways:

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

# Add to the end
fruits.append("cherry")
print(fruits)  # ['apple', 'banana', 'cherry']

# Add at a specific position
fruits.insert(1, "mango")
print(fruits)  # ['apple', 'mango', 'banana', 'cherry']

# Add multiple elements at once
fruits.extend(["grape", "orange"])
print(fruits)  # ['apple', 'mango', 'banana', 'cherry', 'grape', 'orange']
```

---

**2. Deletion (Removing Elements)**

Different ways to remove elements:

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

# Remove by value (first occurrence)
fruits.remove("banana")
print(fruits)  # ['apple', 'cherry', 'banana']

# Remove by index (returns removed item)
removed_item = fruits.pop(1)
print(removed_item)  # cherry
print(fruits)        # ['apple', 'banana']

# Delete by index without return
del fruits[0]
print(fruits)        # ['banana']

# Remove all elements
fruits.clear()
print(fruits)        # []
```

---

**3. Updating (Changing Elements)**

Lists are **mutable**, so we can change elements directly:

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

# Update a single element
fruits[1] = "mango"
print(fruits)  # ['apple', 'mango', 'cherry']

# Update a range using slicing
fruits[0:2] = ["grape", "orange"]
print(fruits)  # ['grape', 'orange', 'cherry']
```

---

**4. Slicing (Accessing a Range of Elements)**

Slicing lets us access or update parts of a list:

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

# Get elements from index 1 to 3 (excluding 3)
print(fruits[1:3])   # ['banana', 'cherry']

# From start to index 2
print(fruits[:2])    # ['apple', 'banana']

# From index 2 to end
print(fruits[2:])    # ['cherry', 'mango', 'grape']

# Using step (every 2nd element)
```

```
print(fruits[::2])   # ['apple', 'cherry', 'grape']
```

---

✅ **Summary Table**

| Operation | Method / Syntax | Example |
|---|---|---|
| Add | append(), insert(), extend() | fruits.append("cherry") |
| Delete | remove(), pop(), del, clear() | fruits.pop(1) |
| Update | Index assignment, slicing | fruits[0] = "mango" |
| Slice | list[start:end:step] | fruits[1:4:2] |

***9.Introduction to tuples, immutability.***

***Ans:***

**1. What is a Tuple?**

A **tuple** in Python is a collection of elements, **similar to a list**, but **immutable** (unchangeable).
Tuples can store **different data types**: integers, strings, floats, booleans, or even other tuples.

**Syntax:**

```
tuple_name = (item1, item2, item3, ...)
```

**Examples:**

```
# Tuple of strings
fruits = ("apple", "banana", "cherry")
```

```
# Tuple of mixed data types
person = ("John", 25, True)

# Empty tuple
empty_tuple = ()

# Tuple with one element (needs a comma!)
single_tuple = ("apple",)
```

---

**2. Tuples vs Lists**

| Feature | List ([]) | Tuple (()) |
|---|---|---|
| Mutability | Mutable (can change) | Immutable (cannot change) |
| Syntax | Square brackets [] | Parentheses () |
| Performance | Slower (more flexible) | Faster (less flexible) |
| Use case | When data changes | When data stays constant |

---

**3. Immutability**

**Immutable** means once a tuple is created:

- You **cannot add** elements.

- You **cannot remove** elements.
- You **cannot modify** existing elements.

**Example:**

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

# Trying to change an element will cause an error

fruits[1] = "mango"  # ❌ TypeError: 'tuple' object does not support item assignment

📌 However, you **can**:

- Access elements (positive and negative indexing)
- Slice tuples
- Concatenate or repeat tuples (which creates new tuples)
- Convert a tuple to a list, modify it, then convert back to tuple.

