ASSIGNMENT FOR DATA STRUCTURE NEW

Name : UZAIR MUSHTAQ

Course name : Data analytics course june - 24

Q1:

String slicing is like cutting up a cake—you take only the piece you want! In Python, slicing lets you grab a part of a string by specifying a start and end point.

The syntax looks like this:
string[start:end:step]

- **Start**: Where you begin (the first index is 0).
- **End**: Where you stop (but it **doesn't include** the character at this index).
- **Step**: How many characters you skip (optional—by default, it's 1).

Now, let's dive into an example:

```
my_string = "Hello, World!"

sliced_string = my_string[0:5]

print(sliced_string)
```

Hello

Q2:

Here are the **key features of lists** in Python, briefly:

1. **Ordered**: Elements have a specific sequence, accessed by index.
2. **Mutable**: You can change, add, or remove elements after creation.
3. **Heterogeneous**: Can store different data types (integers, strings, etc.).
4. **Dynamic Size**: Lists grow or shrink as needed.
5. **Indexed Access**: Supports both positive and negative indexing.
6. **Slicing**: Allows extracting sublists using slice notation.
7. **Built-in Methods**: Includes methods like .append(), .remove(), .sort(), etc.
8. **Nested Lists**: Can contain lists within lists (multi-dimensional).

These features make Python lists versatile for many tasks.

Here are the **key features of lists** in Python with examples:

1. **Ordered**: Elements are stored in a sequence.

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

print(my_list[0])  # Outputs: 1
```

2. **Mutable**: You can modify elements.

```
my_list[1] = 10

print(my_list)  # Outputs: [1, 10, 3]
```

3. **Heterogeneous**: Can hold different data types.

```
my_list = [1, "hello", 3.5]
print(my_list)  # Outputs: [1, "hello", 3.5]
```

4. **Indexed Access**: Use indices to access elements.

```
print(my_list[-1])  # Outputs: 4 (last element)
```

Q3:

In Python, you can **access**, **modify**, and **delete** elements in a list easily. Here's how:

1. Accessing Elements

- You can access list elements using **indexing** (starting at 0) or **negative indexing** (starting at -1 for the last element).

```
my_list = [10, 20, 30, 40, 50]
print(my_list[2])    # Accesses the element at index 2: Outputs 30
print(my_list[-1])   # Accesses the last element: Outputs 50
```

2. Modifying Elements

- Lists are mutable, so you can **change the value** of an element using its index.

Example:

```
my_list = [10, 20, 30, 40, 50]
my_list[1] = 25     # Modifies the element at index 1
print(my_list)      # Outputs: [10, 25, 30, 40, 50]
```

- You can also **modify multiple elements** using slicing.

Example:

```
my_list[1:3] = [15, 35]  # Modifies elements from index 1 to 2
print(my_list)           # Outputs: [10, 15, 35, 40, 50]
```

3. Deleting Elements

- You can **delete elements** using the del keyword, remove(), or pop().

- **Using del**: Deletes an element by index.

```
del my_list[1]    # Deletes the element at index 1
print(my_list)    # Outputs: [10, 35, 40, 50]
```

Using remove(): Removes the first occurrence of a value.

```
my_list.remove(40)   # Removes the element with value 40
print(my_list)       # Outputs: [10, 35, 50]
```

. **Using pop()**: Removes and returns the element at a specific index (default is the last element).

