எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு ... *புரட்சிக்கவி*

NOTICE

www.DataScienceInTamil.com

Day 13, 14, 15,16, 17 - Batch 3 - Python Language

Chapter 10 – Dictionary (Collection Data Type)

To watch the recorded Python and Data Science videos in YouTube:

Day 13- Batch 3 - Dictionary - Python Collection Data Type - Part 1

https://youtu.be/r0juRMqroCc

Day 14- Batch 3 - Dictionary - Python Collection Data Type - Part 2

https://youtu.be/YgEt37uT89U

Day 15- Batch 3 - Dictionary - Python Collection Data Type - Part 3

https://youtu.be/_ensdQ06qvw

Day 17- Batch 3 - Dictionary - Python Collection Data Type - Part 4

https://youtu.be/G0GRKQqV5wA

Day 18- Batch 3 - Dictionary - Python Collection Data Type - Part 5

https://youtu.be/b7zqLXkep5A

Official Website:

https://DataScienceInTamil.com/

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

https://www.DatascienceInTamil.com/#faq

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

https://t.me/joinchat/lUZEsr-zidpjZjEx

To Join the class, please fill the form :

https://forms.gle/QFpLHwAoinFaX2cE6

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVpTzZ2S1Vsdz09

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language

- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.
- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.
- All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content /code example, definitions, etc., please use these websites for further reading:

- Book : Python Notes For Professionals
- https://www.w3schools.com
- https://www.geeksforgeeks.org
- https://docs.python.org
- https://www.askpython.com
- https://docs.python.org
- https://www.programiz.com
- https://www.programiz.com/
- https://www.openriskmanagement.com/
- https://pynative.com/python-sets/
- https://www.alphacodingskills.com/
- https://codedestine.com/
- https://appdividend.com/
- https://freecontent.manning.com/

WHAT TO COVER TODAY IN DICTIONARY - DICT?

1. Introduction to Dictionary/Dict
2. Normal Declaration of Dict
3. Empty Dict creation
4. Key Value in a Dict
5. The dict() constructor
6. keyword arguments // dict()
7. Passing in a list of key/value pairs
8. Don't try to create Dict using list of sets - why
9. Don't try create set of tuples in Dict - why
10. Add , access, modify and del a value from the Dict
11. Different way of initiating a dictionary
12. Different ways of Creating a dictionary
13. Updating a value in Dict
14. All combinations of dictionary values
15. Membership operator in Dict
16. Methods in Dict
17. Modifying a Dict

18. Change Values in a Dictionary
19. Copy() a Dict
20. Copy using =
21. Copy using **
22. Copy using ** / shallow copy
23. How copy() works in Dict
24. Copy -Same keys in 2 dicts
25. Copy - Different keys in 2 dict
26. Merging dictionaries
27. How to add 2 or more dictionaries? Using **
28. Update a dict
29. Copy() vs update()
30. Del in dict
31. Using Sorted() in dict
32. Using reversed() – it gives iterator object
33. Iterating Over a Dictionary
34. Accessing keys and values
35. Unpacking dictionaries using the ** operator

36. clear()
37. Popitem()
38. Pop()
39. Avoiding KeyError Exceptions
40. Dictionary with default values
41. Setdefault()
42. Diff between get() and setdefault()
43. Use Setdefault ()to add a list of values
44. How to convert a single value to list of values
45. check if a key is in the dictionary
46. Collections - Counter in Dict
47. Use of counter in a sequence

Introduction to Dictionary /Dict

A dictionary is an example of a **key value store also known as Mapping in Python**. It allows you to store and retrieve elements by referencing a key. As dictionaries are referenced by key, **they have very fast lookups.**

As they are primarily used for referencing items by key, **they are not sorted (ie key and value together cannot be sorted. But using sorted(), only keys can be sorted)**

NOTE: Because a dict is unsorted, keys(), values(), and items() have no sort order. **Use sort() in list, sorted() -built in method, or an OrderedDict** if you care about the order that these methods return

Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

1. A dictionary is an ordered collection of key-value pairs.
2. A dictionary has a length, specifically the number of key-value pairs.
3. A dictionary provides fast look up by key.
4. The keys must be immutable object types (as set)

Strings and integers work as keys, since they are immutable

Normal Declaration of Dict

```
d=  {'name': 'Anna University', 'Location': 'Chennai'}
print(d)
```

output

```
{'name': 'Anna University', 'Location': 'Chennai'}
```

```
d=  {'name': 'Anna University', 'Location': 'Chennai', "VC":"Rajagopal", "pincode": 600005,
"marks": 96.5, 20:1}
print(d)
print(type(d))
print(id(d))
print(len(d))
```

Empty Dict creation

```
Bank = dict()
print(Bank)
```

o/p:

{}

Different ways of Creating a dictionary

1. Key - Value in a Dict
2. keyword arguments // dict()
3. Passing a list of tuples - key/value pairs

Key - Value in a Dict

```
d = {"Bank": "SBI"}
print(d)
```

output

{'Bank': 'SBI'}

---

keyword arguments

```
bank = dict(Bankbranch = 'SBI', IFSC = "AAA")
print(bank)
```

Important note: when we use keyword arguments, the key should not have the double quote

```
bank = {"Bankbranch" : 'SBI', "IFSC":  "AAA"}
print(bank)
```

Output

```
{'Bankbranch': 'SBI', 'IFSC': 'AAA'}
{'Bankbranch': 'SBI', 'IFSC': 'AAA'}
name_Marks = { "Grade" = "First"} # this is  not right way of declaringn keyword arguments
```

Passing a list of tuples - key/value pairs

```
dict1 = dict([('SBI', 'Location'), ('IFSC', 'ABC1234')])
print(dict1)
```

o/p:
{'SBI': 'Location', 'IFSC': 'ABC1234'}

Don't try to create Dict, using list of sets - why

Note1:**Don't try to create dict using list of sets**, as it creates unexpected key – value pair of dict(as set does not have order)

```
dict1 = dict(    [ {'SBI',"Bank"}, {'IFSC', 'ABC1234'} ]       )
print(dict1)
```

Output
{'SBI': 'Bank', 'IFSC': 'ABC1234'}
See below the key and value has interchanged
{'Bank': 'SBI', 'ABC1234': 'IFSC'}

