❖ Python – Collections, functions and Modules

(1) Accessing List

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

- A **list** is an ordered collection of items. Lists can hold items of any type (numbers, strings, booleans, other lists), and you can change them after creating them.

- **Creating a List:**

  - Lists in Python are created using square brackets [], with items separated by commas [1, 2].

    - **Example:**

```
# An empty list
empty_list = []

# A list of integers
numbers = [1, 2, 3, 4, 5]

# A list of strings
fruits = ["apple", "banana", "cherry"]
```

 # A list with mixed data types

mixed_list = [1, "hello", 3.14, True]

 # Creating a list using the list() constructor

another_list = list(("dog", "cat", "bird"))

- **Accessing Elements in a List:**

  - You access individual elements in a list using **indexing**, where the first element is at index `0`, the second at index `1`, and so on.

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

- **Positive Indexing**

  - Access elements from the beginning of the list to the end.

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

    # Access the first element
    first_item = fruits[0] # first_item is "apple"

    # Access the third element
    third_item = fruits[2] # third_item is "cherry"

- **Negative Indexing**

  - Access elements from the end of the list to the beginning. The last element is at index -1, the second to last at -2, etc.

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

# Access the last element
last_item = fruits[-1]  # last_item is "elderberry"

# Access the second to last element
second_to_last = fruits[-2] # second_to_last is "date"
```

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

- You can access a range of elements using the slicing.

- **syntax:**

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

  - **start:** The index where the slice begins (inclusive).
  - **stop:** The index where the slice ends (exclusive).
  - **step:** The interval between items (defaults to 1).

- **Example:**

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get elements from index 2 up to (but not including) index 5
slice1 = numbers[2:5]   # slice1 is [2, 3, 4]

# Get elements from the beginning up to index 3
slice2 = numbers[:4]    # slice2 is [0, 1, 2, 3]

# Get elements from index 4 to the end
slice3 = numbers[4:]    # slice3 is [4, 5, 6, 7, 8, 9]

# Get a copy of the entire list using slicing
copy_list = numbers[:]  # copy_list is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get every second element
every_other = numbers[::2] # every_other is [0, 2, 4, 6, 8]
```

(2) List Operation

Q-1) Common list operations: concatenation, repetition, membership.

- In Python, **concatenation**, **repetition**, and **membership** are common operations for lists. They are performed using specific operators:

  - **Concatenation** uses the + operator.
  - **Repetition** uses the * operator.
  - **Membership** uses the in and not in operators.

- **Concatenation**

  - The + operator is used to combine two or more lists into a single, new list. The original lists remain unchanged.

  - **syntax:**

    list1 + list2

  - **Example:**

    list_a = [10, 20]

    list_b = [30, 40]

    combined_list = list_a + list_b

    print(combined_list)

    # Output: [10, 20, 30, 40]

- **Repetition**

  - The `*` operator is used to create multiple copies of a list and join them together into a new list. The number of copies is specified by an integer operand.

  - **syntax:**

    list * n

  - **Example:**

    original_list = ['apple', 'banana']

    repeated_list = original_list * 3

    print(repeated_list)

    # Output: ['apple', 'banana', 'apple', 'banana', 'apple', 'banana']

- **Membership**

  - Membership operators check if an element exists within a list and return a boolean value (`True` or `False`).

  - **in**: Returns True if the element is found in the list, False otherwise.

  - **not in**: Returns True if the element is not found in the list, False otherwise.

- **syntax:**

  `element in list` or `element not in list`

- **Example:**

  fruits = ['orange', 'grape', 'kiwi']

  # Check for existence

  print('grape' in fruits)

  # Output: True

  # Check for non-existence

  print('mango' not in fruits)

  # Output: True

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

- The `append()`, `insert()`, `remove()`, and `pop()` methods are fundamental for adding and removing elements from a list.

  - **append()**

    - The `append()` method adds a single item to the end of a list. This method modifies the original list in place and does not return a value.

- **syntax:**

list_name.append(item)

- **Example:**

fruits = ['apple', 'banana']

fruits.append('cherry')

fruits is now ['apple', 'banana', 'cherry']

- **insert()**

  - The `insert()` method adds an item at a specific position (index) in the list. The first argument is the index where you want to insert the item, and the second argument is the item itself.

    - **syntax:**

      list_name.insert(index, item)

    - **Example:**

      fruits = ['apple', 'cherry']

      fruits.insert(1, 'banana')

      # fruits is now ['apple', 'banana', 'cherry']

- **remove()**

  - The `remove()` method removes the first occurrence of a specified value from the list. If the item is not found, a `ValueError` is raised.

