PW SKILLS

Assignmnet

Data Structure

1. Discuss string slicing and provide examples.

2. Explain the key features of lists in Python.

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

4. Compare and contrast tuples and lists with examples.

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

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

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

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

Ques1: Discuss string slicing and provide examples?

Ans1: String slicing is a method in programming, particularly in Python, that allows you to extract a portion (or slice) of a string. This is done using a specific syntax that involves brackets and indices.

Syntax

The general syntax for string slicing is

string[start:end:step]

- **start: The index where the slice begins (inclusive).**
- **end: The index where the slice ends (exclusive).**
- **step: The number of characters to step through the string (optional)**

Examples

1. **Basic Slicing**

```
s = "Hello, World!"
slice1 = s[0:5]  # 'Hello'
```

2. **Slicing with Omitted Parameters**

- **If you omit start, it defaults to 0.**
- **If you omit end, it defaults to the length of the string.**
- **If you omit step, it defaults to 1.**

```
s = "Python Programming"
slice2 = s[:6]   # 'Python'
slice3 = s[7:]   # 'Programming'
```

3. **Using Step Parameter**

```
s = "abcdefg"
slice4 = s[::2]  # 'aceg' (every second character)
```

4. **Negative Indices** Negative indices count from the end of the string.

```
s = "Hello, World!"
slice5 = s[-6:-1]  # 'World'
```

5. **Reversing a String** You can use slicing to reverse a string

```
s = "abcdef"
reversed_s = s[::-1]  # 'fedcba'
```

String slicing is a powerful feature that enables easy manipulation and extraction of substrings in a concise manner.

Ques2: Explain the key features of lists in Python?

Ans2: Features of Lists in Python

Python lists are versatile data structures that allow you to store and manipulate collections of items. Here are their key features:

1. Ordered:

- Elements in a list are stored in a specific order.
- This order is maintained, and you can access elements by their index.

2. Mutable:

- You can modify the contents of a list after it's created.
- You can add, remove, or change elements at any position.

3. Heterogeneous:

- A list can contain elements of different data types (e.g., integers, strings, floats, even other lists).

4. Dynamic:

- Lists can grow or shrink in size as needed.
- You can add or remove elements without specifying a fixed size upfront.

5. Indexing and Slicing:

- Indexing: Access individual elements using their zero-based index.

```
my_list = [10, 20, 30]
print(my_list[0])  # Output: 10
```

Slicing: Extract a portion of the list using slice notation

```
print(my_list[1:3])  # Output: [20, 30]
```

Python provides a rich set of built-in methods for working with lists:

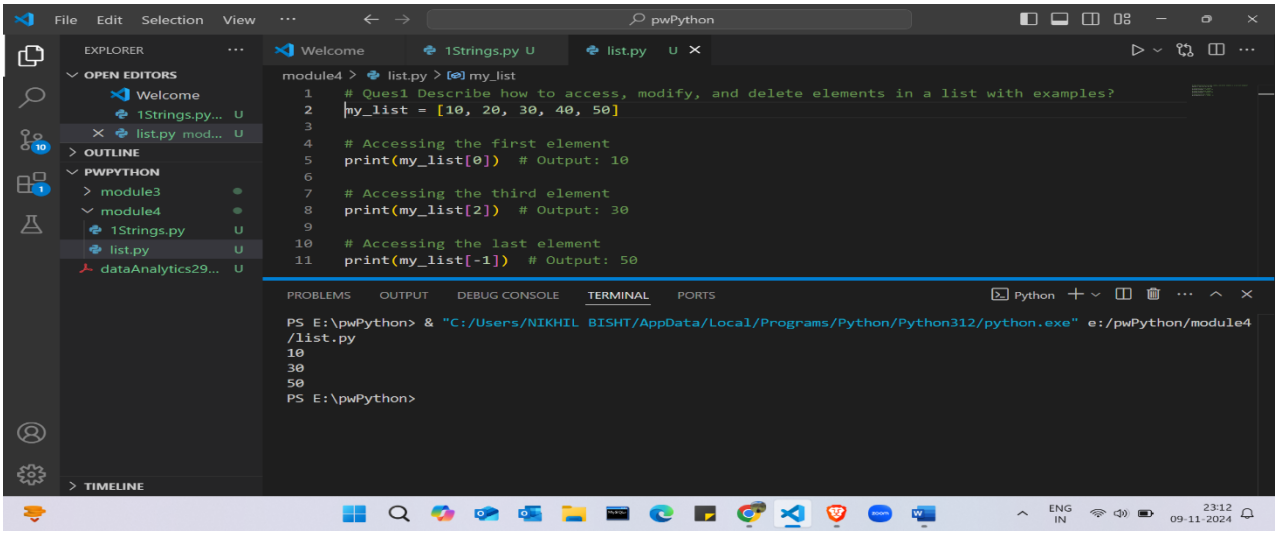
- **append():** Adds an element to the end.
- **insert():** Inserts an element at a specific index.
- **remove():** Removes the first occurrence of an element.
- **pop():** Removes and returns an element at a specific index or the last one.
- **clear():** Removes all elements from the list.
- **sort():** Sorts the list in ascending order.
- **reverse():** Reverses the order of elements.

