***18.What is a dictionary in Python, and what kind of data does it store?***

**Dictionary in Python** is a collection of keys values, used to store data values like a map, which, unlike other data types which hold only a single value as an element.

## **Example of Dictionary in Python**

Dictionary holds **key:value** pair. Key-Value is provided in the dictionary to make it more optimized.

Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}

print(Dict)

Output:

{1: 'Geeks', 2: 'For', 3: 'Geeks'}

*19.How do you create an empty dictionary in Python?*

In [Python](https://www.geeksforgeeks.org/python-programming-language/), a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by ‘comma’. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can’t be repeated and must be *immutable*.

# Creating a Dictionary

# with Integer Keys

Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}

print("\nDictionary with the use of Integer Keys: ")

print(Dict)

# Creating a Dictionary

# with Mixed keys

Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}

print("\nDictionary with the use of Mixed Keys: ")

print(Dict)

**Output:**

Dictionary with the use of Integer Keys:

{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:

{'Name': 'Geeks', 1: [1, 2, 3, 4]}

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

# ***20.How to get a value for a given key from a Python dictionary?***

In Python, you can access the value associated with a specific key in a dictionary using the key as an index. Since dictionaries are unordered collections, they do not support indexing by numerical positions like lists. Instead, you use the key to retrieve the corresponding value. Here's how you can access the value associated with a key:

```python

# Define a dictionary

person = {

'name': 'John Doe',

'age': 30,

'occupation': 'Engineer',

'email': 'john.doe@example.com'

}

# Accessing values using keys

name\_value = person['name']

age\_value = person['age']

occupation\_value = person['occupation']

email\_value = person['email']

print(name\_value) # Output: John Doe

print(age\_value) # Output: 30

print(occupation\_value) # Output: Engineer

print(email\_value) # Output: john.doe@example.com

```

Keep in mind that if you try to access a key that does not exist in the dictionary, it will raise a `KeyError`. To avoid this, you can use the `get()` method, which allows you to provide a default value if the key is not found:

```python

# Using get() method with a default value

location\_value = person.get('location', 'Unknown')

print(location\_value) # Output: Unknown (since 'location' key does not exist)

```

Using the `get()` method is helpful when you expect that a key might be missing in the dictionary, and you want to handle such cases gracefully without raising an error.

***21.Can a dictionary have duplicate keys? Explain.***

No, a dictionary in Python cannot have duplicate keys. The keys in a dictionary must be unique. If you try to add a new key-value pair with an existing key, it will overwrite the previous value associated with that key. The uniqueness of keys is a fundamental property of dictionaries, ensuring efficient retrieval and mapping of values.

If you attempt to create a dictionary with duplicate keys, the dictionary will only retain the last occurrence of each key-value pair. For example:

```python

# Creating a dictionary with duplicate keys

my\_dict = {

'name': 'John Doe',

'age': 30,

'occupation': 'Engineer',

'name': 'Jane Smith', # This will overwrite the previous 'name' key

'email': 'john.doe@example.com'

}

print(my\_dict)

```

Output:

```

{'name': 'Jane Smith', 'age': 30, 'occupation': 'Engineer', 'email': 'john.doe@example.com'}

```

As you can see, the initial 'name' key-value pair with 'John Doe' was replaced by the new 'name' key-value pair with 'Jane Smith'.

To avoid overwriting values inadvertently, it is essential to ensure that keys are unique within the dictionary. If you need to associate multiple values with a single key, you can use a list or another data structure as the value, effectively creating a dictionary of lists or a nested dictionary to handle such scenarios.