    - **syntax:**

      list_name.remove(value)

    - **Example:**

      ```
      fruits = ['apple', 'banana', 'cherry']
      fruits.remove('banana')
      # fruits is now ['apple', 'cherry']
      ```

- **pop()**

  - The `pop()` method removes an item at specified index and returns the removed item. If no index is specified, it removes and returns the last item in the list.

    - **syntax:**

      - list_name.pop(index) (removes item at specified index)
      - list_name.pop() (removes the last item)

- **Example:**

```
fruits = ['apple', 'banana', 'cherry']
removed_fruit = fruits.pop(1)
# removed_fruit is 'banana'
# fruits is now ['apple', 'cherry']

last_fruit = fruits.pop()
# last_fruit is 'cherry'
# fruits is now ['apple']
```

(3) Working with Lists

Q-1) Iterating over a list using loops.

- In Python, you can iterate over a list using `for` loops or `while` loops. The most common and "Pythonic" way is using a simple `for` loop, which iterates directly over the elements of the list.

- **`for` Loop**

  - This method iterates directly through the items in the list without needing to manage index counters manually.

    - **Example:**

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

      - **Output:**

```
apple
banana
cherry
```

- **Using `enumerate()`**

  - Use the `enumerate()` function when you need both the element's value and its index during iteration. This is a clean alternative to using `range(len(list))`.

- **Example:**

```
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
```

- **Using `range()` and `len()`**

  - This approach is similar to loops in other programming languages where you iterate over a sequence of index numbers.
    The `range(len(list))` generates indices from 0 up to the length of the list minus one.

    - **Example:**

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

      - **Output:**

```
apple
banana
cherry
```

- **Using a `while` Loop**

  - A `while` loop can be used when the number of iterations is not known in advance, or if you prefer a condition-based loop. You must manually manage the index counter and ensure it increments to avoid an infinite loop.

    - **Example:**

```
fruits = ["apple", "banana", "cherry"]
i = 0
while i < len(fruits):
  print(fruits[i])
  i += 1 # Increment the counter
```

      - **Output:**

```
apple
banana
cherry
```

Q-2) Sorting and reversing a list using sort(), sorted(), and reverse().

- Sorting and reversing lists in Python can be done using the `list.sort()`, `sorted()`, and `list.reverse()` methods/functions.

- **`list.sort()` and `list.reverse()`**

  - These methods modify the list directly (in-place) and return `None`. They are ideal when you don't need to preserve the original list.

    - **Example:**

      ```
      my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

      # Sorting in ascending order (default)
      my_list.sort()
      # my_list is now [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

      # Reversing the order
      my_list.reverse()
      # my_list is now [9, 6, 5, 5, 4, 3, 3, 2, 1, 1]

      # Sorting in descending order using the reverse parameter
      my_list.sort(reverse=True)
      # my_list is now [9, 6, 5, 5, 4, 3, 3, 2, 1, 1]
      ```

- **`sorted()`**

  - The built-in `sorted()` function creates a new sorted list from any iterable (lists, tuples, strings, etc.), leaving the original iterable unchanged.

    - **Example:**

      ```
      my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

      # Create a new list sorted in ascending order
      new_sorted_list = sorted(my_list)
      # new_sorted_list is [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
      # my_list remains unchanged
      ```

```
# Create a new list sorted in descending order
new_desc_list = sorted(my_list, reverse=True)
# new_desc_list is [9, 6, 5, 5, 4, 3, 3, 2, 1, 1]
```

Q-3) Basic list manipulations: addition, deletion, updating, and slicing.

- In Python, lists are versatile and support several built-in methods and operations for common tasks:

(1) Addition (Adding elements)

- There are several ways to add elements to a list:
  - **append()** : Adds a single element to the end of a list.
    - **Example:**

```
my_list = [1, 2, 3]
my_list.append(4)
# Result: [1, 2, 3, 4]
```

  - **insert() :** Inserts an element at a specific index. The first argument is the index, and the second is the value.
    - **Example:**

```
my_list = [1, 3, 4]
my_list.insert(1, 2)  # Insert 2 at index 1
# Result: [1, 2, 3, 4]
```

- **extend()** : Appends elements from another iterable (like another list) to the current list.

    - **Example:**

      ```
      my_list = [1, 2]
      my_list.extend([3, 4])
      # Result: [1, 2, 3, 4]
      ```

(2) Deletion (Removing elements)

- Methods for removing elements vary depending on whether you know the element's value or its index:

    - **remove()** : Removes the first occurrence of a specified value.

        - **Example:**

          ```
          my_list = [1, 2, 3, 2]
          my_list.remove(2)
          # Result: [1, 3, 2]
          ```

    - **pop()** : Removes and returns the element at a specific index. If no index is specified, it removes and returns the last element.

