Dictionary :

1. Using Curly Braces `{}`

- This is the most common way to create a dictionary.
- **Syntax:** `dictionary_name = {key1: value1, key2: value2, ...}`

Example:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
```

2. Using the `dict()` Constructor

- You can create a dictionary using the `dict()` function, which takes key-value pairs as arguments.
- **Syntax:** `dictionary_name = dict(key1=value1, key2=value2, ...)`

Example:

```
my_dict = dict(name='John', age=25, city='New York')
```

3. Creating an Empty Dictionary

- You can create an empty dictionary and then add key-value pairs to it later.
- **Syntax:** `dictionary_name = {}`

Example:

```
my_dict = {}
my_dict['name'] = 'John'
my_dict['age'] = 25
my_dict['city'] = 'New York'
```

4. Using `zip()` to Create a Dictionary

- You can create a dictionary by zipping together two lists—one containing keys and the other containing values.
- **Syntax:** `dictionary_name = dict(zip(keys_list, values_list))`

Example:

```
keys = ['name', 'age', 'city']
values = ['John', 25, 'New York']
my_dict = dict(zip(keys, values))
```

6. Using `fromkeys()` Method

- The `fromkeys()` method creates a new dictionary with keys from a specified iterable and all values set to a specified value (default is `None`).
- **Syntax:** `dictionary_name = dict.fromkeys(iterable, value)`

Example:

```
keys = ['name', 'age', 'city']
my_dict = dict.fromkeys(keys, 'Unknown')
# Output: {'name': 'Unknown', 'age': 'Unknown', 'city': 'Unkn
```

```
own'}
```

8. Creating Dictionaries from a List of Tuples

- You can convert a list of key-value tuples directly into a dictionary.
- **Syntax:** `dictionary_name = dict(list_of_tuples)`

Example:

```
my_list = [('name', 'John'), ('age', 25), ('city', 'New Yor
k')]
my_dict = dict(my_list)
```

These methods provide flexibility in creating dictionaries, whether you need an empty one, a pre-populated one, or a dynamically generated one.

Features of Dictionaries :

1. ordered Collection

- **Feature:** Dictionaries are ordered collections of items.

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
print(my_dict)
# Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

2. Mutable

- **Feature:** Dictionaries are mutable, meaning you can change, add, or remove items after the dictionary has been created.
- **Example:**

```
my_dict = {'name': 'John', 'age': 25}
my_dict['age'] = 26  # Modifying the value
my_dict['city'] = 'New York'  # Adding a new key-value pai
r
print(my_dict)
# Output: {'name': 'John', 'age': 26, 'city': 'New York'}
```

3. Unique Keys

- **Feature:** Each key in a dictionary must be unique. If you try to add a duplicate key, the new value will overwrite the existing value.
- **Example:**The key `'age'` appears twice, but only the last value (`30`) is retained.

```
my_dict = {'name': 'John', 'age': 25, 'age': 30}
print(my_dict)
# Output: {'name': 'John', 'age': 30}
```

4. Keys Must Be Immutable

- **Feature:** Keys in a dictionary must be of an immutable type, such as strings, numbers, or tuples. Mutable types like lists or other dictionaries cannot be used as keys.
- **Example:**Here, a tuple is used as a key because tuples are immutable.

```
my_dict = {('name', 'age'): 'John'}
print(my_dict)
# Output: {('name', 'age'): 'John'}
```

5. Efficient Lookup, Insertion, and Deletion

- **Feature:** Dictionaries provide average O(1) time complexity for lookups, insertion, and deletion operations, making them very efficient.
- **Example:**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
print(my_dict['name'])  # O(1) lookup
my_dict['country'] = 'USA'  # O(1) insertion
del my_dict['age']  # O(1) deletion
print(my_dict)
# Output: {'name': 'John', 'city': 'New York', 'country':
'USA'}
```

6. Dynamic Size

- **Featur**ve items. There's no need to declare the size of a dictionary ahead of time.
- **Example:**

```
my_dict = {}
my_dict['name'] = 'John'
my_dict['age'] = 25
my_dict['city'] = 'New York'
print(my_dict)
```

```
# Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

7. Heterogeneous Keys and Values

- **Feature:** A dictionary can contain keys and values of different types.
- **Example:**

