Dictionary: Properties of Dictionary Keys, Basic operations on Dictionary, Built-in Dictionary Functions & Methods.

Python Dictionary

Dictionaries are a useful data structure for storing data in Python because they are capable of imitating real-world data arrangements where a certain value exists for a given key.

The data is stored as key-value pairs using a Python dictionary.

- This data structure is mutable
- The components of dictionary were made using keys and values.
- Keys must only have one component.
- Values can be of any type, including integer, list, and tuple.

A dictionary is, in other words, a group of key-value pairs, where the values can be any Python object. The keys, in contrast, are immutable Python objects, such as strings, tuples, or numbers. Dictionary entries are ordered as of Python version 3.7. In Python 3.6 and before, dictionaries are generally unordered.

Creating the Dictionary

Curly brackets are the simplest way to generate a Python dictionary, although there are other approaches as well. With many key-value pairs surrounded in curly brackets and a colon separating each key from its value, the dictionary can be built. (:). The following provides the syntax for defining the dictionary.

Syntax:

```
1. Dict = {"Name": "Gayle", "Age": 25}
```

In the above dictionary **Dict**, The keys **Name** and **Age** are the strings which comes under the category of an immutable object.

Let's see an example to create a dictionary and print its content.

Code

```
1. Employee = {"Name": "Johnny", "Age": 32, "salary":26000,"Company":"^TCS"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
{'Name': 'Johnny', 'Age': 32, 'salary': 26000, 'Company': TCS}
```

Python provides the built-in function **dict()** method which is also used to create the dictionary.

The empty curly braces {} is used to create empty dictionary.

Code

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook'})
print("\nCreate Dictionary by using  dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(4, 'Rinku'), (2, Singh)])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output

```
Empty Dictionary:
{}

Create Dictionary by using  dict():
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}

Dictionary with each item as a pair:
{4: 'Rinku', 2: 'Singh'}
```

Accessing the dictionary values

To access data contained in lists and tuples, indexing has been studied. The keys of the dictionary can be used to obtain the values because they are unique from one another. The following method can be used to access dictionary values.

Code

```
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
```

```
4. print("Name : %s" %Employee["Name"])
5. print("Age : %d" %Employee["Age"])
6. print("Salary : %d" %Employee["salary"])
7. print("Company : %s" %Employee["Company"])
```

Output

```
ee["Company"])
Output
<class 'dict'>
printing Employee data ....
Name : Dev
Age : 20
Salary : 45000
Company : WIPRO
```

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Adding Dictionary Values

The dictionary is a mutable data type, and utilising the right keys allows you to change its values. Dict[key] = value and the value can both be modified. An existing value can also be updated using the update() method.

Note: The value is updated if the key-value pair is already present in the dictionary. Otherwise, the dictionary's newly added keys.

Let's see an example to update the dictionary values.

Example - 1:

Code

```
1. # Creating an empty Dictionary
2. Dict = {}
3. print("Empty Dictionary: ")
4. print(Dict)
5.
6. # Adding elements to dictionary one at a time
7. Dict[0] = 'Peter'
8. Dict[2] = 'Joseph'
9. Dict[3] = 'Ricky'
10. print("\nDictionary after adding 3 elements: ")
11. print(Dict)
12.
13. # Adding set of values
14. # with a single Key
```

```
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[3] = 'JavaTpoint'
print("\nUpdated key value: ")
print(Dict)
```

Output

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Example - 2:

Code

```
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....");
Employee["Name"] = input("Name: ");
Employee["Age"] = int(input("Age: "));
Employee["salary"] = int(input("Salary: "));
Employee["Company"] = input("Company:");
print("printing the new data");
print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"} Enter the details
of the new employee....
Name: Sunny
Age: 38
Salary: 39000
Company:Hcl
printing the new data
```

```
{'Name': 'Sunny', 'Age': 38, 'salary': 39000, 'Company': 'Hcl'}
```

Deleting Elements using del Keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

Code

```
1. Employee = {"Name": "David", "Age": 30, "salary":55000,"Company":"WIPRO"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Deleting some of the employee data")
6. del Employee["Name"]
7. del Employee["Company"]
8. print("printing the modified information ")
9. print(Employee)
10. print("Deleting the dictionary: Employee");
11. del Employee
12. print("Lets try to print it again ");
13. print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
{'Name': 'David', 'Age': 30, 'salary': 55000, 'Company': 'WIPRO'}
Deleting some of the employee data
printing the modified information
{'Age': 30, 'salary': 55000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined.
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

Deleting Elements using pop() Method

A dictionary is a group of key-value pairs in Python. You can retrieve, insert, and remove items using this unordered, mutable data type by using their keys. The pop() method is one of the ways to get rid of elements from a dictionary. In this post, we'll talk about how to remove items from a Python dictionary using the pop() method.

The value connected to a specific key in a dictionary is removed using the pop() method, which then returns the value. The key of the element to be removed is the only argument needed. The pop() method can be used in the following ways:

Code

```
# Creating a Dictionary
Dict1 = {1: 'JavaTpoint', 2: 'Educational', 3: 'Website'}
# Deleting a key
# using pop() method
pop_key = Dict1.pop(2)
print(Dict1)
```

Output