        - **Example:**

          ```
          my_list = [1, 2, 3]
          removed_element = my_list.pop(1) # Removes element at index 1 (value 2)
          # my_list is now: [1, 3]
          # removed_element is: 2
          ```

- **del** : Removes an element at a specific index or a slice of elements.
    - **Example:**

```
my_list = [1, 2, 3, 4]
del my_list[1]  # Removes element at index 1 (value 2)
# Result: [1, 3, 4]
del my_list[1:3] # Removes elements at index 1 and 2
# Result: [1]
```

(3) Updating (Modifying elements)

- You can change an element by accessing it via its index and assigning a new value:
    - **Index assignment:**
        - **Example:**

```
my_list = [1, 2, 3]
my_list[1] = 5 # Change element at index 1 (value 2) to 5
# Result: [1, 5, 3]
```

    - **Slice assignment**: You can replace a range of elements with a new list (the number of items does not need to match).
        - **Example:**

```
my_list = [1, 2, 3, 4]
my_list[1:3] = [5, 6, 7] # Replace elements 2 and 3 with 5, 6, and 7
# Result: [1, 5, 6, 7, 4]
```

(4) Slicing (Accessing subsets)

- Slicing creates a new list containing a subset of the original list's elements. The syntax is `my_list[start:stop:step]`
  - **start**: The index where the slice begins (inclusive, defaults to 0).
  - **stop**: The index where the slice ends (exclusive, defaults to the end of the list).
  - **step**: The increment between indices (defaults to 1).

- **Example:**

```
my_list = [0, 1, 2, 3, 4, 5, 6]

# Get elements from index 1 up to (but not including) index 4
subset1 = my_list[1:4]
# Result: [1, 2, 3]

# Get elements from the beginning up to index 3
subset2 = my_list[:4]
# Result: [0, 1, 2, 3]

# Get elements from index 4 to the end
subset3 = my_list[4:]
# Result: [4, 5, 6]
```

```
# Get the entire list (a copy)
subset4 = my_list[:]
# Result: [0, 1, 2, 3, 4, 5, 6]

# Get every second element
subset5 = my_list[::2]
# Result: [0, 2, 4, 6]

# Reverse the list using a negative step
reversed_list = my_list[::-1]
# Result: [6, 5, 4, 3, 2, 1, 0]
```

(4) Tuple

Q-1) Introduction to tuples, immutability.

- **Tuples**

➢ Tuples in Python are **ordered, immutable collections** of elements. Immutability means that once a tuple is created, its size and content cannot be modified.

➢ A tuple is a sequence of objects that can contain elements of any data type, such as integers, strings, or even other tuples and lists. They are typically defined by enclosing the comma-separated values in parentheses `()`.

- **Syntax:**

```
# A tuple of different data types
my_tuple = (1, "hello", 3.14)
```

- **Immutability**

➢ Immutability is a core concept in Python where an object's state or value cannot be altered after its creation.

- **No Item Assignment**: Attempting to change an element within a tuple after it has been created will result in a `TypeError`.
- **Fixed Size**: Tuples do not have methods to add or remove elements (like append() or remove() for lists) because their size is fixed upon creation.

- **Why Immutability?**:
    - **Data Integrity**: Immutability ensures that the data remains constant throughout the program, preventing accidental modification.
    - **Performance**: Tuples generally use less memory and are slightly faster to process than lists because Python can optimize their memory allocation.
    - **Dictionary Keys**: Only immutable (and hashable) objects can be used as keys in a dictionary. Tuples can serve this purpose, while lists cannot.

Q-2) Creating and accessing elements in a tuple.

- In Python, you can create tuples by using parentheses `()` and access their elements using **indexing** (square brackets `[]`) or **slicing**.

Creating Tuples

- Tuples are created by placing comma-separated elements inside parentheses. Parentheses are optional in most cases, a technique known as "tuple packing".

- **Empty tuple:**

```
empty_tuple = ()
```

- **Tuple with a single element:**

  single_element_tuple = (10,) # The comma is necessary

- **Tuple with elements:**

  my_tuple = (1, 2, "three", 4.0)

- **Tuple without parentheses (tuple packing):**

  another_tuple = 5, 6, "seven" # This is also a tuple

- **Using the `tuple()` constructor:**

  list_to_tuple = tuple([1, 2, 3])

  string_to_tuple = tuple("hello")

Accessing elements in a tuple

➢ Elements in a tuple are accessed by their index, which is zero-based.

- **Example:**

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

  # Access the first element (index 0)

  print(fruits[0]) # Output: apple

```
# Access the third element (index 2)
print(fruits[2]) # Output: cherry
```

Q-3) Basic operations with tuples: concatenation, repetition, membership.

