PW SKILL
DATA STRUCTURE ASSIGNENT

1. Discuss string slicing and provide examples.

String Slicing in Python

String slicing in Python allows you to extract a part of a string by specifying a **start**, **end**, and optionally a **step**. This is a powerful feature for working with substrings, as it gives you precise control over how you extract and manipulate portions of strings.

Syntax of String Slicing

```
python
Copy code
string[start:end:step]
```

- **start**: The index where the slice begins (inclusive). If omitted, the default is `0`.
- **end**: The index where the slice ends (exclusive). If omitted, the default is the end of the string.
- **step**: The step value, indicating how many characters to skip between each character in the slice. The default is `1`.

Examples of String Slicing

1. **Basic Slicing (Extracting Substrings)**

```
python
Copy code
text = "Hello, World!"

# Extract substring from index 0 to index 4 (inclusive)
print(text[0:5])  # Output: 'Hello'

# Extract substring from index 7 to index 11 (inclusive)
print(text[7:12])  # Output: 'World'
```

PW SKILL
DATA STRUCTURE ASSIGNENT

Explanation:

- `text[0:5]` extracts characters starting from index 0 up to but not including index 5, i.e., `"Hello"`.
- `text[7:12]` extracts characters starting from index 7 up to but not including index 12, i.e., `"World"`.

2. **Omitting Start and End Indices**

```
python
Copy code
text = "Hello, World!"

# Omit the start index, the slice begins from the
beginning of the string
print(text[:5])  # Output: 'Hello'

# Omit the end index, the slice goes till the end
of the string
print(text[7:])  # Output: 'World!'

# Omit both start and end indices, returns a copy
of the whole string
print(text[:])  # Output: 'Hello, World!'
```

Explanation:

- `text[:5]` is equivalent to `text[0:5]`, extracting the first five characters.
- `text[7:]` means start from index 7 and go to the end of the string.
- `text[:]` returns the entire string, as no bounds are specified.

PW SKILL
DATA STRUCTURE ASSIGNENT

3. **Using Negative Indices**

   Negative indices allow you to count from the end of the string. `-1` represents the last character, `-2` the second last, and so on.

```
python
Copy code
text = "Hello, World!"

# Slice the last 6 characters of the string
print(text[-6:])  # Output: 'World!'

# Extract all characters except the last one
print(text[:-1])  # Output: 'Hello, World'

# Extract all characters except the first and last
ones
print(text[1:-1])  # Output: 'Hello, World'
```

   Explanation:

   - `text[-6:]` starts from the 6th-to-last character and goes to the end, i.e., `"World!"`.
   - `text[:-1]` extracts everything except the last character.
   - `text[1:-1]` extracts everything except the first and last characters, i.e., `"ello, World"`.

PW SKILL
DATA STRUCTURE ASSIGNENT

4. **Using Step**

   The step argument allows you to skip characters in the string.

```
python
Copy code
text = "Hello, World!"

# Take every second character
print(text[::2])  # Output: 'Hoo ol!'

# Take every second character from index 1 to 8
print(text[1:9:2])  # Output: 'el,W'

# Reverse the string using a step of -1
print(text[::-1])  # Output: '!dlroW ,olleH'
```

   Explanation:

   - `text[::2]` selects every second character from the entire string, resulting in `"Hoo ol!"`.
   - `text[1:9:2]` selects every second character from index 1 to index 8, producing `"el,W"`.
   - `text[::-1]` reverses the string by starting from the end and stepping backwards

     .

5. **Empty String or Invalid Slices**

   When the `start` index is greater than or equal to the `end` index, or if the slice steps in the wrong direction, you will get an empty string.

PW SKILL
DATA STRUCTURE ASSIGNENT

```
python
Copy code
text = "Hello, World!"

# Invalid slice (start index greater than
end index)
print(text[5:2])  # Output: ''

# Invalid slice (step is negative, but start
< end)
print(text[2:5:-1])  # Output: ''
```

Explanation:

- `text[5:2]` is invalid because the start index is greater than the end index, so the result is an empty string.
- `text[2:5:-1]` is invalid because the step is negative, but the start index is smaller than the end index, so again the result is an empty string.

