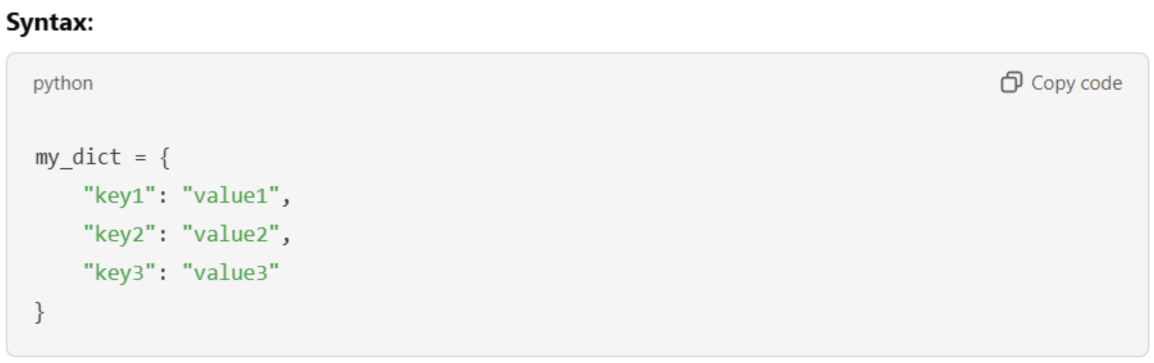
1) Dictionaries in Python

Definition:

- A **dictionary** is a collection of **key-value pairs**.
- Each key in the dictionary is **unique** and immutable (e.g., string, number, or tuple).
- Values can be of any data type and may even be lists or other dictionaries.



Syntax:

```python
my_dict = {
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

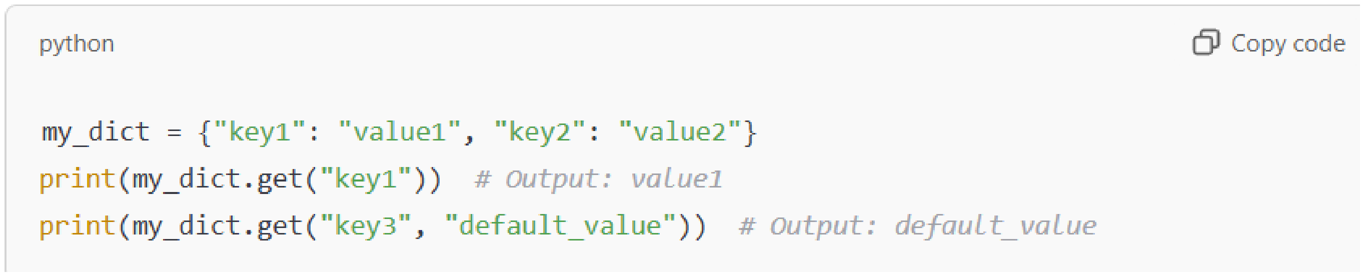
Key Points:

1. **Unordered (Python < 3.7)**: Dictionaries are unordered collections. From Python 3.7+, dictionaries maintain insertion order.
2. **Mutable**: You can add, update, or delete key-value pairs.
3. **Fast Lookup**: Dictionaries are optimized for quick key-based lookups.
4. **Nested Dictionaries**: Dictionaries can contain other dictionaries as values.

Some Important and Frequently used methods : -

1. `dict.get(key)`

- **Description**: Returns the value for the specified key. If the key does not exist, it returns `None` (or a default value if specified).
- **Syntax**: `dict.get(key, default=None)`



```python
my_dict = {"key1": "value1", "key2": "value2"}
print(my_dict.get("key1"))  # Output: value1
print(my_dict.get("key3", "default_value"))  # Output: default_value
```

The `dict.keys()` method returns a **view object** that belongs to the `dict_keys` class. This is a **dynamic view** of the dictionary's keys, which means:

Dynamic Nature:
If the dictionary is modified (e.g., keys are added or removed), the `dict_keys` object will reflect these changes immediately without needing to re-fetch the keys.
Example:
python
CODE