- **count():** Counts the number of occurrences of an element.
- **index():** Returns the index of the first occurrence of an element

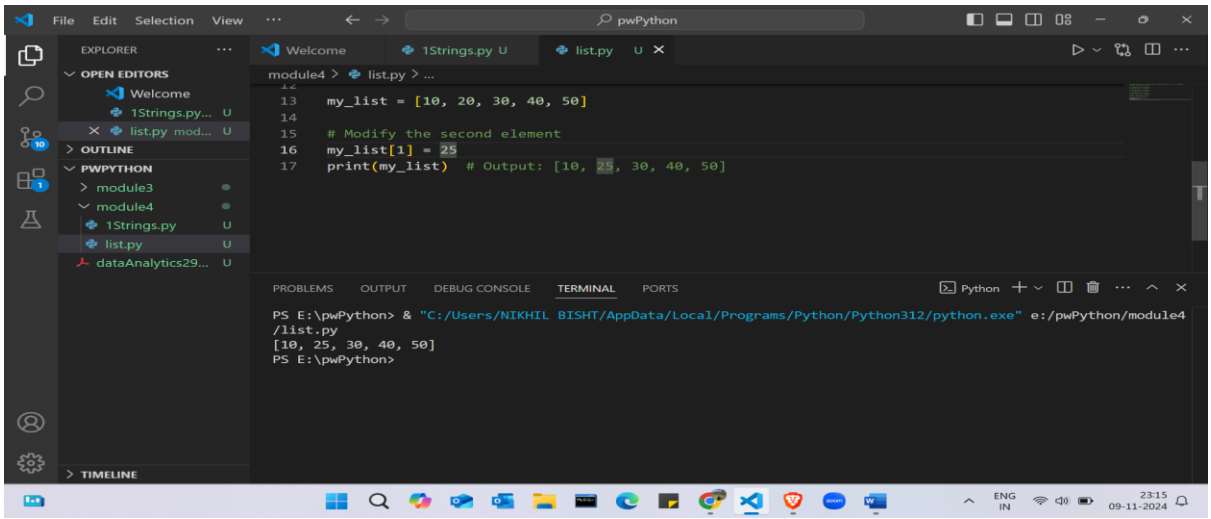
```
my_list = [1, "hello", 3.14, [2, 4]]

# Accessing elements
print(my_list[1])  # Output: hello

# Modifying elements
my_list[2] = 10
print(my_list)  # Output: [1, 'hello', 10, [2, 4]]

# Adding elements
my_list.append(5)
print(my_list)  # Output: [1, 'hello', 10, [2, 4], 5]

# Removing elements
my_list.remove("hello")
print(my_list)  # Output: [1, 10, [2, 4], 5]
```

Ques3: Describe how to access, modify, and delete elements in a list with examples?