2. Explain the key features of lists in Python.

- In Python, lists are one of the most versatile and commonly used data structures. A **list** is an ordered, mutable collection that can store a sequence of elements, which can be of different types (e.g., integers, strings, objects). Below are the key features of Python lists:

PW SKILL
DATA STRUCTURE ASSIGNENT

1. Ordered

Lists maintain the order of elements. This means that the order in which items are added to a list is preserved, and you can access elements by their index.

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

2. Mutable

Lists are **mutable**, meaning their elements can be changed (updated, added, or removed) after the list has been created.

```
python

Copy code

my_list = [10, 20, 30]

my_list[1] = 99

print(my_list) # Output: [10, 99, 30]
```

PW SKILL
DATA STRUCTURE ASSIGNENT

3. Heterogeneous Elements

A Python list can contain elements of different types. For example, a list can hold integers, strings, and other lists.

python

Copy code

```
my_list = [10, "hello", 3.14, [1, 2, 3]]
```

4. Indexing and Slicing

Lists support **indexing** (accessing an element by its position) and **slicing** (getting a sublist of elements).

Indexing: Starts at 0 for the first element.

python

Copy code

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

print(my_list[2]) # Output: 30
```

Negative indexing allows you to access elements from the end of the list.

python

Copy code

```
print(my_list[-1]) # Output: 40 (last element)
```

Slicing allows you to extract a part of the list.

python

Copy code

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

5. Dynamic Size

Lists in Python are **dynamic**. They can grow or shrink as elements are added or removed, without needing to declare a size upfront.

Appending an element:

python

Copy code

```
my_list.append(50)
```

```
print(my_list) # Output: [10, 20, 30, 40, 50]
```

Popping an element (removes the last element by default):

python

Copy code

```
my_list.pop()
```

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

Inserting an element at a specific index:

python

Copy code

```
my_list.insert(2, 25)

print(my_list) # Output: [10, 20, 25, 30, 40]

Removing an element by value:

python

Copy code

my_list.remove(25)

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

6. Supports Iteration

Lists are iterable, which means you can loop through their elements using a for loop or other iteration methods.

```
python

Copy code

for item in my_list:

  print(item)
```

7. Contains Built-in Methods

Python lists come with a variety of built-in methods for common tasks:

len(): Returns the number of elements in the list.

PW SKILL
DATA STRUCTURE ASSIGNENT

```
python

Copy code

print(len(my_list)) # Output: 4
```

sort(): Sorts the list in place.

```
python

Copy code

my_list.sort()

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

reverse(): Reverses the list in place.

```
python

Copy code

my_list.reverse()

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

extend(): Adds all elements of another iterable (like another list) to the current list.

```
python

Copy code

my_list.extend([50, 60])

print(my_list) # Output: [40, 30, 20, 10, 50, 60]
```

copy(): Returns a shallow copy of the list.

PW SKILL
DATA STRUCTURE ASSIGNENT

python

Copy code

```
new_list = my_list.copy()
```

8. Supports Nesting

Lists can contain other lists (or any other type of object), allowing for nested structures.

python

Copy code

```
nested_list = [[1, 2], [3, 4], [5, 6]]

print(nested_list[1][0]) # Output: 3
```

9. List Comprehensions

Python supports concise and efficient ways to create lists using **list comprehensions**, which are compact syntax for generating lists.

python

Copy code

```
squares = [x**2 for x in range(5)]

print(squares) # Output: [0, 1, 4, 9, 16]
```

10. Supports Membership Testing

You can test whether an element is in a list using the `in` keyword.

PW SKILL
DATA STRUCTURE ASSIGNENT

```
python

Copy code

print(20 in my_list) # Output: True
```

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

In Python, lists are versatile data structures that allow you to store ordered collections of items. You can **access**, **modify**, and **delete** elements in a list with specific operations.