22.***How can you add a new key-value pair to an existing dictionary?***

We can add a new key-value pair to an existing dictionary in Python using either of the following methods:

1. Bracket notation: Use square brackets and the assignment operator (=) to add a new key-value pair to the dictionary.

```python

# Existing dictionary

person = {

'name': 'John Doe',

'age': 30,

'occupation': 'Engineer',

}

# Adding a new key-value pair

person['email'] = 'john.doe@example.com'

print(person)

```

Output:

```

{'name': 'John Doe', 'age': 30, 'occupation': 'Engineer', 'email': 'john.doe@example.com'}

```

2. `update()` method: Use the `update()` method to add one or more key-value pairs to the dictionary from another dictionary or an iterable of key-value pairs (e.g., list of tuples).

```python

# Existing dictionary

person = {

'name': 'John Doe',

'age': 30,

'occupation': 'Engineer',

}

# Using update() to add a new key-value pair

person.update({'email': 'john.doe@example.com'})

print(person)

```

Output:

```

{'name': 'John Doe', 'age': 30, 'occupation': 'Engineer', 'email': 'john.doe@example.com'}

```

Both methods achieve the same result of adding a new key-value pair to the dictionary. If the key already exists, the value will be updated to the new one. If the key is new, it will be added to the dictionary.

***23.How do you remove a key-value pair from a dictionary by its key?***

The [del keyword](https://www.geeksforgeeks.org/what-is-difference-between-del-remove-and-pop-on-python-lists/) can be used to in-place delete the key that is present in the dictionary in [Python](https://www.geeksforgeeks.org/python-programming-language/). One drawback that can be thought of using this is that it raises an exception if the key is not found and hence non-existence of the key has to be handled. Demonstrating key-value pair deletion using del.

# Initializing dictionary

test\_dict = {"Arushi": 22, "Mani": 21, "Haritha": 21}

# Printing dictionary before removal

print("The dictionary before performing remove is : ", test\_dict)

# Using del to remove a dict

# removes Mani

del test\_dict['Mani']

# Printing dictionary after removal

print("The dictionary after remove is : ", test\_dict)

# Using del to remove a dict

# raises exception

del test\_dict['Mani']

Output :

The dictionary before performing remove is : {'Arushi': 22, 'Mani': 21, 'Haritha': 21}

The dictionary after remove is : {'Arushi': 22, 'Haritha': 21}

Exception :

Traceback (most recent call last):

File "/home/44db951e7011423359af4861d475458a.py", line 20, in

del test\_dict['Mani']

KeyError: 'Mani'

The time complexity of initializing the dictionary and removing an item from the dictionary using the “del” statement is O(1).

The auxiliary space required for this code is O(1), as we are only modifying the existing dictionary and not creating any new data structures.

Method 2: Remove a Key from a Dictionary using pop()

The pop() can be used to delete a key and its value inplace. The advantage over using del is that it provides the mechanism to print desired value if tried to remove a non-existing dict. pair. Second, it also returns the value of the key that is being removed in addition to performing a simple delete operation. Demonstrating key-value pair deletion using pop()

Python3

# Initializing dictionary

test\_dict = {"Arushi": 22, "Anuradha": 21, "Mani": 21, "Haritha": 21}

# Printing dictionary before removal

print("The dictionary before performing remove is : " + str(test\_dict))

# Using pop() to remove a dict. pair

# removes Mani

removed\_value = test\_dict.pop('Mani')

# Printing dictionary after removal

print("The dictionary after remove is : " + str(test\_dict))

print("The removed key's value is : " + str(removed\_value))

print('\r')

# Using pop() to remove a dict. pair

# doesn't raise exception

# assigns 'No Key found' to removed\_value

removed\_value = test\_dict.pop('Manjeet', 'No Key found')

# Printing dictionary after removal

print("The dictionary after remove is : " + str(test\_dict))

print("The removed key's value is : " + str(removed\_value))

Output:

The dictionary before performing remove is : {'Arushi': 22, 'Anuradha': 21,

'Mani': 21, 'Haritha': 21}

The dictionary after remove is : {'Arushi': 22, 'Anuradha': 21, 'Haritha': 21}

The removed key's value is : 21

The dictionary after remove is : {'Arushi': 22, 'Anuradha': 21, 'Haritha': 21}

The removed key's value is : No Key found

Time Complexity: O(1)

Auxiliary Space: O(1)

Method 3: Using items() + dict comprehension to Remove a Key from a Dictionary

items() coupled with dict comprehension can also help us achieve the task of key-value pair deletion but, it has the drawback of not being an in-place dict. technique. Actually, a new dict is created except for the key we don’t wish to include. Demonstrating key-value pair deletion using items() + dict comprehension.