- In Python, basic operations with tuples, such as concatenation, repetition, and membership testing, are performed using standard operators that work similarly to those for other sequence types like strings and lists.

- **Concatenation**

- The + operator is used to concatenate two or more tuples. This operation creates a **new** tuple containing the elements from both original tuples, as tuples are immutable (cannot be changed after creation). Concatenation is only supported between the same data types; you cannot concatenate a tuple with a list or a string directly.

  - **Syntax:**

    ```
    tuple1 + tuple2
    ```

  - **Example:**

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

- **Repetition**

➢ The * operator performs repetition, creating multiple copies of a tuple and combining them into a new tuple. The number of repetitions must be a non-negative integer.

- **Syntax:**

  `tuple * n` (where `n` is an integer)

- **Example:**

```
tuple_items = ('Hi!',)
# The comma is necessary for a single-element tuple
result = tuple_items * 4
print(result)
# Output: ('Hi!', 'Hi!', 'Hi!', 'Hi!')
```

- **Membership**

➢ The in and not in operators are used to test whether a specific item exists within a tuple. They return a boolean value (True or False).

- **Syntax:**
  - item in tuple
  - item not in tuple

- **Example:**

```
my_tuple = (1, 9, 8, 8)

print(1 in my_tuple)

# Output: True


print(25 in my_tuple)

# Output: False


print(25 not in my_tuple)

# Output: True
```

(5) Accessing Tuples

Q-1) Accessing tuple elements using positive and negative indexing.

- In Python, you can access tuple elements using both positive and negative indexing. Tuples are zero-indexed, meaning the first element is at index 0

- **Positive Indexing**

- Positive indexing starts from the beginning of the tuple. The first element is at index 0, the second at 1, and so on.

  - **Example:**

```
my_tuple = ('apple', 'banana', 'cherry', 'date')


# Accessing elements using positive indexing
first_element = my_tuple[0]
second_element = my_tuple[1]
third_element = my_tuple[2]


print(f"First element: {first_element}")   # Output: First element: apple
print(f"Second element: {second_element}") # Output: Second element: banana
print(f"Third element: {third_element}")   # Output: Third element: cherry
```

- **Negative Indexing**

➢ Negative indexing starts from the end of the tuple. The last element is at index -1, the second to last at -2, and so on.

- **Example:**

```
my_tuple = ('apple', 'banana', 'cherry', 'date')


# Accessing elements using negative indexing
last_element = my_tuple[-1]
second_last_element = my_tuple[-2]
third_last_element = my_tuple[-3]


print(f"Last element: {last_element}")          # Output: Last element: date
print(f"Second last element: {second_last_element}") # Output: Second last element: cherry
print(f"Third last element: {third_last_element}")   # Output: Third last element: banana
```

Q-2) Slicing a tuple to access ranges of elements.

- In Python, you can access ranges of elements in a tuple using **slicing**, which involves using the colon (:) notation within square brackets. Slicing allows you to create a new tuple containing a sequence of elements from the original tuple without modifying the original.

- **Syntax:**

  tuple[start:stop:step]

- **start**: The index where the slice begins (inclusive). If omitted, it defaults to the beginning of the tuple (index 0).

- **stop**: The index where the slice ends (exclusive). The element at this index is **not** included in the new tuple. If omitted, it defaults to the end of the tuple.

- **step**: The interval between elements to select. If omitted, it defaults to 1.

- **Basic Slicing Examples**

Syntax	Description	Example (if tup = (0, 1, 2, 3, 4, 5, 6, 7, 8))	Resulting Tuple
tup[start:stop]	Elements from start (inclusive) to stop (exclusive).	tup[2:5]	(2, 3, 4)
tup[start:]	Elements from start to the end of the tuple.	tup[4:]	(4, 5, 6, 7, 8)

tup[:stop]	Elements from the beginning to stop (exclusive).	tup[:3]	(0, 1, 2)
tup[:]	A copy of the entire tuple.	tup[:]	(0, 1, 2, 3, 4, 5, 6, 7, 8)

- **Slicing with a Step**

➢ You can also specify a step to select elements at specific intervals.

Syntax	**Description**	**Example (if tup = (0, 1, 2, 3, 4, 5, 6, 7, 8))**	**Resulting Tuple**
tup[::step]	All elements with the given interval.	tup[::2]	(0, 2, 4, 6, 8)
tup[start:stop:step]	Elements within a range with an interval.	tup[1:8:3]	(1, 4, 7)

- **Using Negative Indices**

➢ Negative indices count from the end of the tuple backward (e.g., -1 is the last element, -2 is the second to last).

