Day 13- Trainer FG
(Dictionary)

Index

I. Recap
II. What is a Dictionary?
III. Creating a Dictionary
IV. Methods of Dictionary
V. Traversing Dictionaries
VI. List v/s Dictionary

For Trainer(1.50 hrs) ILT

I. Recap

In our last session we learned:

- **What is a Set?** A set in Python is an unordered collection of unique elements. It is a built-in data type that is used to store a collection of values, with the key characteristic being that each element in a set must be unique. In other words, a set does not allow duplicate values.
- **Features of Set:** Set is an unordered collection it stores only unique elements, it is mutable and holds only unique values.
- **Creating sets:** We can create sets using several ways like using {} braces,using set constructor.
- **Various Operations of sets:** Sets in Python support various operations and methods that allow you to work with them effectively like adding,removing elements.

Try this: Predict the output
```
my_list = [1, 2, 2, 3, 4, 4, 5]
x = list(set(my_list))
print(x)
```

In this session we are going to understand What is a Dictionary,Characteristics of Dictionary,Creating a Dictionary,Methods of Dictionary,Traversing Dictionaries, List Vs Dictionary.

II. What is a Dictionary?

A dictionary is a built-in data structure in Python that allows you to store a collection of key-value pairs. It is a fundamental and versatile data type used for mapping values to their associated keys. Dictionaries are sometimes also known as "hash maps" or "associative arrays" in other programming languages.

Characteristics of Dictionary

- **Key-Value Pairs**:A dictionary is a collection of key-value pairs.
  Each key in a dictionary is unique and acts as an identifier for the associated value.
  The key and its associated value are separated by a colon (:).

- **Unordered:**Dictionaries in Python are unordered, meaning that the key-value pairs do not have a specific order.Starting from Python 3.7, dictionaries maintain insertion order, which means that the order in which key-value pairs are added is preserved in the dictionary.

- **Mutable:**Dictionaries are mutable, which means you can add, modify, or remove key-value pairs after creating a dictionary.

- **Dynamic:**Dictionaries can grow or shrink in size as you add or remove key-value pairs.

- **Flexible Key Types:**Keys in a dictionary can be of various data types, such as strings, numbers, or even tuples, as long as they are hashable (immutable).

- **Access by Key:**You can access the value associated with a key in a dictionary by using square brackets [] and providing the key.
  If the key does not exist in the dictionary, it raises a KeyError unless you use the get() method with a default value.

- **Efficient Key Lookup:**Dictionaries use a hash table under the hood, making key lookup operations highly efficient with an average time complexity of O(1).

- **No Duplicate Keys:**Each key in a dictionary must be unique. If you attempt to add a duplicate key, it will overwrite the existing value associated with that key.

- **Iteration:**You can iterate through the keys, values, or both in a dictionary using for loops and methods like keys(), values(), and items().

Here's a simple example of a dictionary:

```
student = {
    "name": "Alice",
    "age": 20,
    "major": "Computer Science"
}
```

In this dictionary, "name", "age", and "major" are keys, and "Alice", 20, and "Computer Science" are their respective values. You can access the values by specifying the keys, like student["name"] to get "Alice".

Dictionaries are a versatile and powerful data structure that plays a crucial role in Python programming for a wide range of applications.

Common Use Cases

Dictionaries are commonly used for tasks like:

- Storing and retrieving configuration settings.
- Counting occurrences of items in a collection.
- Representing data in a structured format (e.g., JSON).
- Creating look-up tables or mapping data.
- Storing information with meaningful labels.

III. Creating a Dictionary

You can create a dictionary in Python using curly braces {} or the dict() constructor. Here's an example of both methods:

Using Curly Braces {}: You can create an empty dictionary or initialize it with key-value pairs enclosed in curly braces.

```
# Creating an empty dictionary
empty_dict = {}

# Creating a dictionary with key-value pairs
student = {
    "name": "Alice",
    "age": 20,
    "major": "Computer Science"
}
```

In this example, we've created an empty dictionary empty_dict and a dictionary named student with key-value pairs representing information about a student.

Using the dict() Constructor: You can also create dictionaries using the dict() constructor, which accepts key-value pairs as arguments.

```
# Creating an empty dictionary
empty_dict = dict()

# Creating a dictionary with key-value pairs
student = dict(name="Alice", age=20, major="Computer Science")
```

In this example, we've created an empty dictionary empty_dict and a dictionary named student using the dict() constructor with key-value pairs as arguments.