1. Accessing Elements in a List

You can access elements in a list using **indexing**. Python uses zero-based indexing, which means the first element in a list is at index 0.

Example:

```
python

Copy code

# Define a list

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

# Accessing the first element

print(my_list[0]) # Output: 10
```

PW SKILL
DATA STRUCTURE ASSIGNENT

```
# Accessing the last element (negative index counts
from the end)

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



# Accessing an element in the middle

print(my_list[2]) # Output: 30
```

2. Modifying Elements in a List

You can modify an element by assigning a new value to an existing index.

Example:

python

Copy code

```
# Define a list

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

# Modify the second element (index 1)

my_list[1] = 25

# Modify the last element using a negative index

my_list[-1] = 55

# Print the modified list

print(my_list) # Output: [10, 25, 30, 40, 55]
```

PW SKILL
DATA STRUCTURE ASSIGNENT

3. Deleting Elements in a List

You can delete elements using the del statement, the remove() method, or the pop() method.

a. Using del

del removes an element at a specified index.

Example:

python

Copy code

```
# Define a list
my_list = [10, 20, 30, 40, 50]
# Delete the third element (index 2)
del my_list[2]
# Print the list after deletion
print(my_list) # Output: [10, 20, 40, 50]
```

b. Using remove()

The remove() method deletes the first occurrence of a specific value in the list. If the value isn't found, it raises a ValueError.

Example:

python

Copy code

PW SKILL
DATA STRUCTURE ASSIGNENT

```
# Define a list

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

# Remove the element with value 30

my_list.remove(30)

# Print the list after removal

print(my_list) # Output: [10, 20, 40, 50]
```

c. Using pop()

The pop() method removes and returns the element at the specified index. If no index is provided, it removes and returns the last element.

Example:

python

Copy code

```
# Define a list

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

# Pop the last element

popped_element = my_list.pop()

# Print the popped element and the modified list

print(popped_element) # Output: 50

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

# Pop an element by index
```

```
popped_element = my_list.pop(1)

# Print the popped element and the modified list

print(popped_element) # Output: 20

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

4. Compare and contrast tuples and lists with examples.

Tuples and lists are both data structures in Python that can store multiple items. However, they have several key differences in terms of mutability, syntax, performance, and typical use cases. Below is a detailed comparison of tuples and lists, including examples.

Key Differences Between Tuples and Lists

Feature	**Tuple**	**List**
Mutability	Immutable (cannot be changed after creation)	Mutable (can be changed after creation)
Syntax	Defined with parentheses ()	Defined with square brackets []
Methods	Limited methods (e.g., count, index)	Many methods (e.g., append, remove, extend)
Performance	Faster than lists for iteration and access	Slower due to mutability and extra features

PW SKILL
DATA STRUCTURE ASSIGNENT

Feature	**Tuple**	**List**
Use case	Used for fixed data (constant data)	Used for data that needs to change or grow
Memory	More memory-efficient	Less memory-efficient due to mutability
Nested	Can contain other tuples, lists, etc.	Can contain other tuples, lists, etc.

1. Mutability:

- **Tuple**: Once a tuple is created, it cannot be modified. This means you cannot add, remove, or change elements within the tuple.
- **List**: Lists are mutable, meaning you can modify their elements, add new elements, or remove existing ones.

Example:

python

Copy code

```
# Tuple example (immutable)

my_tuple = (1, 2, 3)

# Attempting to change an element in the tuple will raise an error

# my_tuple[0] = 10  # TypeError: 'tuple' object does not support item assignment

# List example (mutable)
```

PW SKILL
DATA STRUCTURE ASSIGNENT

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

my_list[0] = 10  # Modifying an element in the list

print(my_list)   # Output: [10, 2, 3]
```

2. Syntax:

Tuple: Tuples are created using parentheses ().

List: Lists are created using square brackets [].