Python3

# Initializing dictionary

test\_dict = {"Arushi": 22, "Anuradha": 21,

"Mani": 21, "Haritha": 21}

# Printing dictionary before removal

print("The dictionary before performing\

remove is : " + str(test\_dict))

# Using items() + dict comprehension to remove a dict. pair

# removes Mani

new\_dict = {key: val for key,

val in test\_dict.items() if key != 'Mani'}

# Printing dictionary after removal

print("The dictionary after remove is : " + str(new\_dict))

Output:

The dictionary before performing remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22, 'Mani': 21}

The dictionary after remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22}

Time Complexity: O(n), where n is the length of the list test\_dict

Auxiliary Space: O(n) additional space of size n is created where n is the number of elements in the res list

Method 4: Use a Python Dictionary Comprehension to Remove a Key from a Dictionary

In this example, we will use Dictionary Comprehension to remove a key from a dictionary.

Python3

# Initializing dictionary

test\_dict = {"Arushi": 22, "Anuradha": 21, "Mani": 21, "Haritha": 21}

# Printing dictionary before removal

print("The dictionary before performing remove is : \n" + str(test\_dict))

a\_dict = {key: test\_dict[key] for key in test\_dict if key != 'Mani'}

print("The dictionary after performing remove is : \n", a\_dict)

Output:

The dictionary before performing remove is :

{'Arushi': 22, 'Anuradha': 21, 'Mani': 21, 'Haritha': 21}

The dictionary after performing remove is :

{'Arushi': 22, 'Anuradha': 21, 'Haritha': 21}

Method 5: Iterating and Eliminating

In this example, we will use a loop to remove a key from a dictionary.

Python3

# Initializing dictionary

test\_dict = {"Arushi": 22, "Anuradha": 21, "Mani": 21, "Haritha": 21}

print(test\_dict)

# empty the dictionary d

y = {}

# eliminate the unrequired element

for key, value in test\_dict.items():

if key != 'Arushi':

y[key] = value

print(y)

Output:

{'Arushi': 22, 'Anuradha': 21, 'Mani': 21, 'Haritha': 21}

{'Anuradha': 21, 'Mani': 21, 'Haritha': 21}

***24.What is the purpose of the `get()` method in dictionaries?***

Python dictionary method get() returns a value for the given key. If key is not available then returns default value None.

Syntax

Following is the syntax for get() method −

dict.get(key, default = None)

Parameters

key − This is the Key to be searched in the dictionary.

default − This is the Value to be returned in case key does not exist.

Return Value

This method return a value for the given key. If key is not available, then returns default value None.

Example

The following example shows the usage of get() method.

Live Demo

#!/usr/bin/python

dict = {'Name': 'Zabra', 'Age': 7}

print "Value : %s" % dict.get('Age')

print "Value : %s" % dict.get('Education', "Never")

When we run above program, it produces following result −

Value : 7

Value : Never

***25.Explain the difference between dictionaries and sets?***

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Dictionaries** | **Sets** |  |
| Purpose | Associative mapping of keys to values. | Store a collection of unique elements. |
| Data Structure | Key-value pairs with unique keys. | Unordered collection of unique elements. |
| Syntax | Enclosed in curly braces {}. | Enclosed in curly braces {}. |
| Key-Value Relation | Each key maps to a corresponding value. | No direct mapping, elements are unique. |
| Uniqueness | Keys must be unique within the dictionary. | Elements must be unique within the set. |
| Accessing Elements | Access values using keys (indexing by keys). | Use set operations (e.g., in keyword) for membership checks. |
| Ordering | Unordered. No guaranteed order for key-value pairs. | Unordered. No specific order for elements. |
|  |  |  |

***26.Describe sortly a set in Python and its main characteristics****.*

In Python, a set is an unordered collection of unique elements. It is a built-in data type that allows you to store a group of distinct values. Sets are denoted using curly braces `{}` and do not have duplicate elements.