IV. Methods of Dictionary
Dictionaries in Python come with several built-in methods that allow you to perform various operations on dictionary objects. Here are some of the commonly used methods for dictionaries:

clear(): Removes all key-value pairs from the dictionary, leaving it empty.

```
my_dict = {"name": "Alice", "age": 25}
my_dict.clear()  # Clears the dictionary
```

copy(): Creates a shallow copy of the dictionary.

```
original_dict = {"name": "Alice", "age": 25}
copied_dict = original_dict.copy()  # Creates a copy of original_dict
```

get(key, default=None): Returns the value associated with the specified key. If the key is not found, it returns the default value (or None if not specified).

```
my_dict = {"name": "Alice", "age": 25}
value = my_dict.get("name")  # Returns "Alice"
value = my_dict.get("address", "Unknown")  # Returns "Unknown" because "address"
key doesn't exist
```

items(): Returns a view of all key-value pairs in the dictionary.

```
my_dict = {"name": "Alice", "age": 25}
items = my_dict.items()  # Returns a view object with [("name", "Alice"), ("age", 25)]
```

keys(): Returns a view of all keys in the dictionary.

```
my_dict = {"name": "Alice", "age": 25}
keys = my_dict.keys()  # Returns a view object with ["name", "age"]
```

values(): Returns a view of all values in the dictionary.

```
my_dict = {"name": "Alice", "age": 25}
values = my_dict.values()  # Returns a view object with ["Alice", 25]
```

pop(key, default=None): Removes the key-value pair with the specified key and returns the value. If the key is not found, it returns the default value (or raises a KeyError if not specified).

```
my_dict = {"name": "Alice", "age": 25}
value = my_dict.pop("name")  # Removes the "name" key and returns "Alice"
value = my_dict.pop("address", "Unknown")  # Returns "Unknown" because "address"
key doesn't exist
```

popitem(): Removes and returns an arbitrary (key, value) pair as a tuple. Useful when you want to remove and process items from the dictionary in an unspecified order.

```
my_dict = {"name": "Alice", "age": 25}
item = my_dict.popitem()  # Removes and returns an arbitrary item, e.g., ("name",
"Alice")
```

update(iterable): Updates the dictionary with key-value pairs from another iterable (usually another dictionary or a sequence of tuples).

```
my_dict = {"name": "Alice", "age": 25}
my_dict.update({"address": "123 Main St"})
```

fromkeys(keys, value=None): Creates a new dictionary with the specified keys and an optional default value.

```
keys = ["name", "age", "address"]
default_value = "Unknown"
new_dict = dict.fromkeys(keys, default_value)
```

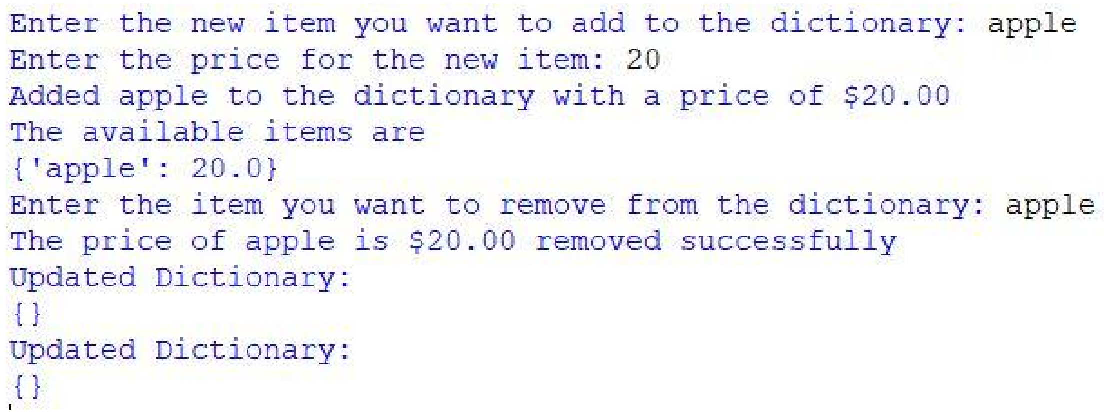
These are some of the commonly used methods for working with dictionaries in Python. Depending on your specific use case, you can choose the appropriate method to manipulate and interact with dictionary data efficiently.