Don’t try to create set of tuples in Dict - why

Note2: **Don’t try create set of tuples,** cos’ the order of ITMES /pair will not be maintained , the dict will not in order

See below

```
dict1 = dict(    { ('SBI',"Bank"), ('IFSC', 'ABC1234') }       )
print(dict1)
```

output

{'SBI': 'Bank', 'IFSC': 'ABC1234'}

Anther execution the order of items has changed
{'IFSC': 'ABC1234', 'SBI': 'Bank'}

Notes

Dictionary creating types	Outputs
1.Creating the dictionary using List of TUPLES	O/P – The Output can be in the Correct form, A new Dictionary is created
2.Creating the dictionary using List of SETS	O/P – We will get O/P without syntax error but the KEY and VALUE will be interchanged.
3.Creating the dictionary using SET of TUPLES	O/P—We will get the O/P without any error but the Order of the result will be Interchanged

==========

```
bank = { ('SB', "FD"): [5000, 3000], 'Location': 'Ngm', "Nithya": ["Math", "Phy", "Che"], 56.7 : 30,
5 : 5, frozenset(): "AA"}
print(bank)
```

output

{('SB', 'FD'): [5000, 3000], 'Location': 'Ngm', 'Nithya': ['Math', 'Phy', 'Che'], 56.7: 30, 5: 5, frozenset(): 'AA'}

- A comma after the last pair is optional //
  d3 = {1: 'RED', 2: 'GREEN', 3: 'BLUE', }
- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries are like lists except that they use keys instead of numbers to look up values
- We can use "[ ]" to get a
  1. key's value or
  2. add a new key + value to the dict
  3. or set a new value to existing key or
  4. delete an item(Key +value)
- We can use the d.items() method to retrieve a list of **tuples** containing key-value pairs
- Each pair /
  item in the dictionary is represented by a key and value separated by a colon . Multiple pairs are separated by commas

Dictionary consists of key-value pairs. It is enclosed by curly braces {} and values can be assigned and accessed using square brackets[].

- An ordered collection of unique key-value pairs;
- keys must be hashable (tuple, frozenset)
- Key must be unique / duplicate key are not allowed , if so it takes last key's value
- Keys can be sorted and reversed – using reversed()

Add and access a value from the Dict

To get a value, refer to it by its key

Different ways of initiating a dictionary

```
dic={'name':'red','age':10}
print(dic)    #will output all the key-value pairs. {'name':'red','age':10}
print(dic['name'])    #will output only value with 'name' key. 'red'
print(dic.values())    #will output list of values in dic. ['red',10]
print(dic.keys())    #will output list of keys. ['name','age']
```

To the above the examples are below

```
bank = { ('SB', "FD"): [5000, 3000], 'Location': 'Ngm', "Nithya": ["Math", "Phy", "Che"], 56.7 : 30,
5 : 5, frozenset(): "AA"}
print(bank)
print("==========")

print(bank["Location"]) # use dictName[key] to access the value
print("==========")

print(bank.keys())
print(type(bank.keys()))
print("==========")

print(bank.values())
print(type(bank.values()))
print("==========")

print(bank.items())
print(type(bank.items()))
print("===========")
```

output

{('SB', 'FD'): [5000, 3000], 'Location': 'Ngm', 'Nithya': ['Math', 'Phy', 'Che'], 56.7: 30, 5: 5, frozenset(): 'AA'}

=========

Ngm

=========

dict_keys([('SB', 'FD'), 'Location', 'Nithya', 56.7, 5, frozenset()])

<class 'dict_keys'>

=========

dict_items([(('SB', 'FD'), [5000, 3000]), ('Location', 'Ngm'), ('Nithya', ['Math', 'Phy', 'Che']), (56.7, 30), (5, 5), (frozenset(), 'AA')])

<class 'dict_items'>

==========

dict_values([[5000, 3000], 'Ngm', ['Math', 'Phy', 'Che'], 30, 5, 'AA'])

<class 'dict_values'>

=========

Set / add new item to the dict

```
dic1= {'name': 'Anna University', 'Location': 'Chennai'}
dic1["State"] = "Tamilnadu"
print(dic1)
```

output

{'name': 'Anna University', 'Location': 'Chennai', 'State': 'Tamilnadu'}

```
bank = { ('SB', "FD"): [5000, 3000], 'Location': 'Ngm', "Nithya": ["Math", "Phy", "Che"], 56.7 : 30,
5 : 5, frozenset(): "AA"}
fz = ["New Customer", "Native" ]
bank[  frozenset(fz)] = "Linda"
print(bank)
```

output

{('SB', 'FD'): [5000, 3000], 'Location': 'Ngm', 'Nithya': ['Math', 'Phy', 'Che'], 56.7: 30, 5: 5, frozenset(): 'AA', frozenset({'New Customer', 'Native'}): 'Linda'}

Get values

```
dic1= {'name': 'Anna University', 'Location': 'Chennai'}
a = dic1["Location"]
print(a)
```

output

Chennai

Creating a dictionary using fromkeys()

```
d1 = dict.fromkeys("Name", "Age")
print(d1)

d1 = dict.fromkeys( ( "Name", "Age")  ) # ( "Name", "Age")  is treated as one whole KEY, but value is not given, so it gives None
print(d1)

#
# d1 = dict.fromkeys(5323, "Age")#TypeError: 'int' object is not iterable
```

```
# print(d1)

d1 = dict.fromkeys("5323", "number")
print(d1)
print("===============")

dictionary = dict.fromkeys(   ("milkPrice", "eggPrice"),    ("Rs10"))# string value
print(dictionary)

dictionary = dict.fromkeys(   ("milkPrice", "eggPrice"), (10))  # integer value
print(dictionary)
print("===============")


dictionary = dict.fromkeys( ("milkPrice",), ("Rs10", "Rs20") ) # ("Rs10", "Rs20") is treated as
one whole value
print(dictionary) # note the comma and it makes the tuple NOT the string


dictionary = dict.fromkeys( ("milkPrice", "eggPrice"), ("Rs10", "Rs20") ) # ("Rs10", "Rs20") is
treated as one whole value (STRING)
print(dictionary)

dictionary = dict.fromkeys(("milkPrice", "eggPrice"), (10,20)) # (10,20) is treated as one whole
key (INT)
print(dictionary)
print("===============")
```

```
dictionary = dict.fromkeys( (range(5)), (10,20))  # (10,20) is treated as one whole key
print(dictionary)
```