---

**4. Why Use Tuples?**

- Data **should not change** (e.g., coordinates, days of the week)
- **Faster** than lists (better performance)
- Can be used as **dictionary keys** (lists cannot because they're mutable)
- Protects data integrity

***10.Creating and accessing elements in a tuple.***

***Ans:***

**1. Creating a Tuple**

Tuples are created using **parentheses ()** (or even without them — Python can infer it).
They can hold **any data type** and even be mixed.

```
# Tuple of strings
fruits = ("apple", "banana", "cherry")


# Tuple of mixed data types
person = ("John", 25, True)


# Empty tuple
empty_tuple = ()


# Tuple without parentheses (tuple packing)
coordinates = 10, 20, 30


# Tuple with one element (comma is required!)
single_tuple = ("apple",)
```

---

**2. Accessing Elements**

Tuples use **indexing** just like lists.
Indexes start at **0** (first element), and negative indexes start from the end (-1 = last element).

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


print(fruits[0])   # apple (first element)
```

```
print(fruits[2])   # cherry (third element)
print(fruits[-1])  # mango (last element)
print(fruits[-3])  # banana (third from last)
```

---

**3. Accessing a Range (Slicing)**

We can use slicing to get multiple elements at once:

```
print(fruits[1:3])   # ('banana', 'cherry')
print(fruits[:2])    # ('apple', 'banana')
print(fruits[2:])    # ('cherry', 'mango')
print(fruits[::-1])  # ('mango', 'cherry', 'banana', 'apple') — reversed tuple
```

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

***Ans:***

**1. Concatenation (+)**

You can join two tuples together to make a new tuple.
This **does not** modify the original tuples — it creates a new one.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

result = tuple1 + tuple2
print(result)
# (1, 2, 3, 4, 5, 6)
```

---

**2. Repetition (*)**

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

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


print(fruits * 3)
# ('apple', 'banana', 'apple', 'banana', 'apple', 'banana')
```

---

**3. Membership (in, not in)**

You can check if an element exists inside a tuple.

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


print("apple" in fruits)      # True
print("mango" not in fruits)  # True
```

📌 Membership checks are **case-sensitive**:

```
print("Apple" in fruits)  # False  (capital A is different)
```

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

***Ans:***

**1. Positive Indexing**

- Indexing starts from **0** for the first element.
- Increases from left to right.

**Example:**

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


print(fruits[0])  # apple   (first element)
```

```
print(fruits[1])  # banana  (second element)
print(fruits[3])  # mango   (fourth element)
```

**Index Map (Positive):**

```
Index:  0      1      2      3
Item:  apple   banana  cherry  mango
```

---

**2. Negative Indexing**

- Indexing starts from **-1** for the last element.
- Decreases from right to left.

**Example:**

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


print(fruits[-1])  # mango   (last element)
print(fruits[-2])  # cherry  (second last element)
print(fruits[-4])  # apple   (first element)
```

**Index Map (Negative):**

```
Index:  -4     -3     -2     -1
Item:  apple   banana  cherry  mango
```

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

***Ans:***

**1. Basic Syntax**

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

- **start** → index to begin slicing (**inclusive**)
- **end** → index to stop slicing (**exclusive**)

- Returns a **new tuple** with the selected elements.

---

**2. Example**

fruits = ("apple", "banana", "cherry", "mango", "grape")

print(fruits[1:4])  # ('banana', 'cherry', 'mango')

📌 Starts at index 1 ("banana") and stops before index 4 ("grape" is excluded).

---

**3. Omitting Start or End**

- **No start** → starts from index 0
- **No end** → goes till the last element

print(fruits[:3])   # ('apple', 'banana', 'cherry')

print(fruits[2:])   # ('cherry', 'mango', 'grape')

---

**4. Using Step**

You can skip elements using a **step value**:

print(fruits[0:5:2])  # ('apple', 'cherry', 'grape')

Here, 2 means “take every second element.”

---

**5. Negative Index Slicing**

Negative indexes can also be used:

print(fruits[-4:-1])  # ('banana', 'cherry', 'mango')

---

**6. Reversing a Tuple**

You can reverse elements with:

print(fruits[::-1])

# ('grape', 'mango', 'cherry', 'banana', 'apple')

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

***Ans:***

**1. What is a Dictionary?**

A **dictionary** in Python is a **collection of key–value pairs**.

- **Key** → unique identifier (like a label).
- **Value** → data or information stored for that key.
- Dictionaries are **unordered** in older Python versions (<3.7) but **preserve insertion order** in Python 3.7+.
- Created using **curly braces {}**.

