Assignment

Data structure

Q1. Discuss string slicing and provide examples.

Ans= String slicing in Python allows you to access a portion (or "slice") of a string using a specific range of indices.

The syntax for slicing is `string[start:end:step]`, where:

- `start` is the index where the slice begins (inclusive).

- `end` is the index where the slice ends (exclusive).

- `step` is the interval between each character in the slice (optional).

Examples

1. Basic Slicing:

```python
s = "Hello, World!"
print(s[0:5])  # Output: "Hello"
```

This slices the string from index `0` to `5` (not including `5`).

2. Slicing with Step:

```python
s = "Hello, World!"
print(s[0:5:2])  # Output: "Hlo"
```

This slices the string from index `0` to `5` with a step of `2`.

3. Omitting Start and End:

```python
s = "Hello, World!"
print(s[:5])   # Output: "Hello"
print(s[7:])   # Output: "World!"
```

Omitting the start defaults to the beginning of the string, and omitting the end defaults to the end of the string.

4. Negative Indices:

```python
s = "Hello, World!"
print(s[-6:-1])  # Output: "World"
```

Negative indices count from the end of the string.

5. Reversing a String:

```python
s = "Hello, World!"
print(s[::-1])  # Output: "!dlroW ,olleH"
```

Using a step of `-1` reverses the string.

String slicing is a powerful tool for manipulating and accessing portions of strings in Python.

Q2. Explain the key features of lists in Python.

Ans = Lists in Python are versatile and widely used data structures with several key features:

1. **Ordered:**
   - Elements in a list have a defined order, and this order will not change unless explicitly modified.
2. **Mutable:**
   - Lists can be changed after their creation. You can add, remove, or modify elements.
3. **Heterogeneous:**
   - Lists can contain elements of different data types (integers, strings, floats, objects, etc.).
4. **Dynamic Size:**
   - Lists can grow or shrink as needed, allowing you to add or remove elements dynamically.
5. **Indexing and Slicing:**
   - Lists support indexing (both positive and negative) and slicing, making it easy to access and modify elements or sublists.
6. **Methods:**
   - Python provides a rich set of built-in methods for lists, such as append(), extend(), insert(), remove(), pop(), clear(), index(), count(), sort(), reverse(), and copy().

7. **Iteration:**
    - Lists can be easily iterated over using loops (e.g., for loops), making them convenient for various operations.
8. **Comprehensions:**
    - List comprehensions provide a concise way to create lists based on existing lists, allowing for efficient and readable code.

Q3. Describe how to access, modify and delete elements in a list with examples.

Ans = **Accessing Elements**

1. By Index:

```python
my_list = [10, 20, 30, 40, 50]

print(my_list[2])  # Output: 30

print(my_list[-1]) # Output: 50
```

2. By Slicing:

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

print(my_list[:3])   # Output: [10, 20, 30]
```

Modifying Elements

1. By Index:

```python
my_list[1] = 25

print(my_list)  # Output: [10, 25, 30, 40, 50]
```

2. By Slicing:

```python
my_list[1:3] = [15, 35]

print(my_list)  # Output: [10, 15, 35, 40, 50]
```

Deleting Elements

1. Using `del`:

```python
del my_list[2]
print(my_list)  # Output: [10, 15, 40, 50]
del my_list[1:3]
print(my_list)  # Output: [10, 50]
```

2. Using `remove()`:

```python
my_list.remove(50)
print(my_list)  # Output: [10]
```

3. Using `pop()`:

```python
popped_element = my_list.pop()
print(popped_element)  # Output: 10
print(my_list)        # Output: []
 my_list = [10, 20, 30, 40, 50]
popped_element = my_list.pop(1)
print(popped_element)  # Output: 20
print(my_list)        # Output: [10, 30, 40, 50]
```

These examples illustrate how to access, modify, and delete elements in a list using various methods available in Python.

Q4. Compare and contrast tuples and lists with examples.

Ans= **Lists vs. Tuples in Python**

Both lists and tuples are used to store collections of items, but they have some key differences:

Lists

1. Mutable:

- Lists can be modified after creation (elements can be added, removed, or changed).

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

2. Syntax:

- Lists are defined using square brackets `[]`.

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

3. Methods:

- Lists have a wide range of built-in methods such as `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `clear()`, `sort()`, and `reverse()`.

```python
my_list.sort()
print(my_list)  # Output: [1, 2, 3, 4]
```

Tuples

1. Immutable:

- Tuples cannot be modified after creation (elements cannot be added, removed, or changed).

```python
my_tuple = (1, 2, 3)
# my_tuple[0] = 4  # This will raise a TypeError
```

2. Syntax:

- Tuples are defined using parentheses `()`.

```python
```

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

3. Methods:

- Tuples have fewer built-in methods compared to lists, mainly `count()` and `index()`.

```python

print(my_tuple.index(2))  # Output: 1```

**Comparison**

- **Performance:**

- Tuples are generally faster than lists due to their immutability.

**- Use Cases:**

- Lists are used when the data needs to be modified frequently.

- Tuples are used for fixed collections of items and can be used as keys in dictionaries.