```
my_dict = {"key1": "value1", "key2": "value2"}
keys_view = my_dict.keys()

print(keys_view)  # Output: dict_keys(['key1', 'key2'])

my_dict["key3"] = "value3"  # Add a new key-value pair
print(keys_view)  # Output: dict_keys(['key1', 'key2',
'key3'])
```

1.
2. **Memory Efficiency**:
   Unlike lists, a `dict_keys` view does not store the keys as a separate data structure but instead references the keys in the dictionary. This saves memory compared to creating a standalone list of keys.

What Can You Do with `dict_keys`?

The `dict_keys` class supports **iteration** and **membership testing**:

Iteration: You can iterate over the `dict_keys` view just like a list:
python
CODE
```
my_dict = {"key1": "value1", "key2": "value2"}
for key in my_dict.keys():
    print(key)
# Output:
# key1
# key2
```

1.

Membership Testing: You can check if a specific key exists in the dictionary:
python
CODE
```
my_dict = {"key1": "value1", "key2": "value2"}
print("key1" in my_dict.keys())  # Output: True
print("key3" in my_dict.keys())  # Output: False
```

2.

Set-Like Operations: The `dict_keys` view supports set-like operations because the keys of a dictionary are unique:
python
CODE
```
my_dict = {"key1": "value1", "key2": "value2"}
another_dict = {"key2": "value3", "key3": "value4"}
common_keys = my_dict.keys() & another_dict.keys()  #
Intersection
print(common_keys)  # Output: {'key2'}
```

3.

How to Get a List of Keys?

If you want a **list** of keys instead of the `dict_keys` view, you can simply use the `list()` constructor:

python
CODE
```
my_dict = {"key1": "value1", "key2": "value2"}
keys_list = list(my_dict.keys())
print(keys_list)  # Output: ['key1', 'key2']
```

This converts the `dict_keys` view into a standard list, which can be indexed, sliced, and used like any other Python list.

When to Use `dict.keys()` vs `list(dict.keys())`?

- **Use `dict.keys()` when:**
    - You only need to **iterate over** the keys or check for membership.
    - Memory efficiency or dynamic updates are important.
- **Use `list(dict.keys())` when:**
    - You need a static, immutable snapshot of the keys at a particular time.

You need to use list-specific methods, like indexing or sorting:
python
CODE
```
my_dict = {"key3": "value3", "key1": "value1", "key2":
"value2"}
keys_list = list(my_dict.keys())
keys_list.sort()
```

-
  ```
  print(keys_list)  # Output: ['key1', 'key2',
  'key3']
  ```

# Back to the methods of dictionary

3. `dict.values()`

- **Description**: Returns a view object containing all the values in the dictionary.
- **Syntax**: `dict.values()`

Example:
python
CODE
`my_dict = {"key1": "value1", "key2": "value2"}`

- `print(my_dict.values())  # Output: dict_values(['value1', 'value2'])`

4. `dict.items()`

- **Description**: Returns a view object containing all key-value pairs as tuples.
- **Syntax**: `dict.items()`

Example:
python
CODE
`my_dict = {"key1": "value1", "key2": "value2"}`

- `print(my_dict.items())  # Output: dict_items([('key1', 'value1'), ('key2', 'value2')])`

5. `dict.update(other)`

- **Description**: Updates the dictionary with key-value pairs from another dictionary or iterable.
- **Syntax**: `dict.update([other])`

Example:
python

CODE

```
my_dict = {"key1": "value1"}

my_dict.update({"key2": "value2", "key3": "value3"})
```

- `print(my_dict)  # Output: {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}`

6. `dict.pop(key)`

- **Description**: Removes the specified key and returns its value. Raises a `KeyError` if the key does not exist.
- **Syntax**: `dict.pop(key, default=None)`

Example:
python
CODE

```
my_dict = {"key1": "value1", "key2": "value2"}

removed_value = my_dict.pop("key1")

print(removed_value)  # Output: value1

print(my_dict)  # Output: {'key2': 'value2'}
```

7. `dict.clear()`

- **Description**: Removes all key-value pairs, making the dictionary empty.
- **Syntax**: `dict.clear()`

Example:
python
CODE