Syntax	Description	Example (if tup = (0, 1, 2, 3, 4, 5, 6, 7, 8))	Resulting Tuple
tup[-N:]	The last N elements of the tuple.	tup[-3:]	(6, 7, 8)
tup[:-N]	All elements except the last N.	tup[:-2]	(0, 1, 2, 3, 4, 5, 6)
tup[::-1]	All elements in reverse order.	tup[::-1]	(8, 7, 6, 5, 4, 3, 2, 1, 0)

(6) Dictionaries

Q-1) Introduction to dictionaries: key-value pairs.

- Python dictionaries are a fundamental data structure used to store data in **key-value pairs**, allowing for efficient retrieval and management of information. They are a mutable, ordered collection (as of Python 3.7) where each unique key maps to a specific value.

- **Key Characteristics**

  - **Key-Value Pairs:** Each item in a dictionary consists of a key and its associated value, separated by a colon (:), within curly braces ({}).

  - **Unique, Immutable Keys:** Keys must be unique within a dictionary and must be of an immutable data type, such as strings, numbers, or tuples. Lists and other dictionaries cannot be used as keys.

  - **Mutable Values:** Values can be of any data type (strings, integers, lists, even other dictionaries) and can be duplicated or changed.

  - **Ordered (Python 3.7+):** Dictionaries maintain the order in which items were inserted.

- **How to Use Dictionaries**

  - **Creating a Dictionary**

    - You can create a dictionary using curly braces.

- **Example:**

```
# Creating a dictionary with data
student = {
    "name": "John",
    "age": 25,
    "course": "Python"
}

# Creating an empty dictionary
empty_dict = {}
```

- **Accessing Values**

- Values are accessed using their corresponding keys within square brackets or the safe get() method.

  - **Example:**

```
# Accessing a value using its key
print(student["name"])
# Output: John

# Using get() to avoid KeyError if the key doesn't exist
print(student.get("address"))
# Output: None (or a specified default value if provided)
```

Q-2) Accessing, adding, updating, and deleting dictionary elements.

- Python dictionaries store data in key-value pairs and are mutable, allowing for efficient accessing, adding, updating, and deleting of elements.

- **Accessing Dictionary Elements**

- You can access values using their corresponding keys.

  - **Using square brackets `[]` :**

    - **Example:**

```
my_dict = {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
print(my_dict['model'])
# Output: Mustang
```

  - **Using the `.get()` method:**

    - **Example:**

```
my_dict = {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

print(my_dict.get('brand'))
# Output: Ford
print(my_dict.get('color', 'Key not found'))
# Output: Key not found (returns a default value instead of an error)
```

- ## Adding & Updating Dictionary Elements

➢ You can add new items or modify existing ones using the same syntax: direct assignment or the .update() method.

- **Using direct assignment [ ] = :**

```
# Adding a new item
my_dict['color'] = 'red'

# Updating an existing item (overwrites the old value)
my_dict['year'] = 2024
```

- **Using the .update() method:**

```
# Add multiple items or update existing ones with another dictionary
my_dict.update({'engine': 'V8', 'year': 2025})
# my_dict is now {'brand': 'Ford', 'model': 'Mustang', 'year': 2025, 'engine': 'V8'}
```

- ## Deleting Dictionary Elements

➢ Various methods are available to remove items depending on whether you need the removed value back or want to clear the entire dictionary.

- **Using the del keyword:**

```
# Delete a specific item by key
del my_dict['model']

# Delete the entire dictionary object (subsequent use of my_dict will raise a NameError)
# del my_dict
```

- **Using the `.pop()` method:**

```
# Removes a key and returns its value
removed_brand = my_dict.pop('brand')
print(removed_brand)
# Output: Ford
```

- **Using the `.popitem()` method:**

```
# Removes and returns the last inserted key-value pair (in Python 3.7+; order can be random in older versions)
last_item = my_dict.popitem()
print(last_item)
# Output: ('year', 1964) (or similar tuple)
```

- **Using the `.clear()` method:**

```
# Removes all elements, making the dictionary empty

my_dict.clear()

print(my_dict)

# Output: {}
```

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

- In Python, the `keys()`, `values()`, and `items()` methods are used to retrieve the components of a dictionary as dynamic view objects. These views reflect any changes made to the original dictionary.

- **keys()**

➢ The keys() method returns a view object that contains all the keys in the dictionary.

- **Syntax**:

  dictionary.keys()

- **Return Value**:

➢ A dict_keys view object, which is an iterable collection of the keys.

- **Example:**

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

all_keys = car.keys()
print(all_keys)

# Output: dict_keys(['brand', 'model', 'year'])
```

- **values()**

➢ The `values()` method returns a view object that contains all the values in the dictionary.

- **Syntax**:

  dictionary.values()

- **Return Value**:

➢ A `dict_values` view object, which is an iterable collection of the values.

- **Example:**

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
all_values = car.values()
print(all_values)
# Output: dict_values(['Ford', 'Mustang', 1964])
```

- **items()**

➢ The `items()` method returns a view object that contains all the key-value pairs in the dictionary as a list of tuples.