Simple Python program using a dictionary

```
# Create a Empty Dictionary
item_prices = {}

# Get user input for a new item and its price to add
new_item = input("Enter the new item you want to add to the dictionary: ")
new_price = float(input("Enter the price for the new item: "))

# Use update() to add the new item and its price to the dictionary
item_prices.update({new_item: new_price})
print(f"Added {new_item} to the dictionary with a price of ${new_price:.2f}")
print("The available items are")
print(item_prices)
# Get user input for the item to remove
item_to_remove = input("Enter the item you want to remove from the dictionary: ")

# Use pop() to remove the item and get its price
if item_to_remove in item_prices:
    price = item_prices.pop(item_to_remove)
    print(f"The price of {item_to_remove} is ${price:.2f} removed successfully")
else:
    print(f"{item_to_remove} not found in the dictionary.")
```

```
# Print the updated dictionary after popping the item
print("Updated Dictionary:")
print(item_prices)

# Print the updated dictionary after adding the new item
print("Updated Dictionary:")
print(item_prices)
```

Output:



```
Enter the new item you want to add to the dictionary: apple
Enter the price for the new item: 20
Added apple to the dictionary with a price of $20.00
The available items are
{'apple': 20.0}
Enter the item you want to remove from the dictionary: apple
The price of apple is $20.00 removed successfully
Updated Dictionary:
{}
Updated Dictionary:
{}
```

Try this:
Enhance the above program to allow user to add multiple elements in the dictionary. (Hint: use while loop). The user should be allowed to exit on some condition.

V. Traversing Dictionaries

Traversing or iterating through dictionaries in Python allows you to access the keys, values, or key-value pairs within a dictionary. Python provides several methods and techniques for traversing dictionaries:

Iterating through keys: You can use a for loop to iterate through the keys in a dictionary using the keys() method.

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

for key in my_dict.keys():
    print(key)
```

Iterating through values: You can use a for loop to iterate through the values in a dictionary using the values() method.

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

for value in my_dict.values():
    print(value)
```

Iterating through Key-Value pairs: You can use a for loop to iterate through the key-value pairs in a dictionary using the items() method.

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

for key, value in my_dict.items():
    print(f"{key}: {value}")
```

These methods provide flexibility for traversing dictionaries in Python, allowing you to work with keys, values, or both based on your specific requirements. The choice of method depends on what you need to accomplish while working with the dictionary data.

VI. List v/s Dictionary

Lists and dictionaries are two fundamental data structures in Python, each with its own characteristics and use cases. Here's a comparison of lists and dictionaries:

Characteristic	List	Dictionary
Ordered / Unordered	Lists are ordered collections of elements. They maintain the order of elements based on their positions, starting from index 0.	Dictionaries are unordered collections of key-value pairs. They do not maintain any specific order of elements; instead, they use keys to access values.
Accessing Elements	You access elements in a list by their index. For example,	You access elements in a dictionary by their keys. For example, my_dict['key'] retrieves

	my_list[0] retrieves the first element in the list.	the value associated with the key 'key'.
Data Retrieval Efficiency	Retrieving an element by index in a list has a time complexity of O(1) because lists are ordered, and Python can directly calculate the memory location of the element.	Retrieving a value by key in a dictionary also has a time complexity of O(1) on average. Dictionaries use a hashtable internally to optimize key-value lookups.
Mutability	Lists are mutable, which means you can change their elements after they are created. You can add, remove, or modify elements within a list.	Dictionaries are also mutable. You can add, update, or remove key-value pairs.
Duplicates	Lists can contain duplicate elements. You can have multiple identical values in a list.	Dictionary keys must be unique. Each key can appear only once in a dictionary, and adding a new value with an existing key will overwrite the old value.
Use Cases	Lists are suitable for storing ordered collections of data where the order of elements matters, such as a sequence of items or a stack.	Dictionaries are used when you need to associate keys with values, create lookup tables, or store data in a structured, non-sequential manner. They are especially useful for tasks like data retrieval and configuration settings.
Syntax	Lists are created using square brackets, e.g., my_list = [1, 2, 3].	Dictionaries are created using curly braces or the dict() constructor, e.g., my_dict = {'key': 'value'} or my_dict = dict(key='value').

Here's a simple comparison:

```
# List
my_list = [1, 2, 3]
print(my_list[0])  # Accessing elements by index

# Dictionary
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict['name'])  # Accessing elements by key
```

In summary, lists are ordered, indexed collections of elements, while dictionaries are unordered collections of key-value pairs used for efficient data retrieval and association of values with keys. The choice between them depends on the specific requirements of your program.

Exercise:

Use GPT to write a program:

1. Hi! Can you create a program using a dictionary which provides a basic interface for managing an inventory system? which allows us to display the current inventory, add or update products, sell products, and quit the program.

After the ILT, the student has to do a 30 min live lab. Please refer to the Day 13 Lab doc in the LMS.