```
my_list.pop()        # Removes and returns the last element (50)
print(my_list)       # Outputs: [10, 35]
```

These are some simple ways to **access**, **modify**, and **delete** elements in a Python list!

Q4:

Tuples and **Lists** are both data structures in Python, but they have key differences and similarities. Here's a comparison:

1. Mutability

- **List**: Mutable, meaning elements can be changed after the list is created.
- **Tuple**: Immutable, meaning elements cannot be modified once the tuple is created.

Example:

```
# List (Mutable)
my_list = [1, 2, 3]
my_list[1] = 10   # Modifies element at index 1
print(my_list)    # Outputs: [1, 10, 3]

# Tuple (Immutable)
my_tuple = (1, 2, 3)
my_tuple[1] = 10  # Raises an error, as tuples cannot be modified
```

2. Syntax

- **List**: Created using square brackets [].
- **Tuple**: Created using parentheses ().

Example:

```
my_list = [1, 2, 3]      # List
my_tuple = (1, 2, 3)     # Tuple
```

3. Use Cases

- **List**: Suitable when you need a collection of items that may need to change (e.g., adding, removing, or modifying elements).
- **Tuple**: Used when the data is fixed and should not change (e.g., representing constant values).

4. Performance

- **List**: Slower compared to tuples due to the overhead of mutability.
- **Tuple**: Faster, as they are immutable and have lower memory overhead.

Example:

```
my_list = [1, 2, 3]  # Lists require more memory
my_tuple = (1, 2, 3) # Tuples are more memory efficient
```

5. Methods

- **List**: Lists have more methods like append(), remove(), pop(), etc.
- **Tuple**: Tuples have fewer methods, mainly count() and index(), since you cannot modify their contents.

Example:

```
my_list = [1, 2, 3]
my_list.append(4)    # List method to add an element
print(my_list)      # Outputs: [1, 2, 3, 4]

my_tuple = (1, 2, 3)
# No method to modify a tuple
```

6. Immutability Benefit

- **Tuples** can be used as **dictionary keys** or elements of sets because they are immutable.
- **Lists** cannot be used as dictionary keys or set elements because they are mutable.

Example:

Summary Table:

Feature	**List**	**Tuple**
Mutability	Mutable	Immutable
Syntax	Square brackets []	Parentheses ()
Performance	Slower	Faster
Methods	Many methods	Limited methods
Use Cases	Dynamic, modifiable	Static, fixed
Length	Can change	Fixed after creation
Immutability	Not hashable	Can be used as keys

Conclusion: Use **lists** when you need a dynamic and modifiable collection. Use **tuples** when the data should remain constant and for performance benefits when working with fixed collections.

Q5:

A **set** in Python is an unordered collection of unique elements. Here are the **key features** of sets:

1. **Unordered**
  - Sets do not maintain any specific order of elements, unlike lists or tuples.
  - Example:
    ```python
    my_set = {1, 2, 3, 4}
    print(my_set)  # Outputs: {1, 2, 3, 4} (but the order may vary)
    ```

```

### 2. **Unique Elements**
  - Sets automatically remove duplicate values. Every element in a set is unique.
  - Example:
    ```python
    my_set = {1, 2, 2, 3, 4}
    print(my_set)  # Outputs: {1, 2, 3, 4} (no duplicates)
    ```

3. **Mutable**
  - Sets are mutable, meaning you can add or remove elements. However, the elements themselves must be immutable (e.g., integers, strings, tuples).
  - Example:
    ```python
    my_set = {1, 2, 3}
    my_set.add(4)   # Adds a new element
    print(my_set)   # Outputs: {1, 2, 3, 4}
    ```

4. **Unindexed**
  - Unlike lists, sets cannot be accessed using an index, since the elements are unordered.
  - Example:
    ```python
    my_set = {1, 2, 3}
    # print(my_set[0])  # Raises an error because sets are unordered
    ```

5. **Supports Set Operations**
  - Sets support mathematical set operations like **union**, **intersection**, **difference**, and **symmetric difference**.