**Examples**

**List Example:**

```python

my_list = [1, 2, 3]

my_list.append(4)

print(my_list)  # Output: [1, 2, 3, 4]

my_list[1] = 5

print(my_list)  # Output: [1, 5, 3, 4]```

**Tuple Example:**

```python

my_tuple = (1, 2, 3)

# my_tuple[1] = 5  # Raises TypeError

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

In summary, the choice between lists and tuples depends on whether you need a mutable sequence or an immutable one.
```

Q5. Describe the key features of sets and provide examples of their use.

Ans= **Key Features of Sets in Python**

1. **Unordered:**
   - Sets do not maintain any order of elements.
   - Elements cannot be accessed by index.
2. **Mutable:**
   - Sets can be modified after creation. Elements can be added or removed.
3. **No Duplicate Elements:**
   - Sets automatically remove duplicates. Each element in a set is unique.
4. **Heterogeneous:**
   - Sets can contain elements of different data types (e.g., integers, strings, tuples).
5. **Set Operations:**
   - Supports mathematical set operations like union, intersection, difference, and symmetric difference.
6. **Dynamic Size:**
   - Sets can grow and shrink dynamically as elements are added or removed.
7. **Hashable Elements:**
   - All elements in a set must be immutable and hashable (e.g., integers, strings, tuples).

Examples

1. **Creating a Set:**

   python
   Copy code
   my_set = {1, 2, 3, 4}
   print(my_set) # Output: {1, 2, 3, 4}

   # Using the set() function
   another_set = set([1, 2, 3, 4, 4, 2])
   print(another_set) # Output: {1, 2, 3, 4}

- **Set Operations:**
  - Useful in scenarios requiring mathematical set operations like finding common elements, differences, etc.
- **Membership Testing:**
  - Efficient for checking whether an element is present in a collection.

Sets provide a powerful way to handle collections of unique items and perform common set operations efficiently.

Q6. Discuss the use cases of tuples and sets in Python programming.

Ans= Use Cases of Tuples

1. **Immutable Data Storage:**

   - Tuples are ideal for storing data that should not be changed throughout the program.
2. **Returning Multiple Values from Functions:**

- Functions can return multiple values using tuples.

3. **Dictionary Keys:**

- Tuples can be used as keys in dictionaries because they are immutable and hashable.

4. **Structuring Data:**

- Tuples can be used to group related data together.

5. **Iteration and Unpacking:**

- Tuples can be easily unpacked into variables, making iteration and assignment straightforward.

Use Cases of Sets

1. **Removing Duplicates:**
   - Sets are used to eliminate duplicate items from a list.
2. **Membership Testing:**

- Sets provide an efficient way to check for the presence of an item.\

3. **Mathematical Set Operations:**

- Sets support operations like union, intersection, difference, and symmetric difference, useful in various algorithms.

4. **Filtering Data:**

- Sets can be used to filter out unwanted elements from a collection.

5. **Finding Common Elements:**

- Useful for finding common elements between different collections.

  Both tuples and sets offer unique advantages that make them suitable for specific use cases in Python programming. Tuples are useful for immutable collections and structured data, while sets excel at handling unique elements and performing set operations.\

Q7. Describe how to add, modify and delete items in a dictionary with examples.

Ans= Adding, Modifying, and Deleting Items in a Dictionary

Adding Items

1. **Adding a New Key-Value Pair:**

   python
   Copy code
   my_dict = {"name": "Alice", "age": 25}
   my_dict["city"] = "New York"
   print(my_dict) # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

Modifying Items

1. **Updating an Existing Key-Value Pair:**

   python
   Copy code
   my_dict["age"] = 26
   print(my_dict) # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}

2. **Using update() Method:**

   python
   Copy code
   my_dict.update({"name": "Bob", "city": "San Francisco"})
   print(my_dict) # Output: {'name': 'Bob', 'age': 26, 'city': 'San Francisco'}

Deleting Items

1. **Using del Statement:**

   python
   Copy code
   del my_dict["age"]
   print(my_dict) # Output: {'name': 'Bob', 'city': 'San Francisco'}

2. **Using pop() Method:**

   python
   Copy code
   city = my_dict.pop("city")
   print(city) # Output: San Francisco

```
print(my_dict)  # Output: {'name': 'Bob'}
```

3. **Using popitem() Method:**
    - Removes and returns the last inserted key-value pair.

```
python
Copy code
my_dict["country"] = "USA"
last_item = my_dict.popitem()
print(last_item)  # Output: ('country', 'USA')
print(my_dict)    # Output: {'name': 'Bob'}
```

These examples show how to add, modify, and delete items in a dictionary in Python, demonstrating the versatility and ease of use of dictionaries for managing key-value pairs.

Q8. Discuss the importance of dictionary keys being immutable and provide example.

Ans= **Importance of Dictionary Keys Being Immutable**

1. **Hashability:**
    - Dictionary keys must be hashable, meaning their hash value remains constant during their lifetime. Immutable types like strings, numbers, and tuples (containing only immutable elements) are hashable.
2. **Consistency:**
    - If a key could change, it would disrupt the integrity of the dictionary, making it impossible to reliably access the associated value. Immutable keys ensure that the key-value mapping remains consistent.
3. **Performance:**
    - Immutable keys guarantee efficient lookups, insertions, and deletions. Mutable objects could change in ways that affect their hash values, leading to performance degradation and unexpected behavior.

Example

Using an immutable key:

```
python
Copy code
my_dict = {("Alice", "NY"): 25, ("Bob", "CA"): 30}
print(my_dict[("Alice", "NY")])  # Output: 25
```

Attempting to use a mutable key:

```
python
Copy code
try:
```

```
    my_dict = {[1, 2]: "value"}  # Raises TypeError
except TypeError as e:
    print(e)  # Output: unhashable type: 'list'
```

In this example, a list (mutable type) cannot be used as a dictionary key, leading to a TypeError. This illustrates the necessity of immutable keys to ensure dictionary operations remain predictable and efficient.