```
{1: 'JavaTpoint', 3: 'Website'}
```

Additionally, Python offers built-in functions popitem() and clear() for removing dictionary items. In contrast to the clear() method, which removes all of the elements from the entire dictionary, popitem() removes any element from a dictionary.

Iterating Dictionary

A dictionary can be iterated using for loop as given below.

Example 1

Code

```
# for loop to print all the keys of a dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee:
    print(x)
```

Output

```
Name
Age
salary
Company
```

Example 2

Code

```
#for loop to print all the values of the dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"} for x in Emplo
yee:
 print(Employee[x])
```

Output

```
John
29
25000
```

```
WIPRO
```

Example - 3

Code

```
#for loop to print the values of the dictionary by using values() method.
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee.values():
    print(x)
```

Output

```
John
29
25000
WIPRO
```

Example 4

Code

```
#for loop to print the items of the dictionary by using items() method
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee.items():
    print(x)
```

Output

```
('Name', 'John')
('Age', 29)
('salary', 25000)
('Company', 'WIPRO')
```

Properties of Dictionary Keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

Code

```
Employee={"Name":"John","Age":29,"Salary":25000,"Company":"WIPRO","Name":
"John"}
for x,y in Employee.items():
        print(x,y)
```

Output

```
Name John
Age 29
Salary 25000
Company WIPRO
```

2. The key cannot belong to any mutable object in Python. Numbers, strings, or tuples can be used as the key, however mutable objects like lists cannot be used as the key in a dictionary.

Consider the following example.

Code

```
1. Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO",[100,201,301]:"
   Department ID"}
2. for x,y in Employee.items():
3.     print(x,y)
```

Output

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in
    Employee          =          {"Name":          "John",          "Age":          29,
"salary":26000,"Company":"WIPRO",[100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

Built-in Dictionary Functions

A function is a method that can be used on a construct to yield a value. Additionally, the construct is unaltered. A few of the Python methods can be combined with a Python dictionary.

The built-in Python dictionary methods are listed below, along with a brief description.

- **len()**

The dictionary's length is returned via the len() function in Python. The string is lengthened by one for each key-value pair.

Code

```
1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
2. len(dict)
```

Output

```
4
```

- **any()**

Like how it does with lists and tuples, the any() method returns True indeed if one dictionary key does have a Boolean expression that evaluates to True.

Code

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
any({'':'','':'','3':''})
```

Output

```
True
```

- **all()**

Unlike in any() method, all() only returns True if each of the dictionary's keys contain a True Boolean value.

Code

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
all({1:'',2:'','':''})
```

Output

```
False
```

- **sorted()**

Like it does with lists and tuples, the sorted() method returns an ordered series of the dictionary's keys. The ascending sorting has no effect on the original Python dictionary.

Code

```
dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
sorted(dict)
```

Output

```
[ 1, 5, 7, 8]
```

Built-in Dictionary methods

The built-in python dictionary methods along with the description and Code are given below.

- **clear()**

It is mainly used to delete all the items of the dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# clear() method
dict.clear()
print(dict)
```

Output

```
{ }
```

- **copy()**

It returns a shallow copy of the dictionary which is created.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# copy() method
dict_demo = dict.copy()
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

- **pop()**

It mainly eliminates the element using the defined key.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# pop() method
dict_demo = dict.copy()
x = dict_demo.pop(1)
print(x)
```

Output

```
{2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

popitem()

removes the most recent key-value pair entered

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# popitem() method
dict_demo.popitem()
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}
```

- **keys()**

It returns all the keys of the dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# keys() method
print(dict_demo.keys())
```

Output

```
dict_keys([1, 2, 3, 4, 5])
```

- **items()**

It returns all the key-value pairs as a tuple.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# items() method
print(dict_demo.items())
```

Output

```
dict_items([(1, 'Hcl'), (2, 'WIPRO'), (3, 'Facebook'), (4, 'Amazon'), (5, 'Flipkart')])
```

- **get()**

It is used to get the value specified for the passed key.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# get() method
print(dict_demo.get(3))
```

Output

```
Facebook
```

- **update()**

It mainly updates all the dictionary by adding the key-value pair of dict2 to this dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# update() method
dict_demo.update({3: "TCS"})
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'TCS'}
```

- **values()**

It returns all the values of the dictionary with respect to given input.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# values() method
print(dict_demo.values())
```

Output

```
dict_values(['Hcl', 'WIPRO', 'TCS'])
```

dictionary keys have several important properties:

1. **Uniqueness**: Dictionary keys must be unique. If you try to insert a key that already exists, it will overwrite the existing key's value. This ensures that each key is associated with only one value.

2. **Immutability**: Dictionary keys must be immutable. This means that they cannot be changed after they are created. Common immutable types that can be used as keys include integers, strings, tuples (if they contain only immutable elements), and other immutable built-in types.

3. **Hashability**: Dictionary keys must be hashable. Hashability is closely related to immutability. It means that the object's hash value should remain constant throughout its lifetime. Mutable types like lists cannot be used as dictionary keys because they can change, and thus their hash value can change as well. Immutable types have a fixed hash value, making them suitable for use as keys in dictionaries.

4. **Ordering (Python 3.7+)**: Prior to Python 3.7, dictionaries did not preserve the order of their keys.