Example:

python

Copy code

```
# Tuple example

my_tuple = (1, 2, 3)

# List example

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

3. Methods:

- **Tuple**: Tuples have fewer built-in methods. You can only perform operations like counting and finding the index of an item.
- **List:** Lists have many more methods, such as append(), extend(), remove(), pop(), etc.

Example:

python

PW SKILL
DATA STRUCTURE ASSIGNENT

Copy code

```
# Tuple methods

my_tuple = (1, 2, 3, 3, 4)

print(my_tuple.count(3))  # Output: 2

print(my_tuple.index(3))  # Output: 2

# List methods

my_list = [1, 2, 3]

my_list.append(4)  # Adds 4 to the end of the list

my_list.remove(2)  # Removes the first occurrence of 2

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

4. Performance:

- **Tuple:** Due to their immutability, tuples are more memory-efficient and faster for iteration than lists. They are ideal for read-only data.
- **List**: Lists are slower than tuples for iteration, due to the overhead of managing dynamic resizing and mutability.

Example

(Performance):

python

Copy code

PW SKILL
DATA STRUCTURE ASSIGNENT

```
import time

# Tuple example

start_time = time.time()

my_tuple = tuple(range(1000000))

for i in my_tuple:

    pass

print("Tuple iteration time:", time.time() - start_time)

# List example

start_time = time.time()

my_list = list(range(1000000))

for i in my_list:

    pass

print("List iteration time:", time.time() - start_time)
```

In general, you'll notice that tuples perform slightly better than lists when iterating over large amounts of data.

5. Use Case:

- **Tuple**: Tuples are typically used for fixed, unchangeable collections of items. They are often used as keys in dictionaries, as they are hashable.
- **List:** Lists are more suited for data that will change over time or require frequent updates, additions, or deletions.

PW SKILL
DATA STRUCTURE ASSIGNENT

Example:

python

Copy code

Tuple example: Using a tuple as a dictionary key

my_dict = {("apple", "red"): 10, ("banana", "yellow"): 5}

List example: Using a list to store a changing collection of data

my_list = [1, 2, 3]

my_list.append(4) # Modify the list

6. **Memory Efficiency:**

- **Tuple:** Since tuples are immutable, Python can optimize their memory usage, making them more memory-efficient than lists.
- **List:** Lists are less memory-efficient due to their ability to be modified (they need extra space to accommodate possible resizing).

7. **Nested Structures**:

- Both tuples and lists can contain other tuples, lists, or other data structures. There is no restriction on this, and both are capable of storing nested structures.

- **Example:**
- python
- Copy code
- # Nested Tuple

- nested_tuple = ((1, 2), (3, 4), (5, 6))
- # Nested List
- nested_list = [[1, 2], [3, 4], [5, 6]]

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

A set is a fundamental concept in mathematics and computer science that refers to an unordered collection of distinct elements. The key features of sets are as follows:

1. Uniqueness of Elements

A set does not allow duplicate elements. Each element can only appear once in a set, meaning that repetition is automatically handled by the set.

Example:

Set: {1, 2, 3}

Trying to add 2 again results in no change: {1, 2, 3}.

2. Unordered Collection

The elements of a set do not have a specific order. The sequence in which elements are listed in a set doesn't matter.

Example:

Set: {3, 1, 2} is the same as {1, 2, 3}.

3. Mathematical Notation

PW SKILL
DATA STRUCTURE ASSIGNENT

Sets are often represented by curly braces {} and elements are listed inside, separated by commas.

Example:

- Set of integers from 1 to 5: {1, 2, 3, 4, 5}.
- Set of vowels in the English alphabet: {a, e, i, o, u}.

4. Membership

An element either belongs to a set or it does not. This is called the membership property, and it's often denoted as $x \in A$ (meaning "x is an element of set A").

Example:

If A = {2, 4, 6}, then $2 \in A$ (true), but $5 \notin A$ (false).

5. Operations on Sets

There are several basic operations that can be performed on sets, including:

Union (∪): Combines all elements from two sets, removing duplicates.

Example:

A = {1, 2, 3} and B = {3, 4, 5}, then $A \cup B$ = {1, 2, 3, 4, 5}.

- **Intersection (∩):** Returns a set of elements that are common to both sets.

