1- Dictionary

In Python, a **dictionary** is a collection of key-value pairs. Each key is unique and immutable (such as a string, number, or tuple), and the associated value can be any data type, including lists or even other dictionaries. Dictionaries are unordered, mutable, and indexed by keys.

Below is a detailed guide to Python dictionaries, including explanations, examples, and outputs.

1. Creating a Dictionary

Dictionaries are created by enclosing key-value pairs in curly braces {}, with each key and value separated by a colon :. Multiple key-value pairs are separated by commas.

Example:

Python Code

```
# Creating a dictionary
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

print(person)
```

Output:

```
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

Explanation:

- The dictionary person contains three key-value pairs:
  - "name" is the key, and "Alice" is the value.
  - "age" is the key, and 25 is the value.
  - "city" is the key, and "New York" is the value.

2. Accessing Dictionary Values

You can access dictionary values by referring to the key inside square brackets [] or by using the get() method.

Example 1: Accessing values with square brackets

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
print(person["name"])  # Accessing the value associated with the key 'name'
```

Output:

```
Alice
```

Example 2: Using get() method

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
```

```
age = person.get("age")
print(age)

# Accessing a non-existing key with get()
non_existing = person.get("email", "Not Found")
print(non_existing)
```

Output:

```
25
Not Found
```

Explanation:

- person["name"] retrieves the value associated with the key "name".
- get() allows you to safely access keys that may not exist, returning None or a specified default value ("Not Found" in this case).

3. Modifying Dictionary Values

You can change the value of a specific key by assigning a new value to it.

Example:

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
person["age"] = 26  # Modify the value associated with the key 'age'

print(person)
```

Output:

```
{'name': 'Alice', 'age': 26, 'city': 'New York'}
```

Explanation:

- The value associated with the key "age" is updated from 25 to 26.

4. Adding and Removing Elements

4.1. Adding New Key-Value Pairs

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

Example:

Python Code

```
person = {"name": "Alice", "age": 25}
person["email"] = "alice@example.com"  # Adding a new key-value pair

print(person)
```

Output:

```
{'name': 'Alice', 'age': 25, 'email': 'alice@example.com'}
```

4.2. Removing Key-Value Pairs

You can remove key-value pairs using pop(), del, or popitem().

Example 1: Using pop()

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
removed_value = person.pop("city")  # Remove the key 'city' and return its value

print(person)
print("Removed value:", removed_value)
```

Output:

```
{'name': 'Alice', 'age': 25}
Removed value: New York
```

Example 2: Using del

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
del person["age"]  # Remove the key 'age'

print(person)
```

Output:

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

Example 3: Using popitem()

The popitem() method removes and returns the last inserted key-value pair.

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
last_item = person.popitem()

print("Last item removed:", last_item)
print(person)
```

Output:

```
Last item removed: ('city', 'New York')
{'name': 'Alice', 'age': 25}
```

5. Checking if a Key Exists

You can check whether a key exists in a dictionary using the in keyword.

Example:

Python Code

```
person = {"name": "Alice", "age": 25}

if "name" in person:
   print("Name is present.")
else:
   print("Name is not present.")
```

Output:

```
Name is present.
```

Explanation:

- The in keyword checks if the key "name" exists in the person dictionary.

6. Looping Through a Dictionary

You can loop through a dictionary to get keys, values, or key-value pairs.

Example 1: Looping through keys

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}

for key in person:
   print(key)
```

Output:

```
name
age
city
```

Example 2: Looping through values

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}

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

Output:

```
Alice
25
New York
```

Example 3: Looping through key-value pairs

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}

for key, value in person.items():
   print(key, ":", value)
```

Output:

```
name : Alice
age : 25
city : New York
```

Explanation:

- The items() method returns key-value pairs as tuples, allowing you to iterate through them.

7. Dictionary Methods

Python dictionaries have several useful methods:

Method	Description
keys()	Returns a view object containing all the keys.
values()	Returns a view object containing all the values.
items()	Returns a view object containing key-value pairs as tuples.

Method	Description
update()	Updates the dictionary with elements from another dictionary or iterable.
clear()	Removes all elements from the dictionary.
copy()	Returns a shallow copy of the dictionary.

7.1. keys(), values(), and items()

You can retrieve all the keys, values, or key-value pairs using the keys(), values(), and items() methods, respectively.

Example:

Python Code
```
person = {"name": "Alice", "age": 25, "city": "New York"}

keys = person.keys()
values = person.values()
items = person.items()

print("Keys:", keys)
print("Values:", values)
print("Items:", items)
```
Output:
```
Keys: dict_keys(['name', 'age', 'city'])
Values: dict_values(['Alice', 25, 'New York'])
Items: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])
```

7.2. update()

The update() method allows you to add or update key-value pairs from another dictionary or an iterable of key-value pairs.

Example:

Python Code
```
person = {"name": "Alice", "age": 25}
update_data = {"city": "New York", "age": 26}  # Updating 'age' and adding 'city'

person.update(update_data)
print(person)
```
Output:
```
{'name': 'Alice', 'age': 26, 'city': 'New York'}
```

7.3. clear()

The clear() method removes all elements from the dictionary.

Example:

Python Code
```
person = {"name": "Alice", "age": 25, "city": "New York"}
person.clear()
```

```
print(person)
```

Output:

```
{}
```

7.4. copy()

The copy() method returns a shallow copy of the dictionary.

Example:

Python Code

```
person = {"name": "Alice", "age": 25, "city": "New York"}
person_copy = person.copy()

print(person_copy)
```

Output:

```
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

8. Nested Dictionaries

Dictionaries can contain other dictionaries, allowing you to store more complex, hierarchical data structures.

Example:

Python Code

```
employees = {
   "emp1": {"name": "John", "age": 30, "department": "HR"},
   "emp2": {"name": "Alice", "age": 25, "department": "Finance"},
}

print(employees)

# Accessing nested values
print(employees["emp1"]["name"])
```

Output:

```
{'emp1': {'name': 'John', 'age': 30, 'department': 'HR'}, 'emp2': {'name': 'Alice', 'age': 25,
'department': 'Finance'}}
John
```

Explanation:

- employees is a dictionary with two keys: "emp1" and "emp2". Each of these keys contains another dictionary representing employee details.

9. Dictionary Comprehension

You can create dictionaries dynamically using dictionary comprehension, which is similar to list comprehension.

Example:

Python Code

```
squares = {x: x ** 2 for x in range(1, 6)}
print(squares)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Explanation:

- The dictionary comprehension creates a dictionary where the keys are numbers from 1 to 5, and the values are their squares.

10. Merging Dictionaries (Python 3.9+)

Starting from Python 3.9, you can use the | operator to merge two dictionaries.

Example:

Python Code

```
dict1 = {"name": "Alice", "age": 25}
dict2 = {"city": "New York", "age": 26}

merged_dict = dict1 | dict2  # Merges dict1 and dict2

print(merged_dict)
```

Output:

```
{'name': 'Alice', 'age': 26, 'city': 'New York'}
```

Explanation:

- The | operator merges dict1 and dict2, with the values from dict2 taking precedence if there are duplicate keys.

Conclusion

Dictionaries in Python are powerful, flexible, and widely used data structures for storing data in key-value pairs. With their fast access, ability to handle complex data, and versatility in handling various operations, dictionaries are one of the most important tools for Python programmers.