```
pythonCopy code
my_dict = {'name': 'John', 'age': 25, 'is_student': True}
print(my_dict)
# Output: {'name': 'John', 'age': 25, 'is_student': True}
```

Can We Do Indexing and Slicing in a Dictionary?

- **Indexing**: Unlike lists and tuples, dictionaries do not support direct indexing because dictionaries are accessed by keys, not by numerical indices.
  - Example:

```
pythonCopy code
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict['a'])  # Output: 1
```

- **Slicing**: Dictionaries do not support slicing either because they are not sequence types like lists or tuples. You cannot access a "slice" or a range of items in a dictionary the way you would with a list.

Methods :

1. `dict.keys()`

Returns a view object that displays a list of all the keys in the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'c
ity'])
```

2. `dict.values()`

Returns a view object that displays a list of all the values in the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(my_dict.values())  # Output: dict_values(['Alice', 25,
'New York'])
```

3. `dict.items()`

Returns a view object that displays a list of key-value pairs (as tuples).

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(my_dict.items())  # Output: dict_items([('name', 'Alic
e'), ('age', 25), ('city', 'New York')])
```

4. `dict.get(key[, default])`

Returns the value for a specified key if the key is in the dictionary, otherwise it returns the default value (which is `None` if not specified).

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.get('name'))  # Output: Alice
print(my_dict.get('city', 'Unknown'))  # Output: Unknown
```

5. `dict.pop(key[, default])`

Removes the key from the dictionary and returns its value. If the key does not exist, it returns the default value (if provided), otherwise it raises a `KeyError`.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
age = my_dict.pop('age')
print(age)  # Output: 25
print(my_dict)  # Output: {'name': 'Alice'}
```

6. `dict.update([other])`

Updates the dictionary with the key-value pairs from another dictionary or iterable of key-value pairs.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
my_dict.update({'city': 'New York', 'age': 26})
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'cit
y': 'New York'}
```

7. `dict.setdefault(key[, default])`

Returns the value of a specified key. If the key does not exist, it inserts the key with the specified default value.

```
pythonCopy code
my_dict = {'name': 'Alice'}
age = my_dict.setdefault('age', 30)
print(age)  # Output: 30
print(my_dict)  # Output: {'name': 'Alice', 'age': 30}
```

8. `dict.clear()`

Removes all items from the dictionary, making it empty.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
my_dict.clear()
print(my_dict)  # Output: {}
```

9. `dict.copy()`

Returns a shallow copy of the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
new_dict = my_dict.copy()
print(new_dict)  # Output: {'name': 'Alice', 'age': 25}
```

10. `dict.fromkeys(seq[, value])`

Creates a new dictionary from a sequence of keys, with all values set to the same value (default is `None`).

```
pythonCopy code
keys = ['name', 'age', 'city']
new_dict = dict.fromkeys(keys, 'Unknown')
```

```
print(new_dict)  # Output: {'name': 'Unknown', 'age': 'Unknow
n', 'city': 'Unknown'}
```

11. `dict.popitem()`

Removes and returns the last key-value pair as a tuple. Raises `KeyError` if the dictionary is empty.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
last_item = my_dict.popitem()
print(last_item)  # Output: ('age', 25)
print(my_dict)  # Output: {'name': 'Alice'}
```

These methods are essential for performing various operations with dictionaries in Python and help with common tasks such as retrieving data, updating values, and managing the dictionary's content.

Functions :

1. `len(dict)`

Returns the number of key-value pairs (items) in the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(len(my_dict))  # Output: 3
```

2. `in` and `not in`

Checks if a key exists or does not exist in the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
print('name' in my_dict)  # Output: True
print('city' not in my_dict)  # Output: True
```

3. `any(dict)`

Returns `True` if at least one key in the dictionary is truthy, otherwise returns `False`.

```
pythonCopy code
my_dict = {'name': '', 'age': 0, 'city': 'New York'}
print(any(my_dict))  # Output: True
```

4. `all(dict)`

Returns `True` if all keys in the dictionary are truthy, otherwise returns `False`.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(all(my_dict))  # Output: True
```

5. `sorted(dict)`

Returns a sorted list of the dictionary's keys.