  - **Union**: Combines elements from two sets (removes duplicates).
    ```python
    set1 = {1, 2, 3}
    set2 = {3, 4, 5}
    print(set1 | set2)  # Outputs: {1, 2, 3, 4, 5}
    ```

  - **Intersection**: Returns elements common to both sets.
    ```python
    print(set1 & set2)  # Outputs: {3}
    ```

  - **Difference**: Returns elements in the first set but not in the second.
    ```python
    print(set1 - set2)  # Outputs: {1, 2}
    ```
    ```

- **Symmetric Difference**: Returns elements in either set but not in both.
  ```python
  print(set1 ^ set2)  # Outputs: {1, 2, 4, 5}
  ```

6. **No Duplicate Elements**
- Sets automatically remove any duplicate elements when created or modified.
- Example:
  ```python
  my_set = {1, 2, 2, 3, 4}
  print(my_set)  # Outputs: {1, 2, 3, 4}
  ```

7. **Immutable Sets (Frozen Sets)**
- Python provides **frozen sets**, which are immutable versions of sets. Once created, their elements cannot be changed.
- Example:
  ```python
  frozen_set = frozenset([1, 2, 3])
  # frozen_set.add(4)  # Raises an error because it's immutable
  ```

Use Case Example: Removing Duplicates
Sets are often used to remove duplicates from a collection of data.

```python
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)
print(unique_numbers)  # Outputs: {1, 2, 3, 4, 5}
```

Use Case Example: Set Operations
Sets are useful in tasks involving common or different elements between groups, such as analyzing user data.

```python
online_users = {"Alice", "Bob", "Carol"}
offline_users = {"Carol", "Dave"}
active_users = online_users & offline_users  # Intersection of sets
print(active_users)  # Outputs: {'Carol'}
```

Summary
- **Sets** are unordered, mutable, and hold unique elements.
- They are ideal for eliminating duplicates and performing mathematical set operations.

Q6:

Use Cases of **Tuples** in Python:

1. **Immutable Data**:
   - When you need a collection of items that should not change, a tuple is the best choice. Tuples are immutable, making them ideal for data that must remain constant, like configuration settings or fixed data structures.

   Example:
   ```python
   coordinates = (10.5, 25.3)  # A fixed coordinate point
   ```

2. **Dictionary Keys**:
   - Since tuples are immutable, they can be used as keys in dictionaries, whereas lists cannot. This is useful when you need to associate data with a combination of values.

   Example:
   ```python
   location_map = {(10, 20): "Park", (30, 40): "Mall"}  # Tuple as a key
   print(location_map[(10, 20)])  # Outputs: "Park"
   ```

3. **Return Multiple Values from a Function**:
   - Tuples are often used to return multiple values from a function in a clean and readable way.

   Example:
   ```python
   def get_user_info():
       name = "Alice"
       age = 30
       return name, age  # Returns a tuple

   user_info = get_user_info()
   print(user_info)  # Outputs: ('Alice', 30)
   ```

4. **Heterogeneous Data**:
   - Tuples can store different types of data, which is useful when you want to group related but different types of values together (like a record).

   Example:
   ```python
   person = ("John", 25, True)  # A tuple holding name, age, and status
   ```

5. **Performance-Sensitive Applications**:

- Since tuples are faster and use less memory compared to lists, they are preferred in performance-sensitive applications where data is not modified.

Example:
```python
tuple_data = (1, 2, 3, 4)  # Faster creation and access
```

Use Cases of **Sets** in Python:

1. **Removing Duplicates**:
- One of the most common use cases of sets is to remove duplicate items from a list. Since sets only store unique elements, converting a list to a set automatically removes duplicates.

Example:
```python
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)
print(unique_numbers)  # Outputs: {1, 2, 3, 4, 5}
```

2. **Membership Testing**:
- Sets offer efficient membership tests due to their underlying hash table structure. This is useful when you need to check if an element exists in a collection, especially with large datasets.

Example:
```python
valid_colors = {"red", "green", "blue"}
print("red" in valid_colors)  # Outputs: True
print("yellow" in valid_colors)  # Outputs: False
```

3. **Mathematical Set Operations**:
- Sets provide powerful operations like union, intersection, difference, and symmetric difference, making them ideal for solving problems involving comparisons between collections.