---

**2. Syntax**

```
dict_name = {
   key1: value1,
   key2: value2,
   key3: value3
}
```

---

**3. Example**

```
student = {
   "name": "Alice",
```

```
    "age": 21,
    "course": "Computer Science"
}

print(student)
# {'name': 'Alice', 'age': 21, 'course': 'Computer Science'}
```

---

**4. Key–Value Concept**

Think of it like a **real dictionary**:

- The **word** is the **key**.
- The **meaning** is the **value**.

Example:

```
capitals = {
    "India": "New Delhi",
    "France": "Paris",
    "Japan": "Tokyo"
}

print(capitals["France"])  # Paris
```

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

***Ans:***

**1. Accessing Elements**

Use the **key** inside square brackets or the .get() method.

```
student = {
   "name": "Alice",
   "age": 21,
   "course": "Computer Science"
}

# Using key indexing (raises KeyError if key not found)
print(student["name"])  # Alice

# Using get() method (returns None or default value if key not found)
print(student.get("age"))      # 21
print(student.get("grade"))    # None
print(student.get("grade", "N/A"))  # N/A
```

---

**2. Adding Elements**

Assign a value to a new key:

```
student["grade"] = "A"
print(student)
# {'name': 'Alice', 'age': 21, 'course': 'Computer Science', 'grade': 'A'}
```

---

**3. Updating Elements**

Assign a value to an existing key:

```
student["age"] = 22
print(student)
# {'name': 'Alice', 'age': 22, 'course': 'Computer Science', 'grade':
'A'}
```

---

**4. Deleting Elements**

Use del statement or .pop() method.

```
# Using del (raises KeyError if key doesn't exist)
del student["grade"]
print(student)
# {'name': 'Alice', 'age': 22, 'course': 'Computer Science'}

# Using pop() (returns removed value)
removed_course = student.pop("course")
print(removed_course)  # Computer Science
print(student)
# {'name': 'Alice', 'age': 22}
```

---

**Summary Table**

| Operation | Syntax | Notes |
|---|---|---|
| Access | dict[key] or dict.get(key) | .get() avoids KeyError |
| Add | dict[new_key] = value | Adds if key doesn't exist |

| Operation | Syntax | Notes |
| --- | --- | --- |
| Update | dict[existing_key] = value | Changes existing value |
| Delete | del dict[key] or dict.pop(key) | .pop() returns removed value |

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

***Ans:***

**1. keys()**

- Returns a **view object** containing all the keys in the dictionary.
- Can be converted to a list if needed.

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


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

---

**2. values()**

- Returns a **view object** containing all the values in the dictionary.

```
print(student.values())        # dict_values(['Alice', 22, 'Computer Science'])
print(list(student.values()))   # ['Alice', 22, 'Computer Science']
```

---

**3. items()**

- Returns a **view object** of **(key, value)** pairs as tuples.
- Useful for looping through both keys and values.

```
print(student.items())
# dict_items([('name', 'Alice'), ('age', 22), ('course', 'Computer Science')])

# Looping through keys and values
for key, value in student.items():
    print(f"{key}: {value}")
```

**Output:**

```
name: Alice
age: 22
course: Computer Science
```

***17.Iterating over a dictionary using loops.***

***Ans:***

**1. Iterate Over Keys (Default)**

By default, looping over a dictionary gives you the **keys**.

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

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

**Output:**

```
name
age
course
```

---

**2. Iterate Over Values**

Use .values() to loop through values only.

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

**Output:**

```
Alice
22
Computer Science
```

---

**3. Iterate Over Key-Value Pairs**

Use .items() to get both key and value in each loop.

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

**Output:**

```
name: Alice
age: 22
course: Computer Science
```

---

**4. Using .keys() (Explicit keys)**

Same as default, but more explicit.