Note: since range creates an iterable it works fine

output

```
{'N': 'Age', 'a': 'Age', 'm': 'Age', 'e': 'Age'}
{'Name': None, 'Age': None}
{'5': 'number', '3': 'number', '2': 'number'}
===============
{'milkPrice': 'Rs10', 'eggPrice': 'Rs10'}
{'milkPrice': 10, 'eggPrice': 10}
===============
{'milkPrice': ('Rs10', 'Rs20'), 'eggPrice': ('Rs10', 'Rs20')}
{'milkPrice': (10, 20), 'eggPrice': (10, 20)}
==============
------------
```

Note: the same values go to all keys. Since the (10, 20) is being tuple the (10,20) is added to each key as its value

---

```
d1 = dict.fromkeys(   {'a', 'e', 'i', 'o', 'u' },[10, 20]    )# set , so the keys are unordered (Keys are separated
```

```
print(d1)

d1 = dict.fromkeys(   ("Tamil", "English", "Maths"  ), [10, 20, 30]    ) #  Keys are separated NOT VALUES
print(d1)

d1 = dict.fromkeys(   "Tamil", [10, 20, 30]    ) #  Keys are separated NOT VALUES
print(d1)

d1 = dict.fromkeys(   ["Tamil", "English"], [10, 20, 30]    ) #  Keys are separated NOT VALUES
print(d1)
```

output

{'u': [10, 20], 'o': [10, 20], 'e': [10, 20], 'a': [10, 20], 'i': [10, 20]}

{'Tamil': [10, 20, 30], 'English': [10, 20, 30], 'Maths': [10, 20, 30]}

{'T': [10, 20, 30], 'a': [10, 20, 30], 'm': [10, 20, 30], 'i': [10, 20, 30], 'l': [10, 20, 30]}

{'Tamil': [10, 20, 30], 'English': [10, 20, 30]}

Updating value in Dict

(ie old value will be replaced by new value)

```
d = {"Name" : "Ashw"}
print(d)
print(d["Name"]) # getting a key value

d  ["Lin"] = "Linda" # setting a new key and value to the dict
print(d)

d["Lin"] = "Sudha" # updating a new value to old value
print(d)
```

Updating a value in the dict's list

value.append(2)#the append(), appending 2 to value = [1] at the same memory location

```
keys = {'a', 'e', 'i', 'o', 'u' }
value = [1]
print(dict.fromkeys(keys, value))
value.append(2)
print(value)
print(dict.fromkeys(keys, value))
```

output

```
{'e': [1], 'o': [1], 'i': [1], 'u': [1], 'a': [1]}
[1, 2]
{'e': [1, 2], 'o': [1, 2], 'i': [1, 2], 'u': [1, 2], 'a': [1, 2]}
```

Membership operator in Dict

The **in** operator tests for the existence of a key in a dictionary

```
dic1= {'name': 'Anna University', 'Location': 'Chennai'}
print("Location" in dic1)
```

```
dic1= {'name': 'Anna University', 'Location': 'Chennai'}
print("Chennai" in dic1)
```

```
print(dic1["Location"]=="Chennai")
```

output

True

False

True

Another code for in operator

```
print("Chennai" in dic1.values())
```

output

True

We can use "[ ]" to get the following

1. key's value or
2. add a new key + value to the dict

3. or set a new value to existing key or
4. delete an item(Key +value)

```
# The in operator tests for the existence of a key in a dictionary
dic1=  {'name': 'Anna University', 'Location': 'Chennai'}

print(dic1["Location"])
print(dic1)

dic1["pincode"] = 60005
print((dic1))

dic1["pincode"] = 600006
print(dic1)

del(dic1["pincode"])
print(dic1)
```

output

```
Chennai
{'name': 'Anna University', 'Location': 'Chennai'}
{'name': 'Anna University', 'Location': 'Chennai', 'pincode': 60005}
{'name': 'Anna University', 'Location': 'Chennai', 'pincode': 600006}
```

{'name': 'Anna University', 'Location': 'Chennai'}

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Modifying a Dict

modifying a dict

To add items to a dictionary, simply create a new key with a value:

```
d['newkey'] = 42
```

It also possible to add `list` and `dictionary` as value:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

To delete an item, delete the key from the dictionary:

```
del d['newkey']
```

```
d = dict()
d['List of Numbers'] = [1,2,3]
d['new_dict'] = {'nested_dict':1, "Marks": 98.5}
print(d)
print(type(d))
```

output

{'List of Numbers': [1, 2, 3], 'new_dict': {'nested_dict': 1, 'Marks': 98.5}}

<class 'dict'>

Change Values to a key in a Dictionary

```
dic = {"Name" : "AAA", "Subject" : "Python"}
dic["Subject"] = "Grammar"
print (dic)
```

output

```
Grammar
{'Name': 'AAA', 'Subject': 'Grammar'}
```

```
print(dict)
print(set)
print(tuple)
print(list)
```

output

<class 'dict'>

<class 'set'>

<class 'tuple'>

<class 'list'>

```
print(dict())
print(set())
print(tuple())
print(list())
```

output

{}

set()

()

[]

To add items to a dictionary, simply create a new key with a value:

It also possible to add list and dictionary as value (Or any container datatype)

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
lst1=[1,2,3]  #DECLARING A LIST
dic1["Sl_Number"] = lst1  # MODIFYING THE EXISTING KEY VALUE IN THE DICTIONARY BY SUPPLYING A "LIST "INSTEAD SINGLE VALUE
print (dic1)
```

output

```
{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': [1, 2, 3]}
```

==========

How to copy a dict -There 3 ways

1. **Copy a Dict using()// dict.copy()**
2. **Copy using =**
3. **Copy using ** / shallow copy {**original dict name}**

Copy a Dict using copy()

Copy function (copy function removes the existing value and add the new values, if we try to copy a dict to another dict, ie we have 2 dicts)

```
d1 = {1 : 'one', 2 : 'two'}
print(id(d1))

d2 = {3 : 'three'} # that is the d2 items become empty and it stores the items of d1
print("id of d2  before copy", id(d2))

d2 = d1.copy()
print (d2)
print("id of d2 after applying copy() ", id(d2))
```

output