Example:
```python
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Union: elements from both sets
print(set1 | set2)  # Outputs: {1, 2, 3, 4, 5}
```

```
# Intersection: common elements
print(set1 & set2)  # Outputs: {3}

# Difference: elements in set1 but not in set2
print(set1 - set2)  # Outputs: {1, 2}
```

4. **Removing Unwanted Elements**:
   - Sets are useful when you need to filter out unwanted values from a collection by performing set difference or intersection operations.

   Example:
   ```python
   all_students = {"Alice", "Bob", "Carol", "Dave"}
   failed_students = {"Carol", "Dave"}

   passed_students = all_students - failed_students  # Difference
   print(passed_students)  # Outputs: {"Alice", "Bob"}
   ```

5. **Handling Large Datasets**:
   - Sets are particularly useful when working with large datasets for operations like duplicate elimination, set operations, or fast membership tests.

   Example:
   ```python
   dataset = set(range(1000000))  # A large set of numbers
   print(500000 in dataset)  # Fast membership test
   ```

Summary:
- **Tuples** are great for representing fixed collections of data, such as coordinates, records, or function returns. Their immutability makes them suitable for use as dictionary keys or when you want data to remain unchanged.
- **Sets** are ideal for situations where you need to ensure uniqueness, perform mathematical set operations, or check membership efficiently, especially in large datasets.

Both data structures are essential tools for efficient and clean Python programming!

Q7:

In Python, **dictionaries** are collections of key-value pairs, where keys are unique, and each key is associated with a value. You can **add**, **modify**, and **delete** items in a dictionary using various methods. Here's how:

1. **Adding Items to a Dictionary**

- You can add a new key-value pair by assigning a value to a new key.

Example:
```python
my_dict = {"name": "Alice", "age": 25}
my_dict["city"] = "New York"  # Adds a new key-value pair
print(my_dict)  # Outputs: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

- Alternatively, you can use the `update()` method to add multiple key-value pairs at once.

Example:
```python
my_dict.update({"country": "USA", "occupation": "Engineer"})
print(my_dict)  # Outputs: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA', 'occupation': 'Engineer'}
```

2. **Modifying Items in a Dictionary**
- To modify a value in the dictionary, simply assign a new value to an existing key.

Example:
```python
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
my_dict["age"] = 26  # Modifies the value of the key 'age'
print(my_dict)  # Outputs: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

- The `update()` method can also be used to modify values for multiple keys simultaneously.

Example:
```python
my_dict.update({"age": 27, "city": "San Francisco"})
print(my_dict)  # Outputs: {'name': 'Alice', 'age': 27, 'city': 'San Francisco'}
```

3. **Deleting Items from a Dictionary**
- You can delete an item by using the `del` statement with the key. This removes the key-value pair.

Example:
```python
my_dict = {"name": "Alice", "age": 27, "city": "San Francisco"}
del my_dict["city"]  # Deletes the key 'city'
print(my_dict)  # Outputs: {'name': 'Alice', 'age': 27}
```

- The `pop()` method removes a key-value pair and also returns the value. If the key doesn't exist, it raises a `KeyError`, but you can provide a default value to avoid errors.

Example:
```python
age = my_dict.pop("age")  # Removes the key 'age' and returns its value
print(age)  # Outputs: 27
print(my_dict)  # Outputs: {'name': 'Alice'}
```

- The `popitem()` method removes and returns the **last inserted** key-value pair as a tuple. It raises an error if the dictionary is empty.

Example:
```python
my_dict = {"name": "Alice", "age": 27}
item = my_dict.popitem()  # Removes the last inserted item
print(item)  # Outputs: ('age', 27)
print(my_dict)  # Outputs: {'name': 'Alice'}
```

- The `clear()` method removes all key-value pairs, leaving the dictionary empty.