Here are the main characteristics of a set in Python:

1. Uniqueness: Sets enforce uniqueness of elements, meaning each element can only appear once in the set. If you try to add a duplicate element, it won't be added again, and the set will remain unchanged.

2. Unordered: Sets are unordered, which means the elements are not stored in any specific order. Therefore, you cannot access elements in a set by index. If you need to iterate over the elements, you can use loops or other methods, but the order of iteration is not guaranteed.

3. Mutable: Sets are mutable, meaning you can add or remove elements after the set is created. You can add elements using the `add()` method or remove elements using the `remove()` or `discard()` methods.

4. Hashable Elements: Since sets use a hash table-like structure to ensure uniqueness, the elements in a set must be hashable. Hashable objects are those with a fixed hash value that does not change during their lifetime. Immutable objects like numbers, strings, and tuples are hashable and can be elements of a set.

***27.How do you create an empty set in Python?***

You can create an empty set in Python using either curly braces `{}` or the `set()` constructor. Here's how you can do it:

Using curly braces:

```python

# Method 1: Using curly braces

empty\_set = set()

print(empty\_set) # Output: set()

```

Using the `set()` constructor:

```python

# Method 2: Using set() constructor

empty\_set = set()

print(empty\_set) # Output: set()

```

Both methods will produce an empty set with no elements. Keep in mind that using just empty curly braces `{}` creates an empty dictionary, not an empty set, as dictionaries also use curly braces for their syntax. To create an empty set explicitly, it's recommended to use the `set()` constructor or `set()` function with no arguments.

***28.Can a set contain duplicate elements? Why or why not?***

No, a set cannot contain duplicate elements. The fundamental characteristic of sets in Python is that they enforce uniqueness among their elements. When you try to add an element to a set that is already present, it won't be added again, and the set remains unchanged.

The reason for this behavior lies in how sets are implemented. Internally, sets use a hash table or hash set data structure, which relies on hash codes to store and manage elements. Each element in a set must have a unique hash code. When you add elements to a set, Python calculates the hash code for each element and checks if it already exists in the set. If the hash code of the new element matches the hash code of any existing element, Python considers them duplicates and does not add the new element to maintain uniqueness.

Here's a simple example demonstrating the uniqueness of elements in a set:

```python

# Creating a set with duplicate elements

my\_set = {1, 2, 3, 2, 4, 5, 1}

print(my\_set)

# Output: {1, 2, 3, 4, 5} (No duplicates)

```

As you can see, when we initialize the set `my\_set` with duplicate elements (1 and 2), they are automatically removed, and the set only contains unique elements. This is one of the primary features of sets, making them useful for storing collections of distinct values and performing set operations efficiently.

***29.How do you add elements to a set?***

You can add elements to a set in Python using the `add()` method or the `update()` method. Both methods allow you to add elements to an existing set or create a new set if the set does not already exist. Here's how you can do it:

1. Using the `add()` method:

```python

# Creating an empty set

my\_set = set()

# Adding elements using the add() method

my\_set.add(1)

my\_set.add(2)

my\_set.add(3)

print(my\_set) # Output: {1, 2, 3}

```

2. Using the `update()` method:

```python

# Creating an empty set

my\_set = set()

# Adding elements using the update() method

my\_set.update([1, 2, 3]) # Add elements from a list

my\_set.update((4, 5, 6)) # Add elements from a tuple

my\_set.update({7, 8, 9}) # Add elements from another set

print(my\_set) # Output: {1, 2, 3, 4, 5, 6, 7, 8, 9}

```

***30.What is the purpose of the `discard()` method in sets?***

The discard() method in sets is used to remove an element from the set if it exists. If the element is not found in the set, the method does nothing, and the set remains unchanged. The primary purpose of the discard() method is to safely remove elements from a set without raising an error if the element is not present.