- **Syntax**:

  dictionary.items()

- **Return Value**:

➢ A `dict_items` view object, where each item is a `(key, value)` tuple. This is particularly useful for iterating through both keys and values simultaneously in a `for` loop.

- **Example:**

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

all_items = car.items()
print(all_items)

# Output: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

# Iterating using items()
for key, value in car.items():
  print(f"{key}: {value}")
```

(7) Working with Dictionaries

Q-1) Iterating over a dictionary using loops.

- In Python, you can iterate over a dictionary using a `for` loop, utilizing different methods depending on whether you need the keys, values, or both.

- **Loop Through All Keys**

- By default, iterating over a dictionary in a for loop iterates through its keys. This is the most efficient method if you only need the keys or if you plan to access the values using the keys.

  - **Example:**

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

for key in my_dict:
  print(key)

# Alternatively, explicitly using .keys() for clarity:
# for key in my_dict.keys():
#   print(key)
```

- ## Loop Through All Values

➢ To iterate only through the values, use the .values() method. This is efficient when you don't need the corresponding keys.

- **Example:**

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

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

- ## Loop Through Both Keys and Values (Items)

➢ To access both the key and the value in the same loop iteration, use the `.items()` method, which returns key-value pairs as tuples. This is a very common and "Pythonic" approach.

- **Example:**

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

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

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

- Merging two lists into a dictionary in Python can be efficiently achieved using either a loop or a `zip()` function.

- **Using `zip()`**

- The `zip()` function is the most common and Pythonic way to merge two lists into a dictionary. It aggregates elements from the two iterables (the lists) into tuples, which are then used as key-value pairs in the dictionary constructor**.**

  - **Example:**

```
keys = ['apple', 'banana', 'cherry']
values = [10, 20, 30]

# Using zip() with the dict() constructor
merged_dict = dict(zip(keys, values))

print(merged_dict)
# Output: {'apple': 10, 'banana': 20, 'cherry': 30}
```

- **Using a Loop**

- While `zip()` is generally more efficient and readable, you can also use a loop, typically a `for` loop combined with `zip()`, to achieve the same result.

  - **Example:**

```
keys = ['apple', 'banana', 'cherry']
values = [10, 20, 30]
```

```
merged_dict = {}

# Using a for loop with zip() to iterate over pairs
for k, v in zip(keys, values):
    merged_dict[k] = v

print(merged_dict)
# Output: {'apple': 10, 'banana': 20, 'cherry': 30}
```

Q-3) Counting occurrences of characters in a string using dictionaries.

- To count occurrences of characters in a string using dictionaries in Python, you can iterate through the string and use the character as the dictionary key, incrementing its value each time it appears.

- **Example:**

```
def count_chars(s):
    """
    Counts character occurrences in a string using a dictionary.

    Args:
        s: The input string.

    Returns:
        A dictionary with character keys and their occurrence counts.
    """
    counts = {}
    for char in s:
        if char in counts:
            counts[char] += 1
        else:
            counts[char] = 1
    return counts
```

```
# Example usage:
my_string = "programming is fun"
char_counts = count_chars(my_string)
print(char_counts)
```

- **Output**

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

Alternative Methods

Using `dict.get()`

- The `get()` method simplifies the logic by allowing you to specify a default value (0 in this case) if the key does not exist.

  - **Example:**

```
def count_chars_get(s):
    counts = {}
    for char in s:
        counts[char] = counts.get(char, 0) + 1
    return counts

# Example usage:
my_string = "hello world"
char_counts = count_chars_get(my_string)
print(char_counts)
```

  - **Output**

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

- **Using `collections.Counter`**

➢ Python's built-in `collections` module provides a `Counter` class that is specifically designed for this exact purpose and is the most Pythonic solution.

- **Example:**

```
from collections import Counter


def count_chars_counter(s):
    # Counter does the counting automatically when initialized with a string
    return Counter(s)


# Example usage:
my_string = "python is powerful"
char_counts = count_chars_counter(my_string)
print(char_counts)
```

- **Output**

```
Counter({'o': 3, 'p': 2, ' ': 2, 't': 1, 'h': 1, 'y': 1, 'n': 1, 'i': 1, 's': 1, 'w': 1, 'e': 1, 'r': 1, 'f': 1, 'u': 1, 'l': 1})
```

(8) Functions

Q-1) Defining functions in Python.

- In Python, functions are defined using the `def` keyword, followed by a name, parentheses for optional parameters, and a colon. The code block within the function is then specified by indentation.