Example:
```python
my_dict.clear()  # Removes all items
print(my_dict)  # Outputs: {}
```

Summary:
- **Add** items by assigning a value to a new key or using the `update()` method.
- **Modify** items by re-assigning values to existing keys or using `update()`.
- **Delete** items using `del`, `pop()`, `popitem()`, or `clear()` methods.

This makes dictionaries a versatile data structure for storing and manipulating key-value pairs!

Q8:

In Python, **dictionary keys must be immutable** to ensure the integrity and reliability of dictionary operations. The immutability of keys is important for several reasons, as outlined below:

1. **Hashability**:
- Python dictionaries use **hashing** to store and retrieve key-value pairs efficiently. A **hash function** generates a unique hash code for each key, which helps to quickly locate the associated value in memory.

- Immutable objects like strings, numbers, and tuples are **hashable**, meaning their hash value remains constant throughout their lifetime.
- Mutable objects, such as lists or sets, cannot be hashed because their content (and hence their hash value) can change. If a key could change after being inserted into the dictionary, it would corrupt the hash table and make it impossible to find the correct value.

Example:
```python
my_dict = {"name": "Alice", 42: "age"}  # Valid keys (immutable)
# my_dict = {[1, 2]: "list as key"}  # Raises an error because lists are mutable
```

2. **Consistency in Dictionary Operations**:
- If dictionary keys were mutable, their values could change after being added, breaking the consistency of key-value pair lookups. This would make it impossible to reliably access or update dictionary entries because the key's hash value could change.

Example:
```python
my_dict = {"name": "Alice"}
# Imagine if "name" could change to "John" – the dictionary wouldn't be able to find the original key "name" anymore
```

3. **Avoiding Key Collisions**:
- Dictionary keys must be unique. If mutable objects were allowed as keys and they changed after being used, they might collide with other keys (i.e., produce the same hash as another key), leading to data loss or incorrect retrieval.

Example:
```python
my_dict = {("x", "y"): 1}  # A tuple (immutable) is a valid key
# If tuples were mutable and changed after insertion, they might collide with other keys
```

4. **Immutability Ensures Stability**:
- When you insert an immutable key (e.g., a string or a tuple) into a dictionary, Python ensures that the key's value will not change, maintaining the stability of the dictionary structure. This is especially critical when dictionaries are used for large-scale or performance-sensitive applications.

Example:
```python
# Strings and numbers are commonly used as dictionary keys because they are immutable
person = {"name": "Alice", "age": 30}
```

Example: Why Mutable Keys Are Not Allowed

Here’s an example that illustrates what happens if you try to use a mutable object, like a list, as a dictionary key.

```python
my_dict = {}
key = [1, 2, 3]  # This is a mutable list

# Attempt to use a list as a key
# my_dict[key] = "value"  # Raises a TypeError: unhashable type: 'list'
```

Lists are mutable, so Python raises an error, preventing you from using them as keys. To fix this, you could convert the list into a tuple, which is immutable:

```python
key = (1, 2, 3)  # A tuple is immutable
my_dict[key] = "value"  # Works perfectly fine
print(my_dict)  # Outputs: {(1, 2, 3): 'value'}
```

Immutable Objects as Dictionary Keys:
- **Valid Immutable Keys**: Strings, numbers, tuples (containing only immutable elements), and booleans.
- **Invalid Mutable Keys**: Lists, dictionaries, sets, or custom objects that do not implement hashability.

Example:
```python
# Valid keys
my_dict = {
    "name": "Alice",  # String (immutable)
    123: "ID",        # Integer (immutable)
    (1, 2): "point"   # Tuple (immutable)
}
```

Summary:
The immutability of dictionary keys ensures:
- **Efficient key lookups** using consistent hash values.
- **Data integrity** by preventing changes to keys after insertion.
- **Avoidance of collisions**, where two different keys could have the same hash value.

By enforcing immutability for keys, Python ensures that dictionaries remain fast, reliable, and predictable for storing and retrieving key-value pairs.