Python Assignment- Data Structure Basics

Q 1. Discuss string slicing and provide examples.

Ans- String slicing in Python allows you to extract a portion of a string by specifying a start index, an end index, and an optional step. The syntax for slicing is string[start:end:step].

Components:
start: The index to start the slice (inclusive).
end: The index to end the slice (exclusive).
step: The number of steps to skip (default is 1).
Examples:
Basic Slicing:

python

Copy
```
text = "Hello, World!"
slice1 = text[0:5]  # Outputs: 'Hello'
```
Omitting start and end:

python

Copy
```
slice2 = text[:5]   # Outputs: 'Hello' (from start to index 5)
slice3 = text[7:]   # Outputs: 'World!' (from index 7 to end)
```
Using Negative Indices:

python

Copy
```
slice4 = text[-6:-1]  # Outputs: 'World' (from index -6 to -1)
```
Using step:

python

Copy
```
slice5 = text[::2]  # Outputs: 'Hlo ol!' (every second character)
```
Summary
String slicing is a powerful feature in Python for extracting specific portions of strings, making it easy to manipulate and process text data.

Q 2. Explain the key features of lists in python.

Ans- Here are the key features of lists in Python:

1. **Ordered**: Lists maintain the order of elements as they are added.
2. **Mutable**: You can change, add, or remove elements after the list is created.
3. **Dynamic Size**: Lists can grow or shrink in size as needed.
4. **Heterogeneous**: Lists can contain elements of different data types (e.g., integers, strings, objects).
5. **Indexed**: Elements can be accessed via their index, starting from 0.
6. **Nested**: Lists can contain other lists, allowing for complex data structures.

These features make lists versatile for various programming tasks in Python.

Q 3. Describe how to access, modify and delete elements in a lists with examples.
Ans- ### Accessing Elements
We can access elements in a list using their index:

```python
```

my_list = [10, 20, 30, 40]
print(my_list[1]) # Output: 20

### Modifying Elements
We can change an element by assigning a new value to its index:

```python
my_list[2] = 300
print(my_list)  # Output: [10, 20, 300, 40]
```

### Deleting Elements
We can remove elements using `del`, `pop()`, or `remove()`:

1. Using `del`:
   ```python
   del my_list[1]
   print(my_list)  # Output: [10, 300, 40]
   ```

2. Using `pop()` (removes and returns the last element by default):
   ```python
   last_element = my_list.pop()
   print(last_element)  # Output: 40
   print(my_list)       # Output: [10, 300]
   ```

3. Using `remove()` (removes the first occurrence of a specified value):
   ```python
   my_list.remove(10)
   print(my_list)  # Output: [300]
   ```

These methods allow us to effectively access, modify, and delete elements in a Python list.

Q 4. Compare and contrast tuples and lists with examples.
Ans- Here's a comparison of tuples and lists in Python:

### **1. Mutability**
- **Lists**: Mutable (can be changed after creation).
  ```python
  my_list = [1, 2, 3]
  my_list[0] = 10  # List is now [10, 2, 3]
  ```

- **Tuples**: Immutable (cannot be changed after creation).
  ```python
  my_tuple = (1, 2, 3)
  # my_tuple[0] = 10  # This will raise an error
  ```

### **2. Syntax**
- **Lists**: Defined using square brackets.
  ```python
  my_list = [1, 2, 3]
  ```

- **Tuples**: Defined using parentheses.
  ```python
  ```

  my_tuple = (1, 2, 3)

### **3. Performance**
- **Lists**: Generally slower due to the overhead of mutability.

- **Tuples**: Faster due to their immutability, making them more memory-efficient.

### **4. Use Cases**
- **Lists**: Suitable for collections of items that may need to change, such as a list of tasks.

- **Tuples**: Ideal for fixed collections of items, such as coordinates or records.

### **5. Methods**
- **Lists**: Have more built-in methods (e.g., `append()`, `remove()`, `pop()`).

- **Tuples**: Have fewer methods (mostly `count()` and `index()`).

### **Example**
- **List**:
  ```python
  fruits = ['apple', 'banana', 'cherry']
  fruits.append('date')  # Now the list is ['apple', 'banana', 'cherry', 'date']
  ```

- **Tuple**:
  ```python
  coordinates = (10.0, 20.0)
  # coordinates[0] = 15.0  # Error; tuples cannot be modified
  ```

In summary, lists are mutable and versatile, while tuples are immutable and typically used for fixed data structures.