```
11152912
{1: 'one', 2: 'two'}
11153312
---------------
```

Copy using =

original dict WILL be modified when we copying using =

```
bank = {'name': 'SBI', 'Location': 'Tanjore'}
print('originalBankdict \n', bank)
```

```
print('*'*30)

bank1= bank
print('original dict is copied to bank1\n', bank1)
print('*'*30)

bank1['IFSC'] = 'ABCD123'
print('New keys added to bank1:\n',bank1)
print('*'*30)
#
print(bank) # original list WILL be modified when we copying using  =
```

Copy using ** / shallow copy

```
original list WILL NOT be modified when we copying using   **
******************************

bank = {'name': 'SBI', 'Location': 'Tanjore'}
print('originalBankdict \n', bank)
print('*'*30)

bank1= {**bank}
print('original dict is copied to bank1\n', bank1)
```

```
print('*'*30)

bank1['IFSC'] = 'ABCD123'
print('New keys added to bank1:\n',bank1)
print('*'*30)
#
print(bank) # original list WILL NOT be modified when we copying using   **
```

output

originalBankdict

{'name': 'SBI', 'Location': 'Tanjore'}

****

original dict is copied to bank1

{'name': 'SBI', 'Location': 'Tanjore'}

****

New keys added to bank1:

{'name': 'SBI', 'Location': 'Tanjore', 'IFSC': 'ABCD123'}

****

{'name': 'SBI', 'Location': 'Tanjore'}

How copy works in Dict

When copy the dict, the dict only copied, not the objects of the dict – see below

```
bank = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank)
print("bank Dict ID ", id(bank))
print("==================")
print("bank dict's KEY  'SBI' object id", id(bank["SBI"]))

bank1 = {**bank}
print("After shallow copy using **\n", bank1)
print("bank1 Dict id ", id(bank1))
print("==================")

print("Bank1's  KEY -  SBI ID ", id(bank1["SBI"]))
```

output

```
{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
bank Dict ID  2207643939072
==================
bank dict's KEY  'SBI' object id 2207643939184
After shallow copy using **
 {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
```

bank1 Dict id 2207643939520
==========
Bank1's KEY - SBI ID 2207643939184

==========

Another pgm

```
bank = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank)
print("bank Dict ID ", id(bank))
print("bank SBI object id", id(bank["SBI"]))
print("==========")

bank1 = {**bank}
print("bank1 Dict id ", bank1)
print("bank1 ", id(bank1))
print("Bank1 SBI ID ", id(bank1["SBI"]))
print("==========")

for item in bank:
    print(id(item))
print("==========")

for item in bank1:
    print(id(item))
```

output

{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}

bank Dict ID 2082992238848

bank SBI object id 2082992238960

==========

bank1 Dict id {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}

bank1 2082992239296

Bank1 SBI ID 2082992238960

==========

2082992285872

2082992282800

2082992311152

==========

2082992285872

2082992282800

2082992311152

==========

Copy with ** -Same keys in 2 dicts

How copy () works if both dict has SAME keys. In this case the second dict over writes the first dict

```
bank1 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank1)
print(id(bank1))
print("===========")

bank2 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank2)
print(id(bank2))
print("===========")

shallowCopy = { **bank1 , **bank2  } # shallowCopy = { **bank2 , **bank1  } tyr this too and see if the ids are same or different
```

```
print(shallowCopy)
print(id(shallowCopy))
```

output

```
{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
2889023369472
==========
{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
2889023369920
==========
{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
2889024126208
----------
```

Copy with **- Different keys in 2 dicts

How copy () works if both dict has DIFFERENT keys. In this case, second dict has been added to the first dict (The last key's value will be considered)

```
bank1 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank1)
print(id(bank1))
print("===========")

bank2 = {'SBI1': 'Location', 'IFSC1': 'ABC1234', 'Number1': 100}
print(bank2)
print(id(bank2))

print("===========")
shallowCopy = { **bank1 , **bank2  }
print(shallowCopy)
print(id(shallowCopy))
```

output

```
{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
1563004580096
==========
{'SBI1': 'Location', 'IFSC1': 'ABC1234', 'Number1': 100}
1563004580544
==========
```

{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100, 'SBI1': 'Location', 'IFSC1': 'ABC1234', 'Number1': 100}

1563005336832

==========

```
bank1 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
# print(bank1)
bank2 = {'SBI': 'Locationnn', 'IFSCCC': 'ABC1234', 'Number': 1000}

shallowCopy = { **bank1}
print(shallowCopy)

s = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100, 'SBI': 'Locationnn', 'IFSCCC': 'ABC1234', 'Number': 1000}
print(s)

shallowCopy = { **bank1 , **bank2  }
print(shallowCopy)
```

output

{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
{'SBI': 'Locationnn', 'IFSC': 'ABC1234', 'Number': 1000, 'IFSCCC': 'ABC1234'}
{'SBI': 'Locationnn', 'IFSC': 'ABC1234', 'Number': 1000, 'IFSCCC': 'ABC1234'}

How to add 2 dictionaries?

```
dic1 = {'Name':'Aafi', 'Age': 20}
dic2 = {'Marks' : 90, "Name" : "Thiruvembavai"}
dic3=dic1.copy() # first we copy the dic1 into dic3
dic3.update(dic2) # then update dic3 with dic2

print (dic3)
```

output

{'Name': 'Aafi', 'Age': 20}

{'Name': 'Aafi', 'Age': 20, 'Marks': 90}

==========

Update() vs {**orignaldict}

- Update(), takes only one dict to update()
- ** takes as many as dicts to add / append / copy and returns one big dict

```
dic1 = {'Name':'Aafi', 'Age': 20}
dic2 = {'Marks' : 90}
dic3 = {"Grade": "First"}

dic1.update(dic2)
print(dic1)

d4 = {**dic1, **dic2, **dic3} # ** takes as many dicts as we wish
print(d4)
----------------
```

Update function

copy function removes the existing value and add the new values

```
d1 = {1 : 'one', 2 : 'two'}
d2 = {3 : 'three'}

d1.update(d2)
```

```
print (d1)
print (d2)
```

Ans

{1: 'one', 2: 'two', 3: 'three'}

{3: 'three'}