Example:

  - A = {1, 2, 3} and B = {2, 3, 4}, then $A \cap B$ = {2, 3}.
- **Difference (−):** Returns a set of elements that are in the first set but not in the second.

PW SKILL
DATA STRUCTURE ASSIGNENT

Example:

- A = {1, 2, 3} and B = {2, 3, 4}, then A − B = {1}.

- **Subset (⊆):** A set A is a subset of set B if all elements of A are also elements of B.

Example:

- A = {1, 2} and B = {1, 2, 3}, then A ⊆ B (true).

- **Complement:** The complement of a set is the set of all elements in a universal set that are not in the given set.

Example:

- If the universal set is U = {1, 2, 3, 4, 5} and A = {2, 4}, then the complement of A is {1, 3, 5}.

6. Finite and Infinite Sets

- **Finite Set:** A set with a limited number of elements.
  - **Example:** {1, 2, 3} is a finite set.
- **Infinite Set:** A set with an unlimited number of elements.
  - **Example:** {1, 2, 3, 4, 5, ...} is an infinite set (the set of natural numbers).

7. Set Builder Notation

- Set builder notation is a concise way to define sets. It is used to describe a set by specifying a property that its elements must satisfy.

Example:

- A = {x | x is a positive integer less than 10} describes the set of integers {1, 2, 3, 4, 5, 6, 7, 8, 9}.

8. Cardinality

- The cardinality of a set is the number of elements in the set.
- Example:
- The cardinality of the set {1, 2, 3} is 3.

9. Power Set

- The power set of a set is the set of all subsets of that set, including the empty set and the set itself.

Example:

- If A = {1, 2}, then the power set of A is:
  P(A) = { {}, {1}, {2}, {1, 2} }.

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

In Python, **tuples** and **sets** are two important collection types that have distinct use cases. They are both used to store collections of items, but each has unique characteristics and advantages that make them suited to different scenarios. Let's dive into the use cases of tuples and sets.

1. Tuples in Python

A tuple is an ordered, immutable collection of elements, often used to store heterogeneous data. Once a tuple is created, its contents cannot be modified, which makes it different from lists (which are mutable).

PW SKILL
DATA STRUCTURE ASSIGNENT

Key Characteristics:

- **Ordered**: The elements in a tuple are indexed and have a defined order.
- **Immutable**: Once created, the elements of a tuple cannot be changed (i.e., no item can be added, removed, or replaced).
- **Hashable**: Since tuples are immutable, they can be used as keys in dictionaries or added to sets.

Use Cases for Tuples:

1. **Storing Fixed Collections of Data**:
   - Tuples are perfect for representing data that shouldn't change throughout the course of a program, such as geographical coordinates (latitude, longitude), RGB color values, or database records.
   - Example: `location = (40.7128, -74.0060)` (represents coordinates of New York City).
2. **Function Return Values**:
   - Tuples are often used to return multiple values from a function. The immutability makes them safe for returning fixed sets of data.
   - Example:

     ```
     python
     Copy code
     def get_min_max(numbers):
         return (min(numbers), max(numbers))
     ```

3. **Efficient Memory Use**:
   - Since tuples are immutable, they are often more memory-efficient than lists, making them ideal when you need to store a small, fixed-size collection of data.
4. **As Dictionary Keys**:
   - Since tuples are hashable, they can be used as keys in a dictionary, unlike lists (which are mutable and thus unhashable).
   - Example:

     ```
     python
     ```

PW SKILL
DATA STRUCTURE ASSIGNENT

```
Copy code
coordinates_dict = { (1, 2): "point A", (3, 4): "point B" }
```

5. **Unpacking**:
   - Tuples support **unpacking**, which allows you to easily assign individual elements to variables.
   - Example:

```
python
Copy code
x, y, z = (1, 2, 3)
```

6. **Protecting Data Integrity**:
   - Tuples can be used in cases where you want to prevent accidental modification of the data (e.g., configuration settings, constant values).

