★ Module 14 : Python – Collections, functions and Modules

➢ Accessing List

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

❖ Creating a List

- A list is an ordered collection of items, which can be of any data type.
- Lists are created by placing items inside square brackets `[]`, separated by commas.
- Example:

```
Python
  fruits = ['apple', 'banana', 'cherry']
  numbers = [1, 2, 3, 4, 5]
  mixed = [1, 'apple', 3.14, True]
  empty_list = []
```

❖ Accessing List Elements

- You access elements in a list by their index (position).
- Indexing starts at `0` for the first item.
- Negative indices access elements from the end, with `-1` being the last element.
- Example:

```
python
    fruits = ['apple', 'banana', 'cherry']

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

2.Indexing in Lists: Positive and Negative Indexing in Python
Ans:

- ❖ ***Positive Indexing:***
  - ❖ *Positive indexing starts from the beginning of the list.*
  - ❖ *The first element has index 0, the second has index 1, and so on.*
  - ❖ *You use positive integers to access elements from left to right.*

Example:
```python
    fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
    print(fruits[0])  # Outputs: apple (first element)
    print(fruits[2])  # Outputs: cherry (third element)
    print(fruits[4])  # Outputs: elderberry (fifth element)
```

- ❖ ***Negative Indexing:***
  - ❖ *Negative indexing starts from the end of the list.*
  - ❖ *The last element has index -1, the second last has index -2, etc.*
  - ❖ *You use negative integers to access elements from right to left.*

Example:
```python
    fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
    print(fruits[-1])  # Outputs: elderberry (last element)
    print(fruits[-3])  # Outputs: cherry (third last element)
    print(fruits[-5])  # Outputs: apple (fifth last element, the first)
```

❖ ***Important Notes:***
- *Positive indexes count forward starting with 0.*
- *Negative indexes count backward starting with -1.*
- *Trying an index out of range (e.g., `fruits[10]` or `fruits[-6]` for this 5-element list) raises an `IndexError`.*
- *Negative indexing is helpful to access elements near the end without knowing the list length.*

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

Ans:

Slicing a List: Accessing a Range of Elements in Python .Python list slicing allows you to access a sub-range (subset) of elements from a list using the syntax:

list[start:end:step]

- start: Starting index (inclusive) of the slice. Defaults to 0.
- end: Ending index (exclusive) of the slice. Defaults to the length of the list.
- step: Step size, selects every nth element. Defaults to 1.

Examples:

python

```python
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry', 'fig',
  'grape']

# Slice from index 1 to 4 (elements at 1, 2, 3)
     print(fruits[1:4])
  # Output: ['banana', 'cherry', 'date']
```

```python
# Slice from index 3 to the end
     print(fruits[3:])
 # Output: ['date', 'elderberry', 'fig', 'grape']
```

```python
# Slice from the start to index 5 (excluding 5)
     print(fruits[:5])
  # Output: ['apple', 'banana', 'cherry', 'date', 'elderberry']
```

```python
# Slice the whole list
     print(fruits[:])
# Output: ['apple', 'banana', 'cherry', 'date', 'elderberry', 'fig',
'grape']
```

```
# Slice with a step, every second element from index 0 to 6
    print(fruits[0:7:2])
# Output: ['apple', 'cherry', 'elderberry', 'grape']
```

```
# Slice the list in reverse order
    print(fruits[::-1])
# Output: ['grape', 'fig', 'elderberry', 'date', 'cherry', 'banana',
'apple']
```

- The slice `[1:4]` extracts elements from index 1 up to (but not including) 4.
- Omitting `start` or `end` defaults it to the start or end of the list.
- The `step` parameter controls the interval of elements selected.
- Using a negative `step` reverses the list slice.

Slicing is a powerful and concise way to access ranges of elements without loops.

➢ 2. List Operations

1.Common list operations: concatenation, repetition, membership.
Ans:

❖ **Common List Operations in Python**

1. Concatenation

- Combining two or more lists into one.
- Can be done using the + operator or the **`extend()`** method.

Examples:

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

# Using + operator (creates a new list)
    concatenated = list1 + list2
    print("Concatenated list using +:", concatenated)
```

```
# Using extend() method (modifies original list1)
list1.extend(list2)
print("List1 after extend:", list1)
```

2. Repetition

- Creating a new list by repeating elements a specified number of times.
- Use the `*` operator.