- **Syntax**

```
def function_name(parameter1, parameter2):
    """Optional docstring to describe the function."""
    # Code block starts here (must be indented)
    result = parameter1 + parameter2
    return result # Optional: returns a value to the caller
```

- **Key Components**

  - **def keyword**: Starts the function definition.

  - **function_name**: A valid Python identifier that names the function. Function names typically use lowercase letters and underscores (snake_case).

  - **(and)**: Enclose any input parameters (placeholders for data the function needs).

- **: (colon)**: Marks the end of the function header and the start of the function body.

- **Indentation**: Python uses indentation (typically four spaces or a tab) to define the function's code block. All code at the same level of indentation belongs to the function.

- **"""Docstring"""**: An optional string literal placed as the first statement in the function body to document its purpose. This can be viewed using help(function_name).

- **return statement**: Used to exit the function and pass data back to the calling code. If no return statement is present, the function implicitly returns None.

Calling a Function

- Once defined, a function is executed by "calling" it using its name followed by parentheses, which contain any necessary arguments (the actual values for the parameters).

  - **Example:**

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

# Function call with an argument
greet("Pythonista")
```

- **Benefits of Functions**

➢ Using functions makes code more organized, readable, and efficient by providing:

  - **Modularity**: Breaking a program into smaller, manageable chunks.
  - **Reusability**: Avoiding repetitive code by calling the same function multiple times.
  - **Abstraction**: Hiding the internal logic and exposing a simple interface (name and parameters).

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

➢ In Python, functions are a core concept and can be categorized based on whether they accept parameters (inputs) and whether they return a value (outputs) using the `return` statement.

(1) Function without Parameters and without Return Values

➢ These functions perform an action without needing external data and do not explicitly return a value. They typically use side effects like printing to the console. The return value is implicitly `None`.

- **Example:**

```
def greet():
    """Prints a simple greeting message."""
    print("Hello, Python!")


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

(2) Function with Parameters and without Return Values

- These functions accept inputs (parameters) to tailor their behavior but do not return a value. They still perform an action with side effects.

- **Example:**

```
def greet_person(name):
    """Prints a personalized greeting message."""
    print(f"Hello, {name}!")

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

(3) Function without Parameters and with Return Values

- These functions perform a calculation or retrieve data internally and then use the `return` statement to pass a value back to the caller.

- **Example:**

```
def get_pi():
    """Returns the value of pi."""
    return 3.14159

# The returned value can be stored in a variable
```

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

(4) Function with Parameters and with Return Values

- These are the most common and versatile functions. They accept inputs, perform some computation based on those inputs, and return a result.

- **Example:**

```
def add_numbers(a, b):
    """Adds two numbers and returns the sum."""
    sum_result = a + b
    return sum_result

# The returned value can be used in further operations
total = add_numbers(5, 3)
print(total) # Output: 8
```

Q-3) Anonymous functions (lambda functions).

- In Python, an **anonymous function** is a function defined without a name using the `lambda` keyword. These are often referred to as **lambda functions** and are typically used for short-term, throwaway tasks.

- **Syntax:** lambda arguments: expression.

- **Basic Structure**

- A lambda function consists of three parts: the `lambda` keyword, one or more arguments, and a single expression.

    - **Automatic Return:** The result of the expression is returned automatically without needing a return keyword.

    - **Arguments:** They can take any number of arguments but are strictly limited to one expression.

- **Example:**

```
# Simple lambda to add 10 to a number
add_ten = lambda x: x + 10
print(add_ten(5)) # Output: 15
```

(9) Modules

Q-1) Introduction to Python modules and importing modules.

- Python **modules** are single `.py` files that logically organize code into reusable units (functions, classes, and variables), while the `import` statement is used to bring that functionality into other programs. This practice promotes code reusability, organization, and easier maintenance.

- **Modules can be:**

  - **Built-in:** Part of Python's standard library (e.g., math, os, random).
  - **External:** Third-party modules that need to be installed using a package manager like pip (e.g., pandas, numpy).
  - **User-defined:** Modules you create yourself by saving your code in a .py file.

- **Methods for Importing Modules**

- The `import` statement is the primary way to access the contents of a module. Imports are typically placed at the top of a Python file for readability.

  - **Import the entire module:** This is the standard and recommended approach. You must use the module name as a prefix to access its functions or variables (dot notation).

- **Example:**

  import math

  # Accessing content using the module name prefix

  print(math.pi)

  print(math.sqrt(25))

- **Import with an alias:** Use the `as` keyword to assign a shorter, alternative name to a module, which is common for modules with long names.

  - **Example:**

    import math as m

    # Accessing content using the alias prefix
    print(m.pi)
    print(m.sqrt(25))

- **Import specific items:** Use the `from` keyword to import only selected functions or variables directly into your program's namespace. This allows you to use them without the module name prefix.

  - **Example:**

  from math import pi, sqrt

  # Accessing content directly by name

  print(pi)

  print(sqrt(25))

- **Import all items (discouraged):** You can import everything from a module into the current namespace using the wildcard `*`. This is generally discouraged in larger programs because it can cause naming conflicts and make code harder to read and debug.

    - **Example:**