2. Sets in Python

A set is an unordered collection of unique elements. It is mutable, meaning items can be added or removed, but duplicates are automatically removed.

Key Characteristics:

- **Unordered**: The elements in a set do not have a specific order.
- **Unique Elements**: Sets automatically discard duplicate values.
- **Mutable**: You can add and remove items from a set.
- **Unhashable Elements**: Sets can only store hashable (immutable) elements.

Use Cases for Sets:

1. **Removing Duplicates**:
2.
   - Sets are commonly used when you need to ensure that a collection contains only unique items. Adding elements to a set automatically eliminates duplicates.
   - Example:

PW SKILL
DATA STRUCTURE ASSIGNENT

```
python
Copy code
numbers = [1, 2, 2, 3, 4, 4]
unique_numbers = set(numbers)  # {1, 2, 3, 4}
```

3. **Mathematical Set Operations**:
   - Sets support a variety of mathematical set operations, such as union, intersection, difference, and symmetric difference. These operations are efficient and can be useful in many algorithms.
   - Example:

```
python
Copy code
A = {1, 2, 3}
B = {3, 4, 5}
print(A & B)  # Intersection: {3}
print(A | B)  # Union: {1, 2, 3, 4, 5}
print(A - B)  # Difference: {1, 2}
```

4. **Fast Membership Testing**:
   - Sets offer fast membership tests, making them ideal when you need to check if an element exists in a collection, typically with an average time complexity of O(1).
   - Example:

```
python
Copy code
unique_words = {"apple", "banana", "cherry"}
"banana" in unique_words  # True
```

5. **Eliminating Duplicates in Data Structures**:
   - When processing data, you may often encounter duplicates. Using a set allows you to quickly eliminate them while maintaining the uniqueness of the data.
   - Example:

```
python
```

PW SKILL
DATA STRUCTURE ASSIGNENT

```
Copy code
log_data = ["error", "info", "error",
"warning", "info"]
unique_log_data = set(log_data)  # {"error",
"info", "warning"}
```

6. **Efficient Set Arithmetic**:
   - Sets are useful when performing operations like finding common elements (intersection), differences, or symmetric differences between collections, which can be handy in problems involving group membership, network analysis, or filtering data.
   - Example:

```
python
Copy code
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print(set1 - set2)  # Output: {1, 2} (elements
in set1 but not in set2)
```

7. **Tracking Unique Elements in Real-time**:
   - Sets are also useful when you need to keep track of unique elements during an operation, such as counting the number of unique visitors or items.
   - Example: Track unique users accessing a system:

```
python
Copy code
users = set()
users.add("user1")
users.add("user2")
users.add("user1")  # Duplicate entry, ignored
print(users)  # Output: {"user1", "user2"}
```

PW SKILL
DATA STRUCTURE ASSIGNENT

8. **Data Deduplication in Databases**:
    - When dealing with data from multiple sources or aggregating data from large datasets, sets can help in eliminating duplicates efficiently, improving data quality.

Summary Comparison:

Feature	Tuples	Sets
Mutability	Immutable	Mutable
Order	Ordered	Unordered
Duplicates	Allows duplicates	No duplicates
Use Case	Fixed collection of items, function returns, immutability, dictionary keys	Unique collections, set operations, fast membership testing, eliminating duplicates
Performance	Memory-efficient, faster iteration	Fast membership testing, efficient set operations (union, intersection, etc.)

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

Dictionaries in Python are unordered collections of key-value pairs. You can perform various operations on dictionaries, such as adding, modifying, and deleting items. Here's how you can do each of these:

PW SKILL
DATA STRUCTURE ASSIGNENT

1. Adding Items to a Dictionary

To add a new key-value pair to a dictionary, you can use the assignment operator (=). If the key doesn't exist, it will be added to the dictionary.