Example:

```
python
list3 = [1, 2, 3]

# Repeat list 3 times
repeated = list3 * 3
print("Repeated list:", repeated)
```

3. Membership

- Checking if an element exists in a list using the `in` keyword.
- Returns `True` or `False`.

Example:

```
python
fruits = ['apple', 'banana', 'cherry']

print('banana' in fruits)  # True
print('mango' in fruits)   # False
```

❖ Summary Table

Operation	Syntax	Description
Concatenation	`list1 + list2` or `list1.extend(list2)`	Combine two lists

Repetition	`list * n`	Repeat list elements `n` times
Membership	`element in list`	Check if element exists in list

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

Ans:

❖ **List Methods**

1. `append(element)`

- Adds an element to the end of the list.

```
python
a = [1, 2, 3]
a.append(4)
print(a)  # Output: [1, 2, 3, 4]
```

---

2. `copy()`

- Returns a shallow copy of the list.

```
python
a = [1, 2, 3]
b = a.copy()
print(b)  # Output: [1, 2, 3]
```

---

3. `clear()`

- Removes all elements from the list.

```
python
a = [1, 2, 3]
a.clear()
print(a)  # Output: []
```

---

4. `count(element)`

- Returns the number of occurrences of a specified element.

```
python
a = [1, 2, 3, 2]
print(a.count(2))  # Output: 2
```

---

5. `extend(iterable)`

- Adds elements from another iterable to the end of the list.

```
python
a = [1, 2]
a.extend([3, 4])
print(a)  # Output: [1, 2, 3, 4]
```

---

6. `index(element)`

- Returns the index of the first occurrence of a specified element.

```
python
a = [1, 2, 3]
print(a.index(2))  # Output: 1
```

---

7. `insert(index, element)`

- Inserts an element at the given index, shifting later elements.

```
python
a = [1, 3]
a.insert(1, 2)
print(a)  # Output: [1, 2, 3]
```

8. `pop(index=-1)`

- Removes and returns an element at the given index (default is last).

```
python
a = [1, 2, 3]
last = a.pop()
print(last)  # Output: 3
print(a)     # Output: [1, 2]
```

---

9. `remove(element)`

- Removes the first occurrence of a specified element.

```
python
a = [1, 2, 3, 2]
a.remove(2)
print(a)  # Output: [1, 3, 2]
```

---

10. `reverse()`

- Reverses the list in-place.

```
python
a = [1, 2, 3]
a.reverse()
print(a)  # Output: [3, 2, 1]
```

---

11. `sort(reverse=False)`

- Sorts the list in ascending order by default; set `reverse=True` for descending.

```
python
a = [3, 1, 2]
a.sort()
print(a)  # Output: [1, 2, 3]

# For descending sort
a.sort(reverse=True)
print(a)  # Output: [3, 2, 1]
```

➢ 3. Working with Lists

1.Iterating over a list using loops.

Ans:

1. Using a for loop to iterate over elements directly

- This is the simplest and most common method to loop through all items:

```
python
```

```
    fruits = ['apple', 'banana', 'cherry']
    for fruit in fruits:
        print(fruit)
```

Output:

```
apple
banana
Cherry
```

2. Using a for loop with index (using `range` and `len`)

- You can loop through the list indices and access elements by index:

python

```
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
    print(fruits[i])
```

---

3. Using a while loop

- Loop using an index counter manually:

python

```
fruits = ['apple', 'banana', 'cherry']
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

---

4. Using enumerate()

- To get both the index and the element during looping:

python