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

---

**5. Example Putting It All Together**

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

print("Keys:")
for key in student.keys():
    print(key)

print("\nValues:")
for value in student.values():
    print(value)

print("\nKey-Value Pairs:")
for key, value in student.items():
    print(f"{key}: {value}")
```

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

***Ans:***

**1. Using a Loop**

Suppose you have two lists: one for keys and one for values.

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

merged_dict = {}

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

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

---

**2. Using zip()**

zip() pairs elements from both lists together, making it very clean and Pythonic.

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

merged_dict = dict(zip(keys, values))
```

```
print(merged_dict)

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

---

**3. Handling Lists of Unequal Length**

- zip() stops at the shortest list length.
- If you want to handle missing keys or values, you can use itertools.zip_longest.

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

***Ans:***

**Method: Using a Dictionary to Count Characters**

**Step-by-step:**

1. Create an empty dictionary.
2. Loop over each character in the string.
3. For each character:
    - If the character is already a key in the dictionary, **increment** its count.
    - If not, **add** it with count 1.

---

**Example Code:**

```
text = "banana"


char_count = {}
```

```
for char in text:
    if char in char_count:
        char_count[char] += 1
    else:
        char_count[char] = 1

print(char_count)
```

**Output:**

```
{'b': 1, 'a': 3, 'n': 2}
```

---

**Alternative: Using dict.get() for cleaner code**

```
text = "banana"

char_count = {}

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

print(char_count)
```

---

**Bonus: Using collections.Counter (built-in)**

```
from collections import Counter

text = "banana"
```

```
char_count = Counter(text)


print(char_count)
```

Output:

Counter({'a': 3, 'n': 2, 'b': 1})

***20.Defining functions in Python.***

***Ans:***

**What is a Function?**

- A **function** is a reusable block of code that performs a specific task.
- You **define** a function once, then **call** (use) it whenever needed.
- Functions help make your code modular, readable, and organized.

---

**Basic Syntax to Define a Function**

```
def function_name(parameters):
    # Code block (function body)
    # Optional: return a value
```

- def keyword starts the function definition.
- function_name is the name you choose for your function.
- parameters (optional) are inputs the function accepts.
- Indentation is important — the code inside the function is indented.

---

**Example 1: Simple Function Without Parameters**

```
def greet():
    print("Hello, world!")

greet()  # Calling the function
```

Output:

```
Hello, world!
```

---

**Example 2: Function With Parameters**

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

greet_person("Alice")
greet_person("Bob")
```

Output:

```
Hello, Alice!
Hello, Bob!
```

---

**Example 3: Function With Return Value**

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

result = add(5, 3)
print(result)  # 8
```

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

***Ans:***

**1. Function Without Parameters and Without Return Value**

- Does **not** take any input.
- Does some action (like printing) but **doesn't return** any value.

```
def greet():
    print("Hello, world!")


greet()  # Output: Hello, world!
```

---

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

- Takes input values (parameters).
- Performs an action but **does not return** anything.

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


greet("Alice")  # Output: Hello, Alice!
```

---

**3. Function Without Parameters and With Return Value**

- Takes **no input**.
- Performs a task and **returns** a value.

```
def get_greeting():
```

```
    return "Hello, world!"

message = get_greeting()
print(message)  # Output: Hello, world!
```

---

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

- Takes input values.
- Processes them and **returns** a result.

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

result = add(5, 3)
print(result)
```

***22.Anonymous functions (lambda functions).***

***Ans:***

**What are Lambda Functions?**

- **Anonymous functions**: functions without a name.
- Created using the **lambda** keyword.
- Usually used for **small, simple functions**.
- Can take any number of arguments but **only one expression** (no statements).
- The expression is **implicitly returned**.

---

**Basic Syntax**

lambda arguments: expression

---

**Example 1: Simple Lambda Function**

square = lambda x: x * x

print(square(5))  # Output: 25

---

**Example 2: Lambda with Multiple Arguments**

add = lambda a, b: a + b

print(add(3, 4))  # Output: 7

---

**Example 3: Using Lambda Inline (Without Assigning)**

print((lambda x, y: x * y)(4, 5))  # Output: 20

---

**When to Use Lambda Functions?**

- When you need a **small function for a short time**.
- Commonly used with functions like map(), filter(), sorted().

Example with sorted():

points = [(2, 3), (1, 4), (4, 1)]

# Sort by y-coordinate using lambda as key

points_sorted = sorted(points, key=lambda point: point[1])

print(points_sorted)  # [(4, 1), (2, 3), (1, 4)]

***23.Introduction to Python modules and importing modules.***

***Ans:***

## What is a Python Module?

- A **module** is a file containing Python code — functions, classes, variables, etc.
- Modules help **organize code** and **reuse** it across different programs.
- Python comes with many **built-in modules** (like math, random, os), and you can create your own.

---

### How to Import Modules

#### 1. Import the whole module

```
import math


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

- Access module contents using module_name.item.

