1) Discuss string slicing and provide examples?

Ans= String slicing in Python allows access to specific parts of a string using string[start:end:step].

Example: strings1 = "hello"

```
strings2 = "world"

strings1 + ", "+strings2

'hello, world'
```

strings1 = "I am a very beautiful"

strings1[0:4]

```
'Iam '
```

strings1[0:5]

```
'Iam a'
strings1[0:8]
 'I am a ve'
```

#modification of string

s = "I am a student"

s.replace("student", "teacher")

```
I am a teacher'
```

#use case

address = "araji no.2108 sect koylanagar, kanpur nagar"

address

```
'araji no.2108 sect koylanagar, kanpur nagar'
```

2) Explain the key features of lists in Python?

Ans= In Python, lists are one of the most versatile and commonly used data structures. Here are the key features of lists:

```
1. Ordered: Elements keep their order and can be accessed by
index.


2. Mutable: Lists can be changed after creation (adding,
removing, or updating items).


3. Heterogeneous Elements: Can store different data types
(integers, strings, etc.) in one list.


4. Dynamic Size: Lists can grow or shrink as needed.


5. Supports Slicing: Parts of a list can be accessed using
slicing (list[start:end]).



Lists are flexible and ideal for various data-handling tasks.
Example:- type([])
            List

grocery_list = ["milk", "bread","biscuit","apple"]
          grocery_list
['milk', 'bread', 'biscuit', 'apple']
```

3) **Describe how to access, modify, and delete elements in a list with examples?**

Ans= In Python, lists are mutable, so you can access, modify, and delete elements easily.

1. Accessing Elements

Use indexing to access elements. Indexes start at 0 for the first element.

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

print(my_list[1])    # Output: 20

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

2. Modifying Elements

You can modify elements by directly assigning a new value to a specific index.

```
my_list[1] = 25       # Changes the element at index 1
print(my_list)        # Output: [10, 25, 30, 40, 50]
```

3. Deleting Elements

Use del, remove(), or pop() to delete elements.

Using del: Deletes element by index.

```
del my_list[2]        # Removes the element at index 2
print(my
```

4) Compare and contrast tuples and lists with examples?

Ans= Both tuples and lists are data structures in Python that can store multiple items, but they have some important differences. Here's a compare.

List vs. Tuple:

1. Mutability:

List: Mutable (can change).

Tuple: Immutable (cannot change).

2. Syntax:

List: []

Tuple: ()

3. Use:

List: Flexible, can add/remove items.

Tuple: Fixed, good for constant data.

4. Performance:

Tuple: Faster due to immutability.

List: Slightly slower.

Example:

```
# List

my_list = [1, 2, 3]

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

# Tuple

my_tuple = (1, 2, 3)

# my_tuple.append(4)  # Error: Tuples are immutable

my_list = [1, 2, 2, 3, 4, 4]

my_set = set(my_list)

print(my_set)  # Output: {1, 2, 3, 4}
```

2. Membership Testing: Checking if an item is in a set is faster than with lists.

```
my_set = {1, 2, 3, 4}

print(3 in my_set)  # Output: True
```

3. Set Operations: Used for mathematical operations like union, intersection, and difference.

```
set_a = {1, 2, 3}

set_b = {2, 3, 4}

print(set
```

5) **Describe the key features of sets and provide examples of their use?**

Ans= Sets in Python are unordered collections of unique elements. They have several key features and are commonly used when handling distinct items or performing set operations.

Key Features of Sets

1. Unordered: Elements in a set have no specific order, so indexing is not possible.

2. Unique Elements: Sets automatically remove duplicate values, ensuring all elements are unique.

3. Mutable: Sets are mutable, meaning elements can be added or removed, but only immutable elements (e.g., numbers, strings, tuples) can be stored in them.

4. Efficient Operations: Sets allow efficient membership testing and set operations like union, intersection, and difference.

Example Use Cases

1. Removing Duplicates: Sets can automatically filter out duplicate values from a list.

```
my_list = [1, 2, 2, 3, 4, 4]

my_set = set(my_list)

print(my_set)  # Output: {1, 2, 3, 4}
```

2. Membership Testing: Checking if an item is in a set is faster than with lists.

```
my_set = {1, 2, 3, 4}

print(3 in my_set)  # Output: True
```

3. Set Operations: Used for mathematical operations like union, intersection, and difference.

```
set_a = {1, 2, 3}

set_b = {2, 3, 4}

print(set
```

6) Discuss the use cases of tuples and sets in Python programming?