```
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
```

5. Using list comprehension (for concise looping with actions)

python

```
    fruits = ['apple', 'banana', 'cherry']
    [print(fruit) for fruit in fruits]
```

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

Ans:

➢ **Using `sort()` (In-place Sort)**

- Modifies the original list, sorting the elements in ascending order.
- Returns `None`, so you don't assign its result to a variable.

Python

```
   numbers = [5, 2, 9, 1]
   numbers.sort()  # Sorts the list in place
   print(numbers)  # Output: [1, 2, 5, 9]
```

➢ **Use `reverse=True` for descending order:**

python

```
   numbers.sort(reverse=True)
   print(numbers)  # Output: [9, 5, 2, 1]
```

➢ **Using `sorted()` (Returns New List)**

- Returns a new sorted list and leaves the original unchanged.
- Works with any iterable (not just lists).

python

```
   numbers = [5, 2, 9, 1]
   sorted_numbers = sorted(numbers)  # Creates sorted copy
   print(sorted_numbers)  # Output: [1, 2, 5, 9]
   print(numbers)         # Output: [5, 2, 9, 1] (original unchanged)
```

➢ **For descending order:**

python

```
   desc_sorted = sorted(numbers, reverse=True)
   print(desc_sorted)  # Output: [9, 5, 2, 1]
```

➢ **Using `reverse()` (Reverses In-place)**

- Reverses the elements of the list in place (does NOT sort).
- Original list is modified.

python

```
numbers = [5, 2, 9, 1]
numbers.reverse()
print(numbers)  # Output: [1, 9, 2, 5]
```

3.Basic List manipulations:addition ,deletion ,updating and slicing.

➢ **Ans : Addition to List**

- You can add elements to a list using:
- `append()` to add a single element at the end.
- `extend()` to add multiple elements.
- `insert()` to add an element at a specific position.

python

```
lst = [1, 2, 3]
lst.append(4)          # [1, 2, 3, 4]
lst.extend([5, 6])     # [1, 2, 3, 4, 5, 6]
lst.insert(2, 9)       # [1, 2, 9, 3, 4, 5, 6]
```

➢ **Deletion from List**

➢ Elements can be removed using:

- `remove()` to delete a specific element by value.
- `pop()` to remove an element by index (default last).
- `del` keyword to delete element(s) by index or slice.

---

python

```
lst = [1, 2, 9, 3, 4, 5, 6]
lst.remove(9)     # [1, 2, 3, 4, 5, 6]
lst.pop(3)        # Removes element at index 3 -> [1, 2, 3, 5, 6]
del lst[0]        # Deletes element at index 0 -> [2, 3, 5, 6]
del lst[1:3]      # Deletes elements from index 1 to 2 -> [2, 6]
```

➢ **Updating List Elements.**

- You can update the value of an element by assigning a new value to a specific index:

python

```
lst = [2, 3, 5, 6]
lst[2] = 10

#output :  [2, 3, 10, 6]
```

➢ **Slicing Lists**

- Extract parts of a list using slicing with the syntax `list[start:stop:step]`:

python

```
lst = [1, 2, 3, 4, 5, 6]
print(lst[1:4])    # [2, 3, 4] (index 1 to 3)
print(lst[:3])     # [1, 2, 3] (start to index 2)
print(lst[3:])     # [4, 5, 6] (index 3 to end)
print(lst[::2])    # [1, 3, 5] (every second element)
```

➢ 4. Tuple

1.Introduction to tuples, immutability.

Ans:

A tuple in Python is an ordered collection of elements, similar to a list, but with the key difference of being immutable. This means that once a tuple is created, its elements cannot be changed, added, or removed. Tuples are defined by enclosing values in parentheses `()` separated by commas:

```
Python

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

❖ **Key Characteristics of Tuples:**

- Ordered: Elements have a defined order, and this order is maintained.
- Immutable: After creation, tuples cannot be modified in any way (no addition, deletion, or updating of elements).
- Indexed: Elements can be accessed by their index, starting at zero.
- Allow Duplicates: Tuples can contain multiple occurrences of the same value.
- Heterogeneous: Tuples can store elements of different data types.