```
pythonCopy code
my_dict = {'b': 2, 'c': 3, 'a': 1}
print(sorted(my_dict))  # Output: ['a', 'b', 'c']
```

6. `min(dict)` and `max(dict)`

Returns the key with the smallest or largest value in the dictionary.

```
pythonCopy code
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(min(my_dict))  # Output: 'a'
print(max(my_dict))  # Output: 'c'
```

7. `reversed(dict)`

Returns a reversed iterator over the keys of the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(list(reversed(my_dict)))  # Output: ['city', 'age', 'na
me']
```

1. Membership Operators (`in` , `not in`)

These operators check if a key is present or absent in the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}

# Check if a key exists
print('name' in my_dict)  # Output: True
print('city' not in my_dict)  # Output: True
```

2. Comparison Operators (`==` , `!=`)

These operators compare two dictionaries.

- `==`: Returns `True` if both dictionaries have the same key-value pairs.

- `!=`: Returns `True` if the dictionaries differ.

```
pythonCopy code
dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 1, 'b': 2}
dict3 = {'a': 1, 'b': 3}

# Compare dictionaries
print(dict1 == dict2)  # Output: True
print(dict1 != dict3)  # Output: True
```

3. Assignment Operators (`=` , `+=` , `|=`)

- `=`: Assigns a new value to a key in the dictionary.
- `+=`: In-place addition to update a key's value.
- `|=`: Merges two dictionaries together (Python 3.9+).

```
pythonCopy code
# Assignment
my_dict = {'a': 1, 'b': 2}
my_dict['a'] = 10
print(my_dict)  # Output: {'a': 10, 'b': 2}

# In-place addition
my_dict['a'] += 5
print(my_dict)  # Output: {'a': 15, 'b': 2}

# Merging two dictionaries
my_dict |= {'c': 3}
print(my_dict)  # Output: {'a': 15, 'b': 2, 'c': 3}
```

Nested Dictionary:

1. Nested Dictionaries

A **nested dictionary** is a dictionary within a dictionary. In a nested dictionary, the values associated with keys can themselves be dictionaries, allowing you to store hierarchical or complex data structures.

Example of a Nested Dictionary:

```
# Creating a nested dictionary
student_info = {
    'student1': {
        'name': 'Alice',
        'age': 24,
        'marks': {
            'math': 88,
            'science': 92
        }
    },
    'student2': {
        'name': 'Bob',
        'age': 23,
        'marks': {
            'math': 78,
            'science': 85
        }
    }
}

# Accessing data in a nested dictionary
print(student_info['student1']['name'])     # Output: Alice
print(student_info['student2']['marks']['science'])  # Outpu
t: 85
```

In the example above:

- `student_info` is a dictionary containing two keys: `'student1'` and `'student2'`.
- Each key points to another dictionary containing information about each student.
- The `'marks'` key in each student's dictionary points to yet another dictionary containing their marks.

Modifying Nested Dictionary:

You can update or add new entries in a nested dictionary the same way you do in a regular dictionary.

```
# Updating a value in the nested dictionary
student_info['student1']['marks']['math'] = 95

# Adding a new subject to student2's marks
student_info['student2']['marks']['english'] = 87

print(student_info)
```

2. Dictionary Comprehension

Dictionary comprehension is a concise way to create dictionaries using an expression, typically involving a loop. It is similar to list comprehension but is used to construct dictionaries.

Syntax:

```
{key_expression: value_expression for item in iterable if con
dition}
```

- `key_expression`: Defines the key for each element in the new dictionary.
- `value_expression`: Defines the value associated with each key.
- `iterable`: The collection or range to iterate over.
- `condition`: (Optional) A condition that filters the items to be included in the new dictionary.

Example of Dictionary Comprehension:

```
# Creating a dictionary of squares using dictionary comprehen
sion
squares = {x: x*x for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

In the example above:

- The keys are the numbers from 1 to 5.
- The values are the squares of those numbers.