Ans = Tuples and sets in Python each serve unique purposes and are useful in different scenarios due to their specific characteristics.

Use Cases of Tuples

1. Fixed Collections: Tuples are immutable, making them ideal for storing fixed data that shouldn't change throughout the program, such as geographic coordinates or RGB color values.

```
coordinates = (40.7128, -74.0060)  # Latitude and longitude

color = (255, 0, 0)  # RGB color for red
```

2. Dictionary Keys: Tuples can be used as keys in dictionaries (unlike lists) because they are hashable.

```
locations = {("New York", "USA"): "NYC", ("Paris", "France"): "PAR"}
```

3. Returning Multiple Values: Functions can return multiple values as tuples, making it easy to unpack them.

```
def get_student():

   return ("Alice", 20)

name, age = get_student()
```

4. Data Integrity: Tuples ensure that data remains unchanged. For example, storing constant configurations or settings.

```
config = ("localhost", 8080, "https")
```

Use Cases of Sets

1. Removing Duplicates: Sets automatically eliminate duplicate values, making them useful for cleaning up lists with duplicate entries.

```
items = ["apple", "banana", "apple", "orange"]

unique_items = set(items)  # Output: {'apple', 'banana', 'orange'}
```

2. Membership Testing: Sets provide fast membership testing, which is helpful for tasks like checking if an item exists in a large dataset.

```
allowed_users = {"Alice", "Bob", "Charlie"}

if "Alice" in allowed_users:

   print("Access granted")
```

3. Set Operations: Sets support mathematical operations like union, intersection, and difference, making them ideal for tasks like comparing or combining data groups.

```
set_a = {1, 2, 3}

set_b = {2, 3, 4}

common_elements = set_a & set_b  # Output: {2, 3}
```

4. Unique Items Collection: Sets are perfect for managing collections where uniqueness is essential, like storing IDs or tags.

```
tags = {"python", "coding", "python"}  # Output: {'python', 'coding'}
```

In summary:

Tuples are great for fixed, unchanging collections and data integrity.

Sets are ideal for uniqueness, membership testing, and efficient data comparisons.

7) Describe how to add, modify, and delete items in a dictionary with examples?

Ans = Python, dictionaries are mutable, so you can add, modify, and delete items easily.

1. Adding Items

To add a new key-value pair, simply assign a value to a new key.

```
my_dict = {"name": "Alice", "age": 25}

my_dict["city"] = "New York"   # Adds a new key-value pair

print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

2. Modifying Items

To modify an existing item, reassign a new value to the key.

```
my_dict["age"] = 26           # Modifies the value of 'age'
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

3. Deleting Items

You can delete items using del, pop(), or popitem().

Using del: Deletes an item by key.

```
del my_dict["city"]         # Removes the 'city' key-value pair
print(my_dict)  # Output: {'name': 'A…
```

In Python, dictionaries are mutable, so you can add, modify, and delete items easily

Adding, Modifying, and Deleting Items in a Dictionary

1. Add:

```
my_dict = {"name": "Alice"}
my_dict["age"] = 25          # Adds new key-value pair
```

2. Modify:

```
my_dict["age"] = 26          # Updates 'age' value
```

3. Delete:

del:

```
del my_dict["age"]         # Deletes 'age'
```

pop():

```
my_dict.pop("name")        # Removes and returns 'name' value
```

popitem():

```
my_dict.popitem()          # Removes the last item
```

8) Discuss the importance of dictionary keys being immutable and provide examples?

Ans = In Python, dictionary keys must be immutable (unchangeable) because they are hashed to quickly retrieve values. If a key could change after being added to a dictionary, its hash would change, causing issues in locating the stored value. Therefore, keys are limited to immutable types like strings, numbers, and tuples.

Why Immutable Keys Are Important

1. Efficient Access: Hashing makes lookups fast, as each key's position is determined by its hash. If keys were mutable, their hash could change, making access unreliable.

2. Data Integrity: Immutable keys ensure the data structure remains stable and prevents unintended behavior when adding, updating, or deleting entries.

Example

```
# Valid keys (immutable)

my_dict = {

   "name": "Alice",

   42: "Answer",

   (1, 2): "Tuple Key"  # Tuples are immutable, so they can be keys

}

# Invalid key (mutable)

# my_dict[{1: "one"}] = "Invalid"  # Error: dictionaries can't have mutable keys
```

Using immutable keys, like strings or tuples, ensures stable, reliable dictionary operations.

Assignment submitted by

SAKSHI MODANWAL