Example:

```
python

    t = (1, "apple", 3.14, True)
    print(t[1])       # Output: apple
    # t[1] = "banana"
    # This will raise a TypeError because tuples are immutable
```

- **Creating a Single-Element Tuple:**

  A tuple with one element requires a trailing comma to differentiate it from a simple value inside parentheses:

```
Python

single = (42,)
```

- Tuples are often used to group related but different types of data together and are useful when the data should remain constant, such as coordinates or fixed records. They are also more memory-efficient than lists and can be used as dictionary keys because of their immutability.

2.Creating and accessing elements in a tuple.

Ans:

❖ **Creating Tuples**

- Use parentheses `()` with comma-separated values:

```
python

    t1 = (1, 2, 3)
    t2 = ("apple", "banana", "cherry")
```

- A tuple with one element requires a trailing comma to distinguish it from a regular value:

```
python

    single = (42,)  # single-element tuple
    not_tuple = (42)  # this is just an integer
```

- Tuples can hold mixed data types, including nested tuples or other objects:

```
python

    mixed = (1, "hello", 3.14, (5, 6))
```

- Alternatively, use the `tuple()` constructor with an iterable:

```
python

    t3 = tuple([1, 2, 3])  # from list
    t4 = tuple("abc")      # from string, becomes ('a', 'b', 'c')
```

➢ **Accessing Elements**

- Access elements by index (starting at 0):

```
python

    print(t2[1])  # Output: banana
```

- Negative indices access from the end (-1 is last element):

```
python

    print(t2[-1])  # Output: cherry
```

- Tuples support slicing like lists:

```
python

    print(t1[1:3])  # Output: (2, 3)
```

- Unpacking allows assigning tuple elements to variables in one statement:

```
Python

    t1 = (1, 2, 3)
    a, b, c = t1
    print(a, b, c)  # Output: 1 2 3
```

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

Ans:

➢ **Concatenation**

- The most common method is using the + operator, which joins two or more tuples into a new one:

```
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b')
result = tuple1 + tuple2  # (1, 2, 3, 'a', 'b')
```

➢ **Repetition**

- You can also use the `*` operator for repetition:

```
tuple1 = ('hi',)

result = tuple1 * 3

#Output :  ('hi', 'hi', 'hi')
```

➢ **Membership Testing**

- To check if an element exists in a tuple, use the `in` operator:

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

print(2 in t)  # True

print(5 in t)  # False
```

➢ 5. Accessing Tuples

1.Accessing tuple elements using positive and negative indexing.

Ans:

➢ **Positive Indexing**

- Starts from 0 for the first element.
- To access an element, specify the index inside square brackets.
- Example:

```
python

    t = ("apple", "banana", "cherry", "date")
    print(t[0])  # Output: apple (first element)
    print(t[2])  # Output: cherry (third element)
```

➢ **Negative Indexing**

- Starts from -1 for the last element, -2 for the second-last, and so forth.
- Useful for accessing elements from the end of the tuple without knowing its length.

Example:

python

```
    t = ("apple", "banana", "cherry", "date")
    print(t[-1])  # Output: date (last element)
    print(t[-3])  # Output: banana (third from last)
```

Key Points

- Positive indices count from the start (0 to length-1).
- Negative indices count backward from the end (-1 to -length).
- Indexing beyond the range will cause `IndexError`.

This dual indexing system makes tuples flexible for element access from either direction.

2.Slicing a tuple to access ranges of elements.

Ans:

Slicing a tuple in Python lets you access a range of elements by specifying a start, stop, and optional step, producing a new tuple.

➢ **Syntax**

python

```
    Tuple[start:stop:step]
```

- start: Index where the slice begins (inclusive). Default is 0.
- stop: Index where slice ends (exclusive). Default is end of tuple.
- step: Interval between elements. Default is 1.
- **Examples:**

python