Conditional Dictionary Comprehension:

You can also include conditions within dictionary comprehension to filter items.

```
# Creating a dictionary with only even numbers and their squa
res
even_squares = {x: x*x for x in range(1, 11) if x % 2 == 0}
print(even_squares)  # Output: {2: 4, 4: 16, 6: 36, 8: 64, 1
0: 100}
```

In this case:

- Only even numbers from 1 to 10 are included in the dictionary.
- The values are the squares of those even numbers.

Example 2: Filtering values

Using dictionary comprehension with a condition to filter items.

```
original = {'apple': 2, 'banana': 5, 'cherry': 10}
filtered = {key: value for key, value in original.items() if
value > 3}
print(filtered)
```

Example 3: Transforming keys and values

Converting keys to uppercase and values to their length.

```
fruits = ['apple', 'banana', 'cherry']
fruit_lengths = {fruit.upper(): len(fruit) for fruit in fruit
s}
print(fruit_lengths)
```

Dictionary questions :

What are dictionary comprehensions, and how are they useful? Provide a scenario where dictionary comprehension is better

than traditional loops.

Answer:

- **Dictionary Comprehensions**: These allow you to construct dictionaries in a concise way using a single line of code, similar to list comprehensions.

Example:

```
squares = {x: x*x for x in range(5)}
print(squares)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- **Advantages**: They are more concise and readable than traditional loops, especially when creating or transforming dictionaries.

What are the potential issues with using mutable objects as dictionary keys?

Answer:

- **Mutable Objects**: Mutable objects like lists or sets can change after they are used as keys. Since dictionary keys must be **immutable**, using mutable objects can lead to unpredictable behavior.

For example, if a list is used as a key and later modified, its hash value may change, making it difficult for the dictionary to retrieve the associated value. This is why mutable objects are not allowed as dictionary keys.

3. Create a dictionary from two lists

```
keys = ['name', 'age', 'city']
```

```
values = ['Alice', 25, 'New York']
result = dict(zip(keys, values))
print(result)
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

4. Count the frequency of characters in a string

```
input_string = "mississippi"
char_frequency = {}
for char in input_string:
    char_frequency[char] = char_frequency.get(char, 0) + 1
print(char_frequency)
# Output: {'m': 1, 'i': 4, 's': 4, 'p': 2}
```

5. Find the most common value in a dictionary

```
from collections import Counter

input_dict = {'a': 1, 'b': 2, 'c': 1, 'd': 2, 'e': 2}
counter = Counter(input_dict.values())
most_common_value = counter.most_common(1)[0][0]
print(most_common_value)
# Output: 2
```

6. Invert a dictionary

```
input_dict = {'a': 1, 'b': 2, 'c': 3}
```

```
inverted_dict = {v: k for k, v in input_dict.items()}
print(inverted_dict)
# Output: {1: 'a', 2: 'b', 3: 'c'}
```

7. Sort a dictionary by values

```
pythonCopy code
input_dict = {'a': 3, 'b': 1, 'c': 2}
sorted_dict = dict(sorted(input_dict.items(), key=lambda ite
m: item[1]))
print(sorted_dict)
# Output: {'b': 1, 'c': 2, 'a': 3}
```

8. Remove keys with empty values from a dictionary

```
pythonCopy code
input_dict = {'a': 1, 'b': '', 'c': 3, 'd': None}
cleaned_dict = {k: v for k, v in input_dict.items() if v}
print(cleaned_dict)
# Output: {'a': 1, 'c': 3}
```

9. **Find the key with the maximum value in a dictionary**

```
pythonCopy code
input_dict = {'a': 3, 'b': 5, 'c': 1}
max_key = max(input_dict, key=input_dict.get)
```

```
print(max_key)
# Output: 'b
```

10 Check if a dictionary is a subset of another dictionary

```
pythonCopy code
def is_subset(dict1, dict2):
    return all(item in dict2.items() for item in dict1.items
())

dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 1, 'b': 2, 'c': 3}
print(is_subset(dict1, dict2))
# Output: True
```

What if we take a sub dictionary that is not present

```
def is_subset(dict1, dict2):
    return all(item in dict2.items() for item in dict1.items
())

dict1 = {'a': 1, 'b': 2}  # Subdictionary
dict2 = {'a': 1, 'b': 2, 'c': 3}  # Larger
```