```
d = {1: "one", 2: "three"}
d1 = {2: "two"}

# updates the value of key 2
d.update(d1)
print(d)

d1 = {3: "three"}

# adds element with key 3
d.update(d1)
print(d)
```

ans

{1: 'one', 2: 'two'}

{1: 'one', 2: 'two', 3: 'three'}

Copy() vs update()

copy function removes the existing value and add the new values

update fn **does not remove** the existing value after the update

---

What happens if we remove the non-existing key from the dict

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
lst1=[1,2,3]
dic1["Sl_Number"] = lst1
print (dic1)
print (dic1["KEY NOT THERE"]) # try to access the key  "KEY NOT THERE", which is not in dict
output error
```

```
    print (dic1["KEY NOT THERE"])
KeyError: 'KEY NOT THERE'
{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': [1, 2, 3]}
```

Del in dict

To delete an item, delete the key from the dictionary:

```
dic1=  {'name': 'Anna University', 'Location': 'Chennai'}
del(dic1['name'])
print(dic1)
del(dic1)
```

output

```
{'Location': 'Chennai'}
```

```
dic1=  {'name': 'Anna University', 'Location': 'Chennai'}
del(dic1)
print(dic1)
```

delete the entire dict

Using Sorted() in dict

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print(sorted(dic1))
```

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print(sorted(dic1,reverse=True))
```

Check ids in =, copy and in dict(dict) // dict's constructor

```
e = {'three': 3, 'one': 1, 'two': 2}
print(e)
print(id(e))

print("---------------")
k = e e = {'three': 3, 'one': 1, 'two': 2}
print(e)
```

```
print(id(e))
print(id(e["three"]))

print("---------------")
k = e
print(k)
print(id(k))

print("---------------")
s = e.copy()
print(s)
print(id(s))
#
print("---------------")
f = dict(e)
print(f)
print(id(f))
print(id(f["three"]))

print(k)
print(id(k))

print("---------------")
s = e.copy()
print(s)
print(id(s))

print("---------------")
f = dict(e)
```

```
print(f)
print(id(f))
```

output

```
{'three': 3, 'one': 1, 'two': 2}
16068112
--------------
{'three': 3, 'one': 1, 'two': 2}
16068112
--------------
{'three': 3, 'one': 1, 'two': 2}
16068512
--------------
{'three': 3, 'one': 1, 'two': 2}
30202248
====================
```

if we update a single item, the original will not be affected / updated, where as if we update a sub dict, the oritinal WILL BE IMPACTED – see below

```
d1 = {"sub1": 78, "sub2": 87, "sub3" : 98, "sub4" : {"s4" : 44, "s5" : 55}, "sub5" : "1"}
print(id(d1["sub3"]))
sc = {**d1}
print(sc)
sc["sub4"]["s5"] = 5500
print(sc)
```

```
print("original dict ", d1)
```

output

```
2204821294352
{'sub1': 78, 'sub2': 87, 'sub3': 98, 'sub4': {'s4': 44, 's5': 55}, 'sub5': '1'}
{'sub1': 78, 'sub2': 87, 'sub3': 98, 'sub4': {'s4': 44, 's5': 5500}, 'sub5': '1'}
original dict  {'sub1': 78, 'sub2': 87, 'sub3': 98, 'sub4': {'s4': 44, 's5': 5500}, 'sub5': '1'}
```

Iterating Over a Dictionary

If you use a dictionary as an iterator (e.g. in a for statement), it traverses the keys of the dictionary. For example:

```
d = {'a' : 1, 'b' : 2, 'c' : 3}
for key in d.keys() :
    print(key)

""" OR WE CAN USE THE BELOW CODE"""

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

OUTPUT

a

b

c

a 1

b 2

c 3

```
for item, value in enumerate(d.items()):
    print(item, value)
```

output

0 ('a', 10)

1 ('b', 20)

2 ('cccc', 3

Accessing keys and values

When working with dictionaries, it's often necessary to access all the keys and values in the dictionary, either in a for loop, a list comprehension, or just as a plain list.

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
for item in dic1.keys():
   print(item)
----------------
for item in dic1.values():
   print(item)
--------------------

for item in dic1.items():
   print(item)
--------------------
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print(list(dic1))
========================
```

Another way of accessing keys(), values(), items()

Melcose note: Watch the type() output

```
dict = {}
dict ["red"] = "255:0:0"
dict ["green"] = "0:255:0"
dict ["blue"] = "0:0:255"

print (dict) # the answer in dictionary format
print (dict["red"]) # using the key, take the value of the dictionary
print (dict.keys()) #give only key values, We can use the d.items() method to retrieve a list of tuples containing  keyvalue pairs
print (dict.values())
print (dict.items()) # displays as Tuple format inside of list format,
print ("red" in dict) # check if the key is there inside the dictionary
print ("The number of items in the dictionary is :", len((dict)))
```

output

```
{'red': '255:0:0', 'green': '0:255:0', 'blue': '0:0:255'}
255:0:0
dict_keys(['red', 'green', 'blue'])
dict_values(['255:0:0', '0:255:0', '0:0:255'])
dict_items([('red', '255:0:0'), ('green', '0:255:0'), ('blue', '0:0:255')])
```

True
The number of items in the dictionary is : 3

** Dictionary Unpacking

Normal function declaration with parameters

```
def method1(internal, external):
    totalMarks = internal + external
    # print (totalMarks)
    return totalMarks


result = method1(24, 74)
print ("Total marks is ", result)
```

output

Total marks is 98

Function declaration with default values for parameters

```
def method1(external, internal=24):
    totalMarks = internal + external
```

```
    # print (totalMarks)
    return totalMarks

result = method1(62)
print ("Total marks is ", result)
```

output

Total marks is 86

Unpacks the contents of a dictionary into the function call.