```
t = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

print(t[2:6])    # (2, 3, 4, 5)  – elements from index 2 to 5
print(t[:4])     # (0, 1, 2, 3)  – from start to index 3
print(t[5:])     # (5, 6, 7, 8, 9) – from index 5 to end
print(t[:])      # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) – entire tuple
print(t[1:9:2])  # (1, 3, 5, 7) – elements from index 1 to 8 with step 2
```

➢ **Negative Indexing with slicing**

```python
print(t[-5:-2])  # (5, 6, 7) – from 5th last to 3rd last index
print(t[::-1])   # (9, 8, 7, ..., 0) – reversed tuple
```

- Slicing creates a new tuple and does not modify the original. This technique is handy to extract subsets or reversed sequences efficiently.

➢ 6. Dictionary

1. Introduction to dictionaries: key-value pairs.

Ans:

A dictionary in Python is a collection of key-value pairs where each key is unique and maps to a corresponding value. It is an unordered, mutable data structure used for storing data efficiently by associating keys to values.

Key Characteristics:

- Keys must be immutable types (like strings, numbers, or tuples).
- Values can be of any data type, including other dictionaries.
- Keys are unique; duplicate keys overwrite existing values.
- Dictionaries are mutable, meaning you can add, update, or remove key-value pairs after creation.
- From Python 3.7 onwards, dictionaries maintain the insertion order.

➢ **Creating a Dictionary**

- You can create a dictionary using curly braces `{}` with key-value pairs separated by colons:

Python

```
my_dict = {

          "name": "Alice",
          "age": 25,
          "city": "New York"
      }
```

- Alternatively, create it using the `dict()` constructor:

```
python

my_dict = dict(name="Alice", age=25, city="New York")
```

➢ **Accessing Values**

- Access values by their keys using square brackets:

```
python

print(my_dict["name"])  # Output: Alice
```

❖ **Example of Key-Value Pair**

- Key: `"name"`
- Value: `"Alice"`
- **Note** : Thus, a dictionary stores data like this: `"key": value` pairs, allowing fast lookup by keys.

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

Ans:

❖ Accessing Elements

- Use square brackets with the key: `value = dict[key]`. Raises `KeyError` if the key is missing.
- Use the `get()` method: `value = dict.get(key)`; returns `None` or a default value if key is missing.

Example:

```
Python
```

```
car = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(car["model"])         # Mustang
print(car.get("model"))    # Mustang
print(car.get("color"))    # None
print(car.get("color", "Unknown"))  # Unknown
```

❖ **Adding Elements**

- Assign a value to a new key:
  `python`

```
car["color"] = "white"
print(car)  # Now includes 'color': 'white'
```

❖ **Updating Elements**

- Assign a new value to an existing key:
  `python`

```
car["year"] = 2020
print(car["year"])  # 2020
```

❖ **Deleting Elements**

- Use `del` keyword:
  `python`

```
del car["model"]
print(car)  # 'model' key deleted
```

- **Use `pop()` method to remove and return a value:**
  `python`

```
color = car.pop("color")
print(color)  # white
print(car)    # 'color' removed
```

3.Dictionary methods like `keys()`, `values()`, and `items()`.

Ans:

➢ **keys()**

- Returns a view object containing all the keys in the dictionary.
- The view updates dynamically as the dictionary changes.
- Useful for iterating over just the keys.

Example:

```
python
```

```
d = {"name": "Alice", "age": 25, "city": "New York"}

print(d.keys())  # dict_keys(['name', 'age', 'city'])

for key in d.keys():

    print(key)
```

➢ **values()**

- Returns a view object containing all values in the dictionary.
- Also dynamically updated on dictionary modification.
- Helpful when you only need the values.

Example:

```
python
```

```
print(d.values())  # dict_values(['Alice', 25, 'New York'])

for value in d.values():

    print(value)
```

➢ **items()**

- Returns a view object with all key-value pairs as tuples `(key, value)`.
- Useful for iterating over dictionary entries simultaneously.

Example:

python

```
print(d.items())

# dict_items([('name', 'Alice'), ('age', 25), ('city', 'New
York')])

for key, value in d.items():

    print(f"Key: {key}, Value: {value}")
```

➢ **Note on View Objects**

- These views reflect the current dictionary state. If you modify the dictionary, the view updates automatically.
- You can convert them to lists if needed:

python

```
list_keys = list(d.keys())

list_values = list(d.values())

list_items = list(d.items())
```

➢ 7. Working with Dictionaries

1.Iterating over a dictionary using loops.

Ans:

Iterating over a dictionary in Python can be done in several ways to access keys, values, or both key-value pairs using loops:

➢ **Iterate Over Keys**

- By default, iterating a dictionary returns its keys:

python

```
d = {"name": "Alice", "age": 25, "city": "New York"}
```

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

➢ **Or explicitly using `keys()`:**

python

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

➢ **Iterate Over Values**

- Use the `values()` method to loop through all values:

python

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

➢ **Iterate Over Key-Value Pairs**

- Use `items()` to get key and value together as tuples and iterate:

python

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

➢ **Additional Techniques**

- Use `enumerate()` with keys or values to get index and element:

python

```
       for index, key in enumerate(d):
           print(index, key)
```

➢ **Iterate in sorted order using `sorted()`:**

python

```
        for key in sorted(d):
            print(key, d[key])
```

- These iteration methods provide flexibility in accessing dictionary contents efficiently with loops.

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

Ans:

1. Using `zip()` and `dict()`

- The `zip()` function pairs elements from the two lists by index, and `dict()` converts these pairs into a dictionary:

python

```python
keys = ['a', 'b', 'c']
values = [1, 2, 3]
merged_dict = dict(zip(keys, values))
print(merged_dict)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

- If the lists have different lengths, `zip()` stops at the shortest.

2. Using a Loop with `zip()`

- Manually iterate through both lists simultaneously and add key-value pairs to a new dictionary:

python

```python
keys = ['a', 'b', 'c']
values = [1, 2, 3]
merged_dict = {}
for k, v in zip(keys, values):
    merged_dict[k] = v
print(merged_dict)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

Notes:

- Both ways create a new dictionary by pairing corresponding elements.
- Use `collections.defaultdict(list)` with loops if you want to group multiple

values under same keys (advanced use).

- These are effective, simple methods to merge two lists into a dictionary.Two common ways to merge two lists into a dictionary in Python are:

❖ **Using zip() and dict()**

- The zip() function pairs elements from the two lists by their index, and dict() converts these pairs into a dictionary.

```
python

    keys = ['a', 'b', 'c']
    values = [1, 2, 3]
    merged_dict = dict(zip(keys, values))
    print(merged_dict)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

➢ **Using a loop with zip()**

- You can manually iterate over the zipped pairs and add them to a dictionary.

```
python

    keys = ['a', 'b', 'c']
    values = [1, 2, 3]
    merged_dict = {}
    for k, v in zip(keys, values):
        merged_dict[k] = v
    print(merged_dict)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

❖ Both methods work well when lists are of equal length. If lists have different lengths, zip() pairs elements up to the shortest list's length. These techniques give an efficient way to merge lists into a key-value mapping dictionary.

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

Ans: Counting occurrences of characters in a string using a Python dictionary involves iterating over each character and using the dictionary to store the count for each character as key-value pairs.

Example code:

python