Example:

```
python
Copy code
# Create an empty dictionary
my_dict = {}

# Add a new key-value pair
my_dict["name"] = "Alice"
my_dict["age"] = 30

print(my_dict)
# Output: {'name': 'Alice', 'age': 30}
```

You can also add a key-value pair to an existing dictionary:

```
python
Copy code
# Add a new key-value pair to the existing dictionary
my_dict["city"] = "New York"

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

2. Modifying Items in a Dictionary

To modify an existing item, you can use the key to access the value and then assign a new value to that key.

Example:

```
python
Copy code
# Modify the value of an existing key
my_dict["age"] = 31

print(my_dict)
```

PW SKILL
DATA STRUCTURE ASSIGNENT

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

3. Deleting Items from a Dictionary

You can delete items from a dictionary using the `del` statement or the `pop()` method.

Using `del`:

The `del` statement removes a key-value pair by specifying the key.

```
python
Copy code
# Delete an item using del
del my_dict["city"]

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

Using `pop()`:

The `pop()` method removes a key-value pair and returns the value associated with the key.

```
python
Copy code
# Delete an item using pop
age = my_dict.pop("age")

print(my_dict)
# Output: {'name': 'Alice'}

print(age)
# Output: 31
```

If you try to delete a key that doesn't exist, it will raise a `KeyError`. You can avoid this by checking if the key exists first.

PW SKILL
DATA STRUCTURE ASSIGNENT

Example with check:

```
python
Copy code
key_to_delete = "city"
if key_to_delete in my_dict:
    del my_dict[key_to_delete]
else:
    print(f"Key '{key_to_delete}' not found!")
```

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

In Python, dictionary keys must be immutable because dictionaries rely on the ability to hash their keys in order to quickly locate values. The hash function computes a unique integer (a hash value) for each key, which is used to determine its position in memory. This design allows for fast lookups, additions, and deletions of key-value pairs. However, for this mechanism to work consistently and correctly, the keys themselves must be immutable, meaning their state cannot change after they've been created.

Why dictionary keys must be immutable:

1. **Hash Consistency**: When a key is added to a dictionary, Python computes its hash value and stores it in the dictionary. If the key were mutable (i.e., it could change), the hash value could change during the key's lifetime. This would make the dictionary's internal structure inconsistent, potentially causing bugs, unexpected behavior, or data corruption, as the dictionary would no longer be able to locate the key correctly.

PW SKILL
DATA STRUCTURE ASSIGNENT

2. **Efficiency**: The dictionary's lookup time depends on being able to quickly retrieve a key's hash value. If keys were mutable, every time a key is modified, its hash value would need to be recalculated and potentially relocated, which would slow down lookups.

Immutable types:

In Python, types such as strings, tuples, and numbers are immutable, so they can safely be used as dictionary keys. On the other hand, types like lists and dictionaries themselves are mutable, meaning their state can change after they are created, so they cannot be used as dictionary keys.

Examples:

Example 1: Valid dictionary with immutable keys

```
python
Copy code
# Tuple (immutable) as a key
my_dict = {('a', 1): 'value1', (2, 3): 'value2'}
print(my_dict[('a', 1)])  # Output: value1
```

Here, the tuple `('a', 1)` is immutable, so it can be used as a key. The dictionary works as expected.

Example 2: Invalid dictionary with mutable keys

```
python
Copy code
# List (mutable) as a key
my_dict = {[1, 2, 3]: 'value1'}
```

This raises a `TypeError` because lists are mutable, and Python cannot use them as dictionary keys.

Example 3: Changing an immutable key (valid)

```
python
Copy code
```

PW SKILL
DATA STRUCTURE ASSIGNENT

```
# String (immutable) key can remain unchanged
my_dict = {'name': 'John'}
print(my_dict['name'])  # Output: John
```

Example 4: Attempting to change a key (invalid scenario)

```
python
Copy code
# Modifying a mutable key will break the dictionary
lookup
my_dict = {}
key = [1, 2, 3]  # List is mutable
my_dict[key] = 'value'
key.append(4)  # Modify the list
# This will cause issues with dictionary lookup because
the key has changed
```

In this case, once the list `key` is modified (by appending an element), its hash value changes, making it effectively "disappear" or "move" in the dictionary. This can result in unpredictable behavior.