ANS3: **Accessing the elements**



Modifying Elements in a List:



Deleting Elements in a List:

There are three main ways to delete elements:

1. **Using the del statement:**

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

# Delete the third element
del my_list[2]
print(my_list)  # Output: [10, 20, 40, 50]
```

```
[10, 20, 40, 50]
```

2. **Using the pop() method:** The pop() method removes and returns the element at a specific index. If no index is specified, it removes and returns the last element.

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

# Remove and return the last element
removed_element = my_list.pop()
print(removed_element)  # Output: 50
print(my_list)  # Output: [10, 20, 30, 40]

# Remove and return the second element
removed_element = my_list.pop(1)
print(removed_element)  # Output: 20
print(my_list)  # Output: [10, 30, 40]
```

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

3. **Using the remove() method:** The remove() method removes the first occurrence of a specified value from the list.

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

# Remove the first occurrence of 20
my_list.remove(20)
print(my_list)  # Output: [10, 30, 20, 40, 50]
```

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

Ques4: Compare and contrast tuples and lists with examples?

Ans4: Tuples and lists are fundamental data structures in Python, each with its own unique properties

Feature	Tuples	Lists
Mutability	Immutable (cannot be changed after creation)	Mutable (can be modified)
Syntax	Enclosed in parentheses `()`	Enclosed in square brackets `[]`
Memory Usage	Generally more memory-efficient	Less memory-efficient
Speed	Faster to iterate over and access elements	Slower to iterate over and access elements

```python
# Tuples
my_tuple = (1, 2, 3, "hello", True)

# Lists
my_list = [10, 20, 30, "world", False]

# Lists
my_list[0] = 15  # This is valid, modifying the first element
print(my_list)  # Output: [15, 20, 30, "world", False]

# Tuples
my_tuple[0] = 5  # This will raise a TypeError: 'tuple' object does not support item assignment
```

```
[15, 20, 30, 'world', False]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-25e899d0e71b> in <cell line: 12>()
     10
     11 # Tuples
---> 12 my_tuple[0] = 5  # This will raise a TypeError: 'tuple' object does not support item assignment

TypeError: 'tuple' object does not support item assignment
```

Ques5: Describe the key features of sets and provide examples of their use?

Ans5: Sets in Python are unordered collections of unique elements. They are defined by enclosing a comma-separated list of elements within curly braces {}.

Key Features of Sets:

1. Unordered: Elements in a set do not have a specific order, unlike lists and tuples.
2. Unique: Sets automatically eliminate duplicate elements, ensuring each element appears only once.
3. Mutable: You can add or remove elements from a set after its creation.
4. No Indexing: Sets do not support indexing, as their elements are not ordered.

Common Set Operations:

```python
# Union: Combines elements from two sets, eliminating duplicates.
set1 = {1, 2, 3}
set2 = {2, 3, 4}
union_set = set1.union(set2)  # {1, 2, 3, 4}
```

```python
union_set
```

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

```python
# Intersection: Finds the common elements between two sets.
intersection_set = set1.intersection(set2)  # {2, 3}
intersection_set
```

```
{2, 3}
```

```python
# Difference: Finds the elements in one set that are not in another
difference_set = set1.difference(set2)  # {1}
difference_set
```

```
{1}
```

```python
# Symmetric Difference: Finds the elements that are in either set, but not both.
symmetric_difference_set = set1.symmetric_difference(set2)  # {1, 4}
symmetric_difference_set
```

Ques6: Discuss the use cases of tuples and sets in Python programming?

Feature	Tuples	Sets
Mutability	Immutable	Mutable
Order	Ordered	Unordered
Duplicate Elements	Allows duplicates	Unique elements only
Common Use Cases		
Storing fixed data	Yes	No
Dictionary keys	Yes	No
Representing records or rows	Yes	No
Removing duplicates	No	Yes
Membership testing	Yes	Yes
Set operations (union, intersection, difference)	No	Yes

Examples

Tuples:

1. Storing fixed data:

```python
person = ("Alice", 30, "New York")
```

Use code with caution.

2. Returning multiple values from a function:

```python
def calculate_area_perimeter(length, width):
    area = length * width
    perimeter = 2 * (length + width)
    return area, perimeter
```

Use code with caution.

3. Dictionary keys:

```python
my_dict = {(1, 2): "value1", (3, 4): "value2"}
```

Use code with caution.

Sets:

1. Removing duplicates:

```python
my_list = [1, 2, 2, 3, 3, 3]
unique_elements = set(my_list)  # {1, 2, 3}
```

Use code with caution.

2. Membership testing:

```python
my_set = {10, 20, 30}
if 20 in my_set:
    print("20 is in the set")
```

Use code with caution.

3. Set operations:

```python
set1 = {1, 2, 3}
set2 = {2, 3, 4}
union_set = set1.union(set2)  # {1, 2, 3, 4}
intersection_set = set1.intersection(set2)  # {2, 3}
difference_set = set1.difference(set2)  # {1}
```

Use code with caution.

Ques7: Describe how to add, modify, and delete items in a dictionary with examples?

Ans7: A Python dictionary is an unordered collection of key-value pairs. Each key must be unique, while the values can be of any data type

Adding Items:

To add a new key-value pair to a dictionary, you simply assign a value to a new key:

```
my_dict = {'name': 'Alice', 'age': 30}
my_dict['city'] = 'New York'
print(my_dict)  # Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

Modifying Items:

To modify an existing value, you assign a new value to the key:

```
my_dict['age'] = 31
print(my_dict)  # Output: {'name': 'Alice', 'age': 31, 'city': 'New York'}
```

```
{'name': 'Alice', 'age': 31, 'city': 'New York'}
```

Deleting Items:

You can delete items using the del keyword:

```
del my_dict['city']
print(my_dict)  # Output: {'name': 'Alice', 'age': 31}
```

```
{'name': 'Alice', 'age': 31}
```

Or, you can use the **pop() method** to remove an item and return its value:

```
city = my_dict.pop('name')
print(city)  # Output: Alice
print(my_dict)  # Output: {'age': 31}
```

```
Alice
{'age': 31}
```

Ques8: Discuss the importance of dictionary keys being immutable and provide examples?

Ans8: Importance of Immutable Dictionary Keys

In Python, dictionary keys must be immutable objects. This is a fundamental requirement that ensures the efficient and reliable operation of dictionaries.

Key Reasons:

1. **Hashing:**
    - Dictionaries use a hashing mechanism to store and retrieve key-value pairs efficiently.
    - The hash value of a key is calculated based on its immutable properties.
    - If a key were mutable, its hash value could change after it's added to the dictionary, leading to inconsistencies and potential data loss.
2. **Consistency and Predictability:**
    - Immutable keys guarantee that the hash value of a key will remain constant throughout its existence in the dictionary.
    - This consistency is essential for accurate and efficient lookups.

    - If keys were mutable, it would become difficult to locate specific values, as their hash values might change.

3. **Efficient Lookup:**
    - Hashing allows for fast lookup of values based on their keys.
    - Immutable keys contribute to the efficiency of this process by ensuring that the hash value remains stable.

Examples of Immutable Objects Suitable for Dictionary Keys:

- **Integers:** 1, 2, 3
- **Floating-point numbers:** 3.14, 2.718
- **Strings:** "hello", "world"
- **Tuples:** (1, 2), ("a", "b")

Examples of Mutable Objects Not Suitable for Dictionary Keys:

- **Lists:** [1, 2, 3]
- **Sets:** {1, 2, 3}
- **Dictionaries:** {'a': 1, 'b': 2}

By adhering to the immutability requirement for dictionary keys, you can ensure the correct functioning of dictionaries and avoid potential errors and performance issues.