Syntax

function(**dict)

```
def method1(internal=0, external=1):
    totalMarks = internal + external
    print (totalMarks)
    return totalMarks
```

```
d = {'internal' : 24, 'external' : 68} # when we call the method we pass keyargs notation (**)

method1(**d)
```

output

92

Another code (sending 2 dicts as arguments and then unpack)

```
def marks (inter,extern):
    total = inter+extern
    return total

d={"inter":24}
d1={"extern":22}

print (marks(**d,**d1))
```

output

46

What would happen if we send duplicate keys during dict unpacking

```
def method1(internal, external):
    totalMarks = internal + external
    # print (totalMarks)
    return totalMarks

d = {'internal' :25, 'external' : 70, "internal" : 20} # when we call the method we pass keyargs notation (**)

print(method1(**d))
```

output: since dict takes the last duplicate vaule, it works well

output

9m

Below works well

```
def method1(internal, external):
    totalMarks = internal + external
    print (totalMarks)
    return totalMarks
```

```
d = {'internal' : 24, 'external' : 68} # when we call the method we pass keyargs notation (**)

method1(**d)
--------------------
```

```
def method1(external=0, internal=1): # parameter order can be interchanged, still it works
    totalMarks = internal + external
    print (totalMarks)
    return totalMarks

d = {'internal' : 24, 'external' : 68} # when we call the method we pass keyargs notation (**)

method1(**d)
---------------------------
```

What happens if we have default argumentn and dict unpacking

```
def method1(external, internal=50): # parameter order can be interchanged, still it works
    totalMarks = internal + external
    print (totalMarks)
```

```
    return totalMarks

d = { 'external' : 68} # when we call the method we pass keyargs notation (**)

method1(**d)
```

output
it works
==========

Avoiding KeyError Exceptions

One common pitfall when using dictionaries is to access a non-existent key. This typically results in a `KeyError` exception

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

One way to avoid key errors is to use the `dict.get` method, which allows you to specify a default value to return in the case of an absent key.

```
value = mydict.get(key, default_value)
```

Which returns `mydict[key]` if it exists, but otherwise returns `default_value`. Note that this doesn't add `key` to `mydict`. So if you want to retain that key value pair, you should use `mydict.setdefault(key, default_value)`, which *does* store the key value pair.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

An alternative way to deal with the problem is catching the exception

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

You could also check if the key is `in` the dictionary.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Do note, however, that in multi-threaded environments it is possible for the key to be removed from the dictionary after you check, creating a race condition where the exception can still be thrown.

Another option is to use a subclass of dict, collections.defaultdict, that has a default_factory to create new entries in the dict when given a new_key.

One common pitfall when using dictionaries is to access a non-existent key. This typically results in a KeyError exception

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
lst1=[1,2,3]
dic1["Sl_Number"] = lst1
print (dic1)
print("=============")

print(dic1["Subject11"]) #KeyError: 'Subject11'
```

output

KeyError: 'Subject11'

```
print(dic1.get("Subject11")) # None
print("=============")
```

```
print(dic1.get("Pen", "Sorry customer, the item that you order is not available")) # None
print("============")

# print(dic1["Subject11"]) #KeyError: 'Subject11'
```

output

```
{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': [1, 2, 3]}

============

None

============

Sorry customer, the item that you order is not available

============

------------------------
```

Get () gives the value of the key- if exists, else no error is thrown

Supply correct key in get() // gives the value from the dict

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print (dic1)
print (dic1.get("Name"))
```

output

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

AAA

To avoid the exception / error use get()

Supply **incorrect** key in get() // gives None

If we try to use the get() to get the value of the key and if the key is not there, it returns 'None' / no error is shown

And key WILL NOT add to original dict

```
# Supply incorrect key in get() // gives None
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print (dic1)
print (dic1.get("Name1"))
print(dic1)
```

Output

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

None

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

Supply incorrect key AND value in get() // gives value from get(). But key WILL NOT be added to dict

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print (dic1)
print (dic1.get("Name1", "Anto"))
print(dic1)
```

output

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

Anto

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

Setdefault()

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print(dic1.get("Pen", "Not available") )#But key WILL NOT be added to dict
print(dic1)
print("==============")

dic1.setdefault("Pen", "Not Available")# key WILL be added to dict
```

```
print(dic1)
```

output

Not available

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

==========

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2, 'Pen': 'Not Available'}

==========

To avoid this use default key concept

One way to avoid key errors is to use the dict.get method, which allows you to specify a default value to return in the case of an absent key.

```
mydict = { }
print (mydict)
print (mydict.get("foo", "bar")) # foo is key, bar is value
print (mydict)
mydict.setdefault("Name", "AAA")
print(mydict)
```

output

{}

bar

{}

{'Name': 'AAA'}

Note: """we are setting the defalut values to the empty dictionary, # if the dict already has an item, that value also will be shown"""

Diff between get() and setdefault()

Get() doesn't add key to dict, but setdefualt() does

Get() gives the value from the dict, if we supply the correct key. Else it will give the default value of the get(), ie None

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
print(dic1)
dic1.setdefault('Zey', "Value")
print(dic1)
```

output

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2}

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2, 'Zey': 'Value'}

```
mydict={1:10, 2:20}
value = mydict.setdefault(3,30)
print(value)
print(mydict)
mydict[5] = 50
print(mydict)
```

output

30

{1: 10, 2: 20, 3: 30}

{1: 10, 2: 20, 3: 30, 5: 50}

==========

```
dic1 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
dic2 = {10 : "AAA", 66 : "Python", 2.5 : 2}

dic1.setdefault('Zey', "Value")
print(dic1)
print(sorted(dic1))
print(sorted(dic2))
```

output

{'Name': 'AAA', 'Subject': 'Python', 'Sl_Number': 2, 'Zey': 'Value'}

['Name', 'Sl_Number', 'Subject', 'Zey']

[2.5, 10, 66]

Sorted() and reversed()

Note: reversed() will work with dict from 3.8 version and gives "dict_reversekeyiterator object"
Teach the below during generator / iterator / iterable

```
dic1 = {10 : "AAA", 5 : 7,  2:"Linda"}
print(sorted(dic1))
print(sorted(dic1, reverse=True))
print("============")

dic1 = {"B" : "BB", "A" : "AA",  "C":"CC"}
print(sorted(dic1))
print(sorted(dic1, reverse=True))
print("============")

dic2 = {"Name" : "AAA", "Subject" : "Python","Sl_Number" : 2}
revObject = (reversed(dic2))
```

```
print(revObject)
print(next(revObject))
print(next(revObject))
print(next(revObject))
print(next(revObject, "Sorry, no more keys"))
```