```
# Avoid this in large projects
from math import *
print(pi)
print(sqrt(25))
```

Q-2) Standard library modules: math, random.

- The Python standard library includes the `math` and `random` modules, which provide extensive functionality for mathematical operations and generating pseudo-random numbers, respectively. Both modules must be imported before use.

- **`math` Module**

- The `math` module provides access to common mathematical functions and constants.

    - **Syntax:**

```
import math
```

- **Common Functions & Constants:**

  - **math.sqrt(x)**: Returns the square root of x.
  - **math.ceil(x)**: Returns the smallest integer greater than or equal to x.
  - **math.floor(x)**: Returns the largest integer less than or equal to x.
  - **math.pow(x, y)**: Returns x raised to the power of y (x**y).
  - **math.factorial(x)**: Returns the factorial of x.
  - **math.log(x, base)**: Returns the logarithm of x to the specified base (natural log if base is not specified).
  - **math.sin(x), math.cos(x), math.tan(x)**: Trigonometric functions where x is in radians.
  - **math.degrees(x), math.radians(x)**: Convert angles between radians and degrees.
  - **math.pi**: The mathematical constant π pi (approximately 3.14159).
  - **math.e**: The mathematical constant *e* (approximately 2.71828).

- **Example:**

```
import math


# Calculate the square root of 64
print(f"Square root of 64: {math.sqrt(64)}")


# Calculate the value of pi
print(f"Value of Pi: {math.pi}")
```

```
# Calculate the factorial of 5
print(f"Factorial of 5: {math.factorial(5)}")
```

- ## `random` Module

➢ The `random` module is used to generate pseudo-random numbers for various distributions and operations.

- **Syntax:**

```
import random
```

- **Common Functions:**

  - **random.random()**: Returns a random float number between 0.0 and 1.0 (inclusive of 0.0, exclusive of 1.0).

  - **random.randint(a, b)**: Returns a random integer N such that a <= N <= b (both inclusive).

  - **random.randrange(start, stop, step)**: Returns a randomly selected element from the specified range.

  - **random.choice(seq)**: Returns a random element from a non-empty sequence (list, tuple, or string).

  - **random.shuffle(x)**: Shuffles a sequence x in place (modifies the original list).

- **random.sample(population, k)**: Returns a list of k unique random elements from the population sequence.

- **Example:**

```
import random

# Generate a random float between 0 and 1
print(f"Random float: {random.random()}")

# Generate a random integer between 1 and 10 (inclusive)
print(f"Random integer: {random.randint(1, 10)}")

# Select a random element from a list
fruits = ["apple", "banana", "cherry", "date"]
print(f"Random fruit: {random.choice(fruits)}")

# Shuffle a list in place
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(f"Shuffled list: {numbers}")
```

Q-3) Creating custom modules.

- Creating a custom module in Python is a straightforward process that allows you to organize and reuse your code effectively. A module is essentially any Python file that can be imported into other Python programs.

- ## Steps to Create and Use a Custom Module

(1) Create the Module File

- Open a new text file and save it with a **.py** extension. The file name will be the name of your module. For example, save the file as mymodule.py.

- Inside this file, you can define functions, variables, classes, or any other Python objects you want to reuse.

- Module names should be short, all-lowercase, and unique to avoid conflicts with built-in modules.

  - **Example: mymodule.py**

```
"""
This is a custom module named mymodule.
It contains a greeting function and a dictionary of information.
"""
def greeting(name):
  """Prints a welcome message."""
  print(f"Hello, {name}!")


person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

(2) Ensure Accessibility (Module Path)

- The Python interpreter needs to know where to find your custom module. The easiest way to ensure this is to place the module file in the **same directory** as the script where you intend to use it.

- Alternatively, you can add the module's directory to the Python path using the sys module in your main script:

```
import sys
# Modify this path to the directory containing your module
sys.path.append('/path/to/your/modules/directory')
```

- You can also store your modules in one of Python's default paths, such as the `site-packages` directory.

(3) Import and Use the Module

- In a separate Python script (in the same directory), you can now import your custom module using the `import` statement.

  - **Example: main_script.py**

```
import mymodule


# Call the function defined in the module
mymodule.greeting("Jonathan")
```

```
# Access the dictionary variable
age = mymodule.person1["age"]
print(f"Jonathan's age is {age}.")
```