TYPE-A:DICTIONARY:CH-13

1) Why is a dictionary termed as an unordered collection of object?

sol:

A dictionary in Python is called an unordered collection of objects because the items are stored as key-value pairs without any guaranteed order. Unlike lists or tuples, the elements in a dictionary do not have a fixed position, so you cannot rely on the order in which items were added when accessing them by key.

2) What type of objects can be used as keys in dictionaries ?

sol:

Keys in a Python dictionary must be immutable and hashable objects. Common types that can be used as keys include numbers, strings, and tuples (if the tuple contains only immutable elements). Lists, dictionaries, or other mutable objects cannot be used as dictionary keys.

3) Though tuples are immutable type, yet they cannot always be used as keys in a dictionary. What is the condition to use tuples as a key in a dictionary ?

sol:

Tuples can be used as dictionary keys only if all the elements inside the tuple are immutable. If a tuple contains a mutable element, like a list or another dictionary, it becomes unhashable and cannot be used as a key. This is because dictionary keys must be hashable.

```
d1 = {(1, 2): "valid"}        # works
d2 = {(1, [2, 3]): "invalid"}  # TypeError, list inside tuple is mutable
```

4) What all types of values can you store in :

1. dictionary-values ?
2. dictionary-keys ?

sol:

- Dictionary values can be of any type, including numbers, strings, lists, tuples, dictionaries, or even functions. There is no restriction on mutability.

- Dictionary keys must be immutable and hashable, such as numbers, strings, or tuples containing only immutable elements. Mutable objects like lists or dictionaries cannot be used as keys.

5) Can you change the order of dictionary's contents, i.e., can you sort the contents of a dictionary ?

sol:

Dictionaries in Python are unordered collections, so you cannot change the order of their contents directly. However, you can sort the dictionary's keys or values and create a new ordered representation using functions like sorted() or by converting it to an OrderedDict from the collections module.

eg:

```
d = {'b': 2, 'a': 1, 'c': 3}
for key in sorted(d):   # iterates keys in sorted order
   print(key, d[key])
```

6) In no more than one sentence, explain the following Python error and how it could arise:

TypeError: unhashable type: 'list'

sol:

This error occurs when you try to use a mutable object like a list as a dictionary key or in a set, because only hashable (immutable) objects can be used as keys.

7) Can you check for a value inside a dictionary using in operator? How will you check for a value inside a dictionary using in operator ?

sol:

The in operator checks only keys in a dictionary, not values. To check for a value, you can use the values() method with in.

eg:

```
d = {'a': 1, 'b': 2}
1 in d         # False, checks keys
1 in d.values() # True, checks values
```

8) dictionary is a mutable type, which means you can modify its contents ? What all is modifiable in a dictionary ? Can you modify the keys of a dictionary ?

sol:

A dictionary is mutable, so you can add, change, or delete values associated with keys, or remove key-value pairs entirely. However, you cannot modify existing keys themselves, because dictionary keys must remain immutable and hashable. You can only replace a key by deleting it and adding a new key-value pair.

9) How is del D and del D[<key>] different from one another if D is a dictionary ?

sol:

If D is a dictionary, del D deletes the entire dictionary from memory, making D undefined, while del D[<key>] removes only the key-value pair corresponding to <key> from the dictionary, leaving the rest of the dictionary intact.

eg:

```
D = {'a': 1, 'b': 2}
del D['a']  # D becomes {'b': 2}
del D       # D no longer exists
```

10) How is clear( ) function different from del <dict> statement ?

sol:

The clear() function removes all key-value pairs from a dictionary but keeps the dictionary object itself intact, whereas del <dict> deletes the entire dictionary object from memory, making it undefined.

eg:

```
D = {'a': 1, 'b': 2}
D.clear()  # D becomes {}, but D still exists
del D      # D no longer exists
```

11) What does fromkeys( ) method do?

sol:

The fromkeys() method creates a new dictionary with the given keys and assigns all keys the same default value (which is None if not specified).

eg:

```
keys = ['a', 'b', 'c']
d = dict.fromkeys(keys, 0)  # {'a': 0, 'b': 0, 'c': 0}
```

12) How is pop( ) different from popitem( ) ?

sol:

Feature	pop()	popitem()
Purpose	Removes and returns the value of a specified key	Removes and returns the last key-value pair as a tuple
Arguments	Requires a key (and optional default)	No arguments
Return Value	Value corresponding to the key	Tuple (key, value)
Error on Missing	Raises KeyError if key not found (unless default given)	Raises KeyError if dictionary is empty
Example	d.pop('a') → removes 'a' and returns its value	d.popitem() → removes last item

13) If sorted( ) is applied on a dictionary, what does it return ?

sol:

If sorted() is applied on a dictionary, it returns a list of the dictionary's keys in sorted order. The original dictionary remains unchanged.

eg:

```
d = {'b': 2, 'a': 1, 'c': 3}
sorted(d)   # ['a', 'b', 'c']
```

14) Will max( ) and min( ) always work for a dictionary ?

sol:

No, max() and min() applied to a dictionary work on the dictionary's keys and require the keys to be comparable. If the keys are of mixed or non-comparable types (e.g., numbers and strings), Python will raise a TypeError.

15) Can you use sum( ) for calculating the sum of the values of a dictionary ?

sol:

Yes, you can use sum() to calculate the sum of the values of a dictionary by applying it to the dictionary's .values() view.

eg:

```
d = {'a': 10, 'b': 20, 'c': 30}
total = sum(d.values())  # 60
```

16) What do you understand by shallow copy of a dictionary ?

sol:

A shallow copy of a dictionary is a new dictionary object that contains references to the same values as the original dictionary. Changes to the dictionary's top-level entries in the copy do not affect the original, but if the values are mutable objects, modifications to those values will affect both dictionaries.

17) What is the use of copy( ) function ?

sol:

The copy() function is used to create a shallow copy of a dictionary (or other mutable objects), producing a new dictionary with the same key-value pairs, so that top-level changes to the copy do not affect the original dictionary.

eg:

```
d1 = {'a': 1, 'b': 2}
d2 = d1.copy()
d2['b'] = 10    # d1 remains {'a': 1, 'b': 2}
```

18) Discuss the working of copy( ) if

(i) the values are of immutable types
(ii) the values are of mutable types

sol:

The copy() function creates a shallow copy of a dictionary, and its behavior depends on the type of values:

1. If the values are of immutable types (like numbers, strings, or tuples), the shallow copy works as expected because immutable objects cannot be changed. Modifying values in the copied dictionary does not affect the original dictionary.

2. If the values are of mutable types (like lists, dictionaries, or sets), the shallow copy only copies the references to these objects. This means that changes made to mutable values in the copy will also reflect in the original dictionary, even though the dictionary objects themselves are different.

eg:

```
# Immutable values
d1 = {'a': 10, 'b': 20}
d2 = d1.copy()
d2['b'] = 50       # d1 unaffected

# Mutable values
d3 = {'x': [1,2], 'y': [3,4]}
d4 = d3.copy()
d4['x'].append(3)   # d3['x'] is also modified
```