```
[2, 5, 10]
[10, 5, 2]
============
['A', 'B', 'C']
['C', 'B', 'A']
============
<dict_reversekeyiterator object at 0x0000023D2AFC51C0>
Sl_Number
Subject
Name
Sorry, no more keys
----------
```

Use Setdefault ()to add a list of values

Notes from Mahendran (DSIT)

if you must use the built-in dict class, using dict.setdefault() will allow you to create a default whenever you access a key that did not exist before:

Coding:

```
d = {}
d.setdefault('Another_key', []).append("This worked!")
print(d)
```

Output:

{'Another_key': ['This worked!']}

Keep in mind that if you have many values to add, dict.setdefault() will create a new instance of the initial value (in this example a []) every time it's called - which may create unnecessary workloads.

Only the empty list within the setdefault() method can be appended, if already value is available in the list.

Empty list can be appended with as many times is as possible.

Coding:

```
d ={"sub": 1, "Name": "AAA"}
d.setdefault("subhead", []).append("Django,TeslorFlow")
d.setdefault("subhead", []).append("Djgo,TeorFlow")
d.setdefault("subhead", []).append("Sam,Van")
print(d)
```

Output:

{'sub': 1, 'Name': 'AAA', 'subhead': ['Django,TeslorFlow', 'Djgo,TeorFlow', 'Sam,Van']}

Coding:

We can give values with in the list[values] only one time. Second time empty[] only appends the values given in append(method).

```
d ={"sub": 1, "Name": "AAA"}
d.setdefault("subhead", ["ML", "NL", "ToraFlow",]).append("Panda,Numpy")
d.setdefault("subhead", []).append("Django,TeslorFlow")
d.setdefault("subhead", []).append("Djgo,TeorFlow")
d.setdefault("subhead", []).append("Samu,Vani")
print(d)
```

Output:

{'sub': 1, 'Name': 'AAA', 'subhead': ['ML', 'NL', 'ToraFlow', 'Panda,Numpy', 'Django,TeslorFlow', 'Djgo,TeorFlow', 'Samu,Vani']}

Second time values within the list will not update, example:

```
d ={"sub": 1, "Name": "AAA"}
d.setdefault("subhead", ["ML", "NL", "ToraFlow",]).append("Panda,Numpy")
d.setdefault("subhead", ["Try", "DS",]).append("Django,TeslorFlow")
d.setdefault("subhead", []).append("Djgo,TeorFlow")
print(d)
```

Output:

{'sub': 1, 'Name': 'AAA', 'subhead': ['ML', 'NL', 'ToraFlow', 'Panda,Numpy', 'Django,TeslorFlow', 'Djgo,TeorFlow']}

Any number of times, we can append() values with empty list[] in the value section of setdefault() method.

Additional code

```
d = {   }
d.setdefault("Subject") # note we don't give default value, so it takes None as its value
print (d)
```

output

{'Subject': None}

```
d = {}
d.setdefault("Subject", "Python")
print (d)

d.setdefault("Data Scinece", [] ).append("Panda, Numpy") #{'Subject': 'Python', 'Data Scinece': ['Panda, Numpy']}
print (d)

d.setdefault("Data Scinece", ["ML"]).append("Panda, Numpy")
print (d) # {'Subject': 'Python', 'Data Scinece': ['ML', 'Panda, Numpy']}
```

```
# d.setdefault("Data Scinece", []).append("Panda, Numpy") #{'Subject': 'Python', 'Data Scinece': None}
# print (d)
```

output

```
{'Subject': 'Python'}

{'Subject': 'Python', 'Data Scinece': ['Panda, Numpy
```

Notes: Keep in mind that if you have many values to add, dict.setdefault() will create a new instance of the initial value (in this example a []) every time it's called - which may create unnecessary workloads.

How to create a key inside a tuple and list of values to the tuple, using setdefault()

```
d = {}
d.setdefault("Subject", "Python")
print (d)

d.setdefault( ("Sudha",), [] ).append("Panda, Numpy") #{{'Subject': 'Python', ('Sudha',): ['Panda,
```

```
Numpy']}
print (d)
```

Another code to create tuple as key, set {} of values

```
d = {}
d.setdefault("Subject", "Python")
print (d)

d.setdefault( ("Sudha",), set(("Seaborn",)) ).update(("Pandas, Numpy",)) #{{'Subject': 'Python',
('Sudha',): ['Panda, Numpy']}
print (d)
```

output

{'Subject': 'Python'}

{'Subject': 'Python', ('Sudha',): {'Seaborn', 'Pandas, Numpy'}}

How to convert a single value to list of values - KPV

How to convert an existing value into list

```
dic1 = {"name" : "AAA", "Subject" : "Python","Sl_Number" : 2}

dic1["Subject"] = [dic1["Subject"]]
print(dic1)
dic1["Subject"].append("Numpy")
print(dic1)
```

output

```
{'name': 'AAA', 'Subject': ['Python'], 'Sl_Number': 2}
{'name': 'AAA', 'Subject': ['Python', 'Numpy'], 'Sl_Number': 2}
```

---

check if a key is in the dictionary

```
mydict = {"key" : "AAA", "Subject" : "Python","Sl_Number" : 2}
if "key1" in mydict:
    value = mydict["key"]
    print(value)
```

```
else:
    value = mydict.setdefault("default_value", "VVVV")
    # value = mydict.setdefault("default_value")# the value will be None

print(mydict)
```

Do note, however, that in multi-threaded environments it is possible for the key to be removed from the dictionary after you check, creating a race condition where the exception can still be thrown.

Another option is to use a subclass of dict, collections.defaultdict, that has a default_factory to create new entries in the dict when given a new_key

---

How zip() works in iterables and in dict

```
phdList = ["Sudha", "Aswathy", "Linda", "Murugan"]
markList = [1110, 30,40]
locationList = ["USA", "Canada", "Australia"]
meritList = zip(phdList, markList,locationList, strict = False)
print(meritList)
print(list(meritList))
print(type(meritList))
```

output

```
<zip object at 0x000001FB4C4699C0>
[('Sudha', 1110, 'USA'), ('Aswathy', 30, 'Canada'), ('Linda', 40, 'Australia')]
<class 'zip'>
======================
```