---

#### 2. Import specific functions or variables

```
from math import sqrt, pi


print(sqrt(25))  # 5.0
print(pi)        # 3.141592653589793
```

- You can import only what you need to keep namespace clean.

---

#### 3. Import module with an alias

```
import numpy as np
```

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

- Shortens module names for convenience.

---

**4. Import all contents (not recommended)**

```
from math import *

print(sin(0))  # 0.0
```

- Avoid this because it can **overwrite existing names** and reduce code clarity.

---

**How to Create and Use Your Own Module**

1. Create a Python file, e.g., mymodule.py:

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

2. Import it in another file or interactive shell:

```
import mymodule

mymodule.greet("Alice")
```

***24.Standard library modules: math, random.***

***Ans:***

**1. math Module**

Provides mathematical functions and constants.

**Common Functions and Constants:**

| Function/Constant | Description | Example |
|---|---|---|
| math.sqrt(x) | Square root of x | math.sqrt(16) → 4.0 |
| math.pow(x, y) | x raised to the power y | math.pow(2, 3) → 8.0 |
| math.sin(x) | Sine of x (x in radians) | math.sin(math.pi/2) → 1.0 |
| math.cos(x) | Cosine of x | math.cos(0) → 1.0 |
| math.factorial(n) | Factorial of n | math.factorial(5) → 120 |
| math.pi | Constant π (approx. 3.14159) | math.pi |
| math.e | Constant e (approx. 2.71828) | math.e |

**Example:**

```
import math

print(math.sqrt(25))       # 5.0
print(math.factorial(6))   # 720
print(math.sin(math.pi/2))  # 1.0
print(math.pi)          # 3.141592653589793
```

---

**2. random Module**

Used for generating random numbers and selecting random items.

**Common Functions:**

| Function | Description | Example |
|---|---|---|
| random.random() | Random float between 0.0 and 1.0 | random.random() → e.g. 0.3745 |
| random.randint(a, b) | Random integer between a and b (inclusive) | random.randint(1, 10) → e.g. 7 |
| random.choice(seq) | Random element from a sequence | random.choice(['apple', 'banana']) |
| random.shuffle(list) | Shuffles a list in place | random.shuffle(my_list) |
| random.sample(population, k) | Returns a list of k unique random elements | random.sample(range(10), 3) |

**Example:**

```
import random


print(random.random())          # e.g. 0.6394267984578837
```

```
print(random.randint(1, 100))   # e.g. 42
print(random.choice(['red', 'blue', 'green']))  # e.g. 'blue'


numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)  # e.g. [3, 5, 1, 4, 2]
```

***25.Creating custom modules.***

***Ans:***

**What is a Custom Module?**

- A custom module is simply a **Python file** (.py) that contains functions, classes, or variables.
- You can create reusable code and **import** it into other Python scripts.

---

**Step-by-Step: Creating and Using a Custom Module**

**1. Create a Python file for your module**

Create a file named, for example, mymodule.py.

```
# mymodule.py


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


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

```
pi = 3.14159
```

---

**2. Use your module in another Python script**

Create another file or open Python shell in the **same directory** and import your module:

```
import mymodule

mymodule.greet("Alice")        # Output: Hello, Alice!

result = mymodule.add(5, 3)
print(result)                  # Output: 8

print(mymodule.pi)             # Output: 3.14159
```

---

**3. Import specific items (optional)**

```
from mymodule import greet, pi

greet("Bob")                   # Output: Hello, Bob!
print(pi)                      # Output: 3.14159
```

---

**4. Module Search Path**

- Python looks for modules in the current directory, then in installed packages.

- If your module is in a different folder, you need to add that folder to sys.path or use packages.