Q 5. Describe the key features of sets and provide examples of their use.
Ans- Sets are fundamental concepts in mathematics and computer science, characterized by the following key features:

1. **Distinctness**: Sets contain unique elements; duplicates are not allowed. For example, the set \(\{1, 2, 3\} \) contains three distinct elements.

2. **Unordered**: The order of elements in a set does not matter. For instance, \(\{2, 1, 3\} \) is considered the same as \(\{1, 2, 3\} \).

3. **Defined membership**: Elements either belong to a set or they do not. For example, in the set \(A = \{a, b, c\} \), it is clear that 'a' is a member but 'd' is not.

Examples of Use:
- **Mathematics**: Sets are used in functions, relations, and to define operations like union and intersection. For example, the union of \(\{1, 2\} \) and \(\{2, 3\} \) is \(\{1, 2, 3\} \).
- **Programming**: Sets are often used to handle collections of items without duplicates, such as storing user IDs. In Python, you might use a set to efficiently check membership or to eliminate duplicates from a list.

These properties make sets a powerful tool for organizing and analyzing data in various fields.

Q 6. Discuss the use cases of tuples and sets in python programming.
Ans- Tuples and sets in Python serve distinct purposes:

### Tuples:
1. **Immutability**: Tuples are immutable, making them suitable for fixed collections of items. Once created, their content cannot change, which is useful for data integrity.
2. **Data Structure**: Often used to group related data, like coordinates (x, y) or returning multiple values from a

function.
3. **Performance**: Less memory overhead compared to lists, hence faster for iteration.

### Sets:
1. **Uniqueness**: Sets automatically eliminate duplicate values, making them ideal for membership checks and ensuring collection of unique items.
2. **Mathematical Operations**: Supports union, intersection, and difference operations, facilitating advanced data manipulation.
3. **Fast Lookups**: Optimal for scenarios where quick membership testing is required.

In summary, use tuples for ordered, fixed collections of data and sets for collections requiring uniqueness and fast membership checks.

Q 7. Describe how to add, modify and delete items in a dictionary with examples.
Ans- In Python, dictionaries are mutable collections of key-value pairs. Here's how to add, modify, and delete items in a dictionary:

### Adding Items
We can add a new key-value pair by simply assigning a value to a new key:

```python
my_dict = {'a': 1, 'b': 2}
my_dict['c'] = 3  # Adding a new item
# my_dict now is {'a': 1, 'b': 2, 'c': 3}
```

### Modifying Items
We can change the value of an existing key by assigning a new value to it:

```python
my_dict['a'] = 10  # Modifying an existing item
# my_dict now is {'a': 10, 'b': 2, 'c': 3}
```

### Deleting Items
We can remove an item using the `del` statement or the `pop()` method:

```python
del my_dict['b']  # Deleting an item
# my_dict now is {'a': 10, 'c': 3}

value = my_dict.pop('c')  # Removes 'c' and returns its value
# my_dict now is {'a': 10}
# 'c' was removed and its value was stored in 'value'
```

These operations allow you to manage items within a dictionary efficiently.

Q 8. Discuss the importance of dictionary keys being immutable and provide examples.
Ans- Immutable dictionary keys are crucial because they ensure the integrity and stability of a dictionary's structure. If keys were mutable, changes to a key could disrupt data retrieval and storage, leading to inconsistencies.

### Importance:
1. **Consistency**: Immutable keys ensure that data remains accessible and reliable.
2. **Hashing**: Dictionaries use hash tables for quick lookups, and mutable objects can't guarantee a stable hash value.
3. **Avoiding Errors**: Prevents accidental changes that could corrupt the data structure.

### Examples:
- **Strings**: Often used as keys because they are immutable. For example:
  ```python
  ```

  my_dict = {"name": "Alice"}
  ~~~
- **Tuples**: Can also serve as keys if they contain only immutable elements. For instance:
  ```python
  my_dict = {(1, 2): "coordinates"}
  ~~~

Overall, immutability in keys is essential for the reliable functionality of dictionaries in programming.
