Python Dictionary Methods & Functions

1. Key-Value Management

- **Methods**:
  - `keys()`: Returns a view object that displays a list of all the keys available in the dictionary.

```
my_dict = {'apple': 1, 'banana': 2, 'cherry': 3}
print(my_dict.keys())
```

Output: `dict_keys(['apple', 'banana', 'cherry'])`

  - `values()`: Returns a view object that displays the values of all entries in the dictionary.

```
print(my_dict.values())
```

Output: `dict_values([1, 2, 3])`

  - `items()`: Returns a view object that displays both keys and values as tuples for each entry in the dictionary.

```
print(my_dict.items())
```

Output: `dict_items([('apple', 1), ('banana', 2), ('cherry', 3)])`

2. Dictionary Creation

- **Methods**:
  - Direct Definition: Simply list key-value pairs between curly braces `{}`.

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

  - From a List of Tuples: Use `dict()` to create a dictionary from an iterable containing key-value pairs.

```
```

```
my_list = [('a', 1), ('b', 2), ('c', 3)]
new_dict = dict(my_list)
```

3. Dictionary Construction

- **Functions**:
  - `dict()`: Creates a dictionary from an iterable of key-value pairs or mappings.

```
my_tuple = (('a', 1), ('b', 2))
created_dict = dict(my_tuple)
```

  - `fromkeys()`: Returns a new dictionary with keys from the given sequence and values set to the specified value (default is `None`).

```
empty_sequence = []
my_dict = dict.fromkeys(empty_sequence, 'initial_value')
```

4. Dictionary Modification

- **Methods**:
  - `update()`: Allows you to add or update key-value pairs in the dictionary.

```
new_dict = {'d': 4}
updated_dict = my_dict.update(new_dict)
```

Output: None, but `my_dict` now contains `{'a': 1, 'b': 2, 'c': 3, 'd': 4}`

  - `setdefault()`: Returns the value of a key if present in the dictionary. If not, it inserts the key with the specified default value and returns that.

```
print(my_dict.setdefault('x', 5))
```

Output: None (since 'x' was not in my_dict) but adds `'x': 5` to `my_dict`

  - `pop()`: Removes a specific key from the dictionary, returning its value. If the key is not found, it raises a KeyError.

```
removed_value = my_dict.pop('b')
```

Output: None (since the dictionary structure changes) and removes `'banana': 2` pair.

5. Iteration

- **Methods & Functions**:
  - `for key in my_dict:`: Iterates through keys.

```
for key in my_dict:
    print(key, my_dict[key])
```

Output: Prints each key-value pair.

  - `for (k, v) in my_dict.items()`: Iterates through both keys and values as tuples.

```
for k, v in my_dict.items():
    print(f'{k} -> {v}')
```

6. Comparison & Searching

- **Methods**:
  - `in`: Checks if a key or value is present in the dictionary.

```
print('cherry' in my_dict)
```

Output: True, if 'cherry' exists as a key.

  - `not in`: Checks for absence of a key or value in the dictionary.

```
print('grape' not in my_dict)
```

7. Sorting

- **Methods & Functions**:

- `sorted(my_dict.keys())`, `sorted(my_dict.values())`, `sorted(my_dict.items())`: Returns sorted lists based on keys, values, or key-value pairs respectively.

```
print(sorted(my_dict.keys()))
```

Output: Sorted list of keys.

8. Advanced Dictionary Operations

- **Methods & Functions**:
  - `get()`: Similar to `setdefault()`, but can take an alternate value to return if the key is not found.

```
print(my_dict.get('e', 'no key'))
```

Output: 'no key'

  - `copy()`: Returns a shallow copy of the dictionary, which is useful for creating a new object without affecting the original.

```
copied_dict = my_dict.copy()
```

  - `clear()`: Removes all items from the dictionary, leaving it empty.

```
my_dict.clear()
```

Notes

- **Understanding Dictionary Objects**: Dictionaries in Python are mutable and unordered collections of key-value pairs. Each key is unique and can be used to look up its associated value.
- **Key Importance**: Keys must be immutable (e.g., strings, numbers, tuples) for dictionary usage but values can be any hashable type including mutable ones.
- **Use Cases**:

    - **Database Mapping**: Use dictionaries as a quick way to map keys to values, similar to database records.
    - **Configuration Management**: Store configuration settings where keys are the setting names and values are their respective values.
    - **Caching**: Implement caching mechanisms using dictionaries to store temporary data based on keys.

- **Performance**:

    - Python uses hash tables internally for dictionaries. This results in an average time complexity of O(1) for operations like insert, delete, and search under ideal conditions (no collisions).

Summary

Dictionaries are a fundamental part of Python's data structure toolkit, offering flexibility in managing key-value pairs that can be used in various applications such as database mapping, configuration storage, and caching. Understanding their methods and functions enables efficient manipulation and utilization within programs for enhanced functionality and performance.