```
my_dict = {"key1": "value1", "key2": "value2"}

my_dict.clear()
```

- `print(my_dict)  # Output: {}`

8. `dict.setdefault()`

- **Description**: Returns the value of a key if it exists, otherwise sets the key with the specified default value and returns the default.
- **Syntax**: `dict.setdefault(key, default=None)`

Example:
python
CODE
```
my_dict = {"key1": "value1"}

value = my_dict.setdefault("key2", "default_value")

print(value)  # Output: default_value
```

- `print(my_dict)  # Output: {'key1': 'value1', 'key2': 'default_value'}`

Different ways in which dict.items() is used in python :

1. Iterating Over Key-Value Pairs

- **Usage**: To iterate through a dictionary and access both keys and values simultaneously.
- **Scenario**: When you need to process or display each key-value pair.

Example:
python
CODE
```
my_dict = {"key1": "value1", "key2": "value2"}
for key, value in my_dict.items():
    print(f"Key: {key}, Value: {value}")
```

```
# Output:
# Key: key1, Value: value1
# Key: key2, Value: value2
```

-

---

2. Unpacking Key-Value Pairs in Functions

- **Usage**: To pass or unpack key-value pairs directly into functions.
- **Scenario**: When using dictionary data with a function that accepts keyword arguments.

Example:
python
CODE
```
def print_key_value(key, value):
    print(f"{key}: {value}")

my_dict = {"name": "Alice", "age": 25}
for key, value in my_dict.items():
    print_key_value(key, value)
# Output:
# name: Alice
# age: 25
```

-

---

3. Membership Testing

- **Usage**: Check for specific key-value pairs in the dictionary.
- **Scenario**: When you need to ensure both the key and value match.

Example:
python

CODE

```
my_dict = {"key1": "value1", "key2": "value2"}
print(("key1", "value1") in my_dict.items())  # Output: True
print(("key3", "value3") in my_dict.items())  # Output:
False
```

-

---

4. Converting to a List of Tuples

- **Usage**: Convert the `dict.items()` view into a list of tuples.
- **Scenario**: When you need to use the key-value pairs in a data structure that supports indexing, slicing, or sorting.

Example:
python
CODE

```
my_dict = {"key1": "value1", "key2": "value2"}
items_list = list(my_dict.items())
print(items_list)  # Output: [('key1', 'value1'), ('key2',
'value2')]
```

-

---

5. Sorting Key-Value Pairs

- **Usage**: Sort the key-value pairs by keys or values.
- **Scenario**: When you need the dictionary's data in a specific order.

Example (Sort by key):
python
CODE

```
my_dict = {"b": 2, "a": 1, "c": 3}
sorted_items = sorted(my_dict.items())
```

```
print(sorted_items)  # Output: [('a', 1), ('b', 2), ('c', 3)]
```

-

Example (Sort by value):
python
CODE
```
my_dict = {"a": 3, "b": 1, "c": 2}
sorted_by_value = sorted(my_dict.items(), key=lambda x: x[1])
print(sorted_by_value)  # Output: [('b', 1), ('c', 2), ('a', 3)]
```

-

---

6. Filtering Key-Value Pairs

- **Usage**: Use list comprehensions or filtering techniques to extract specific pairs.
- **Scenario**: When you want to work with a subset of the dictionary's data.

Example:
python
CODE
```
my_dict = {"a": 1, "b": 2, "c": 3}
filtered_items = [(k, v) for k, v in my_dict.items() if v > 1]
print(filtered_items)  # Output: [('b', 2), ('c', 3)]
```

-

---

7. Using `dict.items()` in Set Operations

- **Usage**: Treat the key-value pairs as a set of tuples for union, intersection, or difference operations.
- **Scenario**: When comparing dictionaries based on their contents.

Example:
python
CODE
```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 2, "c": 3}
common_items = dict1.items() & dict2.items()
print(common_items)  # Output: {('b', 2)}
```

-

---

7. Zipping Key-Value Pairs

- **Usage**: Create two separate iterable tuple of keys and values using unpacking.
- **Scenario**: When you want to work with keys and values separately.

Example:
python
CODE
```
my_dict = {"key1": "value1", "key2": "value2",...}
keys, values = zip(*my_dict.items())
print(keys)   # Output: ('key1', 'key2',...)
print(values) # Output: ('value1', 'value2',...)
```