```
text = "hello world"
char_count = {}

for char in text:
    if char in char_count:
        char_count[char] += 1  #Increment count if char already in dict
    else:
        char_count[char] = 1   # Initialize count for new character
print(char_count)
```

➢ **Explanation:**

- Initialize an empty dictionary `char_count`.
- For each character in the string, check if it is already a key in the dictionary.
- If yes, increment its count.
- If no, add the character as a key with count 1.
- After the loop, `char_count` contains each character as key and the number of occurrences as value.
- **Output for `"hello world"`:**

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

- This approach efficiently counts characters in one pass and is easy to understand and implement. There are also advanced ways using `collections.Counter`, but this dictionary method is fundamental and versatile

➢ 8. Functions

1. Defining functions in Python.

Ans:

In Python, a function is defined using the `def` keyword followed by the function name, parentheses for optional parameters, and a colon. The function body is indented and contains the code to be executed when the function is called.

➢ **Basic Syntax**

```
def function_name(parameters):
    """Optional docstring describing the function"""
    # Function body
    statements
    return value  # Optional
```

➢ **Example of a simple function:**

```
def greet():
    print("Hello from a function")

greet()  # Calling the function
```

➢ **Function with parameters and return value:**

```
    def add_numbers(a, b):
        result = a + b
        return result

sum = add_numbers(3, 5)
print(sum)  # Output: 8
```

❖ **Key points:**

- Functions help organize and reuse code.
- Parameters (also called arguments) are input values you pass.
- The `return` statement sends a result back to the caller.

- The function body must be indented.
- Parentheses are mandatory even if no parameters exist.
- Functions can be much more complex, but this is the foundation for creating and using them in Python.

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

Ans:

In Python, functions can be categorized based on whether they have parameters and whether they return values. Here are the four main types:

1. Function without parameters and without return value

- Takes no inputs and does not return anything.
- Usually performs an action like printing.

```
python

    def greet():
        print("Hello!")

    greet()  # Output: Hello!
```

2. Function with parameters but without return value

- Takes input(s) but does not return any output.
- Performs an action using parameters.

```
python

    def print_sum(a, b):
        print("Sum is:", a + b)

    print_sum(3, 5)  # Output: Sum is: 8
```

3. Function without parameters but with return value

- Takes no inputs but returns a value.

```
python
```

```
def get_pi():
    return 3.14159

print(get_pi())  # Output: 3.14159
```

4. Function with parameters and with return value

- Takes input(s) and returns a result.

python

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

result = add(4, 7)
print(result)  # Output: 11
```

❖ Summary Table:

Function Type	Parameters	Returns Value	Example Purpose
Without parameters/without return	No	No	Print messages or side effects
With parameters/without return	Yes	No	Perform action with inputs
Without parameters/with return	No	Yes	Return fixed or computed value
With parameters/with return	Yes	Yes	Compute and return result

3.Anonymous functions (lambda functions).

Ans:

Anonymous functions in Python, commonly known as lambda functions, are small, unnamed functions defined with the `lambda` keyword. They are typically used for short, simple operations and can take any number of arguments but have only a single expression, which is evaluated and returned.

➢ **Syntax:**

Python

```
lambda arguments: expression
```

- `arguments`: comma-separated input parameters.
- `expression`: single expression computed and returned.

Examples:

1. Lambda that adds 10 to its input:

python

```
add_10 = lambda x: x + 10
print(add_10(5))  # Output: 15
```

2. Lambda with multiple arguments:

python

```
multiply = lambda a, b: a * b
print(multiply(4, 5))  # Output: 20
```

3. Using lambda with `map()` to double a list:

python

```
numbers = [1, 2, 3, 4]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)  # Output: [2, 4, 6, 8]
```

---

4. Using lambda with `filter()` to get even numbers:
   python

```
      even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
      print(even_numbers)  # Output: [2, 4]
```

5. Returning a lambda function from another function:

   python

```
    def multiplier(n):
          return lambda x: x * n

   doubler = multiplier(2)
   tripler = multiplier(3)

   print(doubler(10))  # Output: 20
   print(tripler(10))  # Output: 30
```

➢ **Key Points:**

- Lambda functions are anonymous (no name unless assigned).
- They are useful for simple, short-lived functions.
- Lambda can be passed as arguments to higher-order functions like `map()`, `filter()`, and `sorted()`.
- Limited to one expression (no multi-step computations).