```
name1 = "Linda"
name2="Sudha"
c = zip(name1,name2, strict=True)
print(list(c))
```

output

[('L', 'S'), ('i', 'u'), ('n', 'd'), ('d', 'h'), ('a', 'a')]

==========

Declarttion of dict using key:value, keyword arguments, list of tuples and using zip

class dict(***kwarg*) // Linda

class dict(*mapping*, ***kwarg*)

class dict(*iterable*, ***kwarg*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object.

Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects.

The **first** object of each item becomes a **key** in the new dictionary, and the **second** object the corresponding **value**. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

```
a = dict(one=1, two=2, three=3)
print(a)
```

```
b = {'one': 1, 'two': 2, 'three': 3}
print(b)

# --------------
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
print(c)

d = dict([('two', 2), ('one', 1), ('three', 3)]) # list of tuples
print(d)

e = dict({'three': 3, 'one': 1, 'two': 2})
print(e)

print(a==b==c==d==e)
```

output

```
{'one': 1, 'two': 2, 'three': 3}
{'one': 1, 'two': 2, 'three': 3}
{'one': 1, 'two': 2, 'three': 3}
{'two': 2, 'one': 1, 'three': 3}
{'three': 3, 'one': 1, 'two': 2}
True
```

What happenes if we use SET of values, inside the zip()

Since set gives us unordered list, this is not the right way

```
n1=set("hai")
print(n1)
n2= set("python")
print(n2)
c = zip(n1,n2,strict=False)
print(c)
print(list(c))
```

output

{'a', 'h', 'i'}

{'o', 'n', 'h', 'p', 't', 'y'}

<zip object at 0x00000212BFC74E00>

[('a', 'o'), ('h', 'n'), ('i', 'h')]

==========

Pop()

```
bank1 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank1.pop('IFSC'))
print(bank1)
```

output

ABC1234

{'SBI': 'Location', 'Number': 100}

==========

Popitem()

def popitem(self) -> tuple[_KT, _VT]

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

```
bank1 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank1.popitem())
print(bank1)
```

output

('Number', 100)

{'SBI': 'Location', 'IFSC': 'ABC1234'}

==========

clear()

```
bank1 = {'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}
print(bank1)
bank1.clear()
print(bank1)
```

output

{'SBI': 'Location', 'IFSC': 'ABC1234', 'Number': 100}

{}

Collections - Counter in Dict

Counter is a container that will hold the **count** of each of the elements present in the container / iterable / collection. **Counter** is a sub-class available inside the dictionary class. Using the **Python Counter** tool, you can **count** the key-value pairs in an object, also called a hashtable object.

counter: A **variable** used in a loop to **count** the number of times something happened. We initialize a **counter** to zero and then **increment** the **counter** each time we want to "**count**" something. decrement: An update that decreases the value of a **variable**

```
from collections import Counter

# initialising dictionaries
ini_dictionary1 = Counter({'nikhil': 1, 'AAA': 100,'manjeet': 10, 'BBB': 200})
print ("number of items are:" , len(ini_dictionary1))
ini_dictionary2 = Counter({'AAA': 100, 'BBB': 200,'m': 15})
print ("number of items are:" , len(ini_dictionary2))

# printing initial dictionaries
print("initial 1st dictionary", str(ini_dictionary1))
print("initial 2nd dictionary", str(ini_dictionary2))

# combining dictionaries
# using Counter
```

```
final_dictionary = ini_dictionary1 + ini_dictionary2

# printing final result
print("final dictionary", str(final_dictionary))
print ("number of items are:" , len(final_dictionary))
```

Ans

number of items are: 4

number of items are: 3

initial 1st dictionary Counter({'BBB': 200, 'AAA': 100, 'manjeet': 10, 'nikhil': 1})

initial 2nd dictionary Counter({'BBB': 200, 'AAA': 100, 'm': 15})

final dictionary Counter({'BBB': 400, 'AAA': 200, 'm': 15, 'manjeet': 10, 'nikhil': 1})

number of items are: 5

1. Notes1: When adding 2 dictionaries , if the **same name of keys** are available in both dictionaries, it **adds** the key's values from both dictionaries
2. Note2: If the first dict does not have the key, but the second set has the key it takes the key from the second dict (in plus operation)

3. Note3: It wil not print if the values are **negative** and **zero**. It is applicable for minus and + operations
4. Note 5: If the first dict has a key and if its value is negative, and it will not be printed in (in minus andn + operation)
5. If both dict has the same key, union will take the maximum value, where as intersection takes the min value (Pl check with code if this is correct or not)

```
from collections import Counter
counter1 =  Counter({'x': 5, 'y': 12, 'z': -2, 'x1':0})
counter2 = Counter({'x': 2, 'y':5})

counter3 = counter1 - counter2
print(counter3)
```

output:

```
ounter({'y': 7, 'x': 3})
```

```
from collections import Counter
counter2 =  Counter({'x': 5, 'y': 12, 'z': -2, 'x1':0})
```

```
counter1 = Counter({'x': 2, 'y':5})

counter3 = counter1 - counter2
print(counter3)
```

Beware: we are mixing counter1 with counter2, Not Counter2 with Counter1

Output:

```
Counter({'z': 2})
```

```
counter1 = Counter({'x': 2, 'y':5, 'z':0 })
counter2 =  Counter({'x': 5, 'y': 12,  'x1':0, 'z':-2 })

counter3 = counter1 + counter2
print(counter3)
```

output

```
Counter({'y': 17, 'x': 7})
```

What happens if dict has duplicate values (it takes the last key's value)

```
from collections import Counter
counter1 = Counter({'x': 2, 'y':5, 'z':0, 'x' :100})
counter2 =  Counter({'x': 5, 'y': 12,  'x1':0, 'z':-2 })

counter3 = counter1 + counter2
print(counter3)
```

output

Counter({'x': 105, 'y': 17)}

========

Use of counter in a sequence

Counter is used to returns the number of times the identical values appeared in the list and shows us in dict format

```
from collections import Counter

lst = [10,20,30,10,20,50, 10]
print(Counter(lst))
```

output

Counter({10: 3, 20: 2, 30: 1, 50: 1})

Collections.Counter accepts only “+” and “–“ operators , others raise

TypeError: unsupported operand type(s) for ** or pow(): 'Counter' and 'Counter'

==========

Dict can be initialised in 5 ways as below

1. key : value
2. keyword arguemnts
3. list tuples
4. dict(zip(iter1, iter2)
5. Counter()