The syntax for using the discard() method is as follows:

python

Copy code

my\_set = {1, 2, 3, 4}

# Removing an element using the discard() method

my\_set.discard(3)

print(my\_set) # Output: {1, 2, 4}

***31.How can you perform common set operations like union, intersection, and difference?***

```python

# Create two sets

set\_a = {1, 2, 3, 4, 5}

set\_b = {4, 5, 6, 7, 8}

# Union of set\_a and set\_b (using | operator or union() method)

union\_result = set\_a | set\_b

# or union\_result = set\_a.union(set\_b)

# Intersection of set\_a and set\_b (using & operator or intersection() method)

intersection\_result = set\_a & set\_b

# or intersection\_result = set\_a.intersection(set\_b)

# Difference between set\_a and set\_b (using - operator or difference() method)

difference\_result = set\_a - set\_b

# or difference\_result = set\_a.difference(set\_b)

print("Union:", union\_result) # Output: Union: {1, 2, 3, 4, 5, 6, 7, 8}

print("Intersection:", intersection\_result) # Output: Intersection: {4, 5}

print("Difference:", difference\_result) # Output: Difference: {1, 2, 3}

```

In the code snippet above:

- The `|` operator or the `union()` method is used to perform the union of two sets, resulting in a new set containing all unique elements from both sets.

- The `&` operator or the `intersection()` method is used to perform the intersection of two sets, resulting in a new set containing the elements common to both sets.

- The `-` operator or the `difference()` method is used to perform the difference between two sets, resulting in a new set containing elements that are present in the first set but not in the second set.

These set operations are very useful in various applications, such as data analysis, finding common elements, removing duplicates, and more.

***32.Explain the concept of a frozen set. When and why would you use it?***

A frozen set in Python is an immutable version of a set. It is created using the `frozenset()` constructor and, like sets, contains a collection of unique elements. Once created, a frozen set cannot be modified (i.e., elements cannot be added, removed, or updated).

Use a frozen set when you need a collection of unique elements that should remain constant throughout your program. Since frozen sets are immutable, they can be used as elements of other sets or as keys in dictionaries. Using frozen sets in these scenarios ensures that the collection's integrity remains intact and avoids any unintended changes to the data.

In Python, are arrays a built-in data structure like lists, or do you need to import a specific module to use them?

In Python, arrays are not built-in data structures like lists. To use arrays in Python, you need to import the `array` module from the standard library, which provides an array type.

Here's how you can import and use arrays in Python:

```python

from array import array

# Creating an array of integers

my\_array = array('i', [1, 2, 3, 4, 5])

# Accessing elements in the array

print(my\_array[0]) # Output: 1

# Modifying elements in the array

my\_array[2] = 10

print(my\_array) # Output: array('i', [1, 2, 10, 4, 5])

```

Unlike lists, arrays in Python are typed and can only store elements of a specific data type, which is specified during array creation. The `array` module provides an efficient way to work with large datasets of homogeneous elements.

***33.What is the key difference between arrays and lists in Python?***

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Arrays** | **Lists** |  |
| Data Type | Homogeneous elements. | Heterogeneous elements (can store different data types). |
| Built-in | Need to import the array module. | Built-in data structure; no additional import needed. |
| Creation | Created using the array() constructor. | Created using square brackets [] or the list() constructor. |
| Efficiency | More memory efficient for large datasets of homogeneous elements. | Slightly less memory efficient due to the flexibility of storing different data types. |
| Element Access | Accessed by numeric indices (like lists). | Accessed by numeric indices (like arrays). |
| Dynamic Sizing | Arrays have a fixed size once created. You need to recreate an array to change its size. | Lists are dynamic and can change size during runtime (add or remove elements). |
| Operations | Limited set of operations compared to lists. | Offers a wide range of built-in methods and operations. |
| Example | ```from array import array |  |

***34.When would you choose to use an array instead of a list, and vice versa?***

Choose an array when you need to store a large dataset of homogeneous elements and want to optimize memory usage. Use lists when you require flexibility to store elements of different data types and need dynamic resizing during runtime. Arrays are more suitable for numerical computations, whereas lists are versatile for general-purpose data storage and manipulation in Python.