➢ 9. Modules

1. Introduction to Python modules and importing modules.

Ans:

A Python module is a file containing Python code—such as functions, classes, variables, or runnable statements—stored in a `.py` file. Modules allow you to organize and reuse code by grouping related functionalities together, making programs easier to maintain and modular.

➢ **Key Points about Modules:**

- A module is essentially a Python file with a `.py` extension.

- It can contain functions, classes, variables, and runnable code.
- Using modules helps break large programs into smaller, manageable parts.
- Modules promote code reuse and logical organization.
- Python has many built-in modules (like `math`, `os`, `sys`) and you can also create your own.

➢ **Importing Modules**

- Use the `import` statement to include the module in your script.
- Access module members (functions, variables) with **`module_name.member_name`**.

Example:

- Create a module file `mymodule.py`:

```
python

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

    pi = 3.14159
```

- Import and use it in another script:

```
python

   import mymodule
   mymodule.greet("Alice")  # Hello, Alice!
   print(mymodule.pi)       # 3.14159
```

`2. Standard library modules: math, random.`

`Ans:`

Python's standard library includes many useful modules, two important ones being `math` and `random`.

➢ **Math Module**

- The `math` module provides access to mathematical functions and constants widely used in scientific and engineering computations.

➢ **Key functions and constants:**

- `math.sqrt(x)`: Returns the square root of `x`.
- `math.pow(x, y)`: Returns (x raised to power y).
- `math.factorial(n)`: Returns factorial of `n`.
- `math.ceil(x)`: Rounds `x` up to nearest integer.
- `math.floor(x)`: Rounds `x` down to nearest integer.
- `math.gcd(a, b)`: Greatest common divisor of `a` and `b`.
- `math.sin(x), math.cos(x), math.tan(x)`: Trigonometric functions (input in radians).
- Constants like `math.pi` (π ≈ 3.14159) and `math.e` (Euler's number).

➢ **Example:**

python

```
import math
print(math.sqrt(16))       # 4.0
print(math.factorial(5))   # 120
print(math.pi)             # 3.141592653589793
```

➢ **random Module**

- The `random` module provides functions to generate random numbers and perform random selections.

➢ **Key functions:**

- `random.random()`: Returns a float between 0.0 and 1.0.
- `random.randint(a, b)`: Returns a random integer between `a` and `b`, inclusive.
- `random.choice(seq)`: Returns a random element from the non-empty sequence `seq`.
- `random.shuffle(lst)`: Randomly reorders elements in the list `lst`.
- `random.sample(population, k)`: Returns a list of `k` unique elements sampled from `population`.

➢ **Example:**

python

```
import random
print(random.random())          # e.g., 0.37444887175646646
print(random.randint(1, 10))   # Random int between 1 and 10
print(random.choice(['a', 'b', 'c']))  # Randomly 'a', 'b', or 'c'
```

3. Creating Custom Module.

Ans:

Creating custom modules in Python is straightforward. A module is simply a `.py` file containing Python code—functions, classes, variables—that you want to reuse.

- **Steps to Create and Use a Custom Module**

1. Create a Python file with your module content, for example `mymodule.py`:

   Python

   ```
   # mymodule.py


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


         pi = 3.14159
   ```

2. Import the module in another Python script or interactive session:

   python

   ```
   import mymodule
   mymodule.greet("Alice")  # Output: Hello, Alice!
   print(mymodule.pi)       # Output: 3.14159
   ```

3. Alternatively, import specific functions or variables:

   python

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

4. You can also rename the module during import:

   python

   ```
   import mymodule as mm
   ```

```
    mm.greet("Charlie")      # Output: Hello, Charlie!
```

Important Notes

- The module file (`mymodule.py`) should be in the same directory as your script or in a directory included in Python’s module search path.
- Modules can contain any valid Python code: variables, functions, classes, statements.
- Use custom modules to organize and reuse your code conveniently.