Python Dictionary Methods

Python has a set of built-in methods that you can invoke on dictionary objects.

Method	Description
clear()	**Removes all items from the dictionary**
copy()	**Returns a shallow copy of the dictionary**
fromkeys()	**Creates a new dictionary with the specified keys and values**
get()	**Returns the value of the specified key**
items()	**Returns a list of key:value pair**
keys()	**Returns a list of all keys from dictionary**
pop()	**Removes and returns single dictionary item with specified key.**
popitem()	**Removes and returns last inserted key:value pair from the dictionary.**
setdefault()	**Returns the value of the specified key, if present. Else, inserts the key with a specified value.**
update()	**Updates the dictionary with the specified key:value pairs**
values()	**Returns a list of all values from dictionary**

Python Dictionary clear() Method

Removes all items from the dictionary

Usage

Use `clear()` **method to remove all items from the dictionary. This method does not return anything; it modifies the dictionary in place.**

Syntax

```
dictionary.clear()
```

Example

```
D = {'name': 'Bob', 'age': 25}

D.clear()

print(D)
# Prints {}
```

clear() vs Assigning Empty Dictionary

Assigning an empty dictionary `D = {}` is not same as `D.clear()`. For example,

```
old_Dict = {'name': 'Bob', 'age': 25}

new_Dict = old_Dict

old_Dict = {}

print(old_Dict)

# Prints {}

print(new_Dict)
# Prints {'age': 25, 'name': 'xx'}
```

`old_Dict = {}` **does not empty the dictionary in-place, it just overwrites the variable with a different dictionary which happens to be empty. If anyone else like `new_Dict` had a reference to the original dictionary, that remains as-is.**

On the contrary, `clear()` method empties the dictionary in-place. So, all the references are cleared as well.

```
old_Dict = {'name': 'Bob', 'age': 25}

new_Dict = old_Dict

old_Dict.clear()

print(old_Dict)

# Prints {}

print(new_Dict)
# Prints {}
```

Python Dictionary copy() Method

Copies the dictionary shallowly

Usage

The `copy()` method returns the Shallow copy of the specified dictionary.

Syntax

```
dictionary.copy()
```

Example

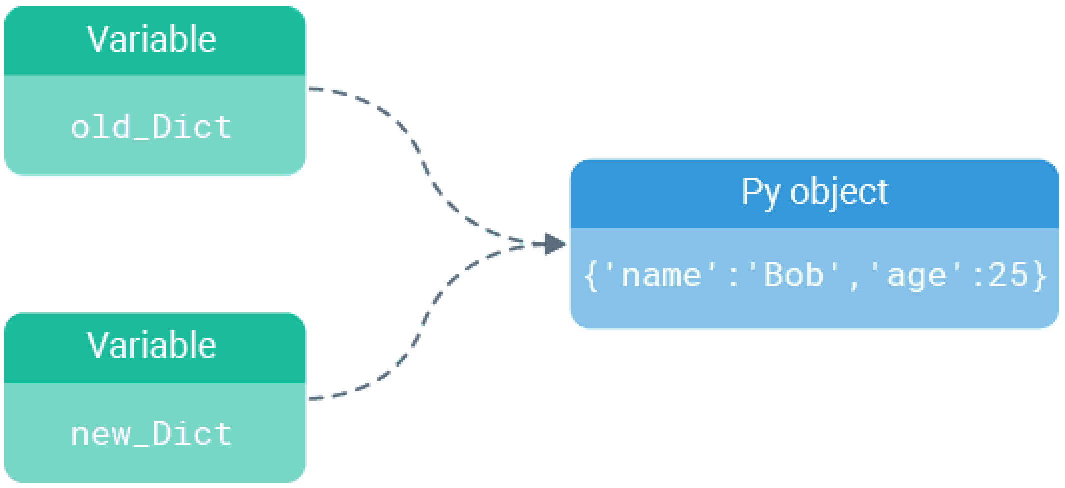
```
D = {'name': 'Bob', 'age': 25}

X = D.copy()

print(X)
# Prints {'age': 25, 'name': 'Bob'}
```

copy() vs Assignment statement

Assignment statement does not copy objects. For example,

```
old_Dict = {'name': 'Bob', 'age': 25}

new_Dict = old_Dict

new_Dict['name'] = 'xx'

print(old_Dict)

# Prints {'age': 25, 'name': 'xx'}

print(new_Dict)
# Prints {'age': 25, 'name': 'xx'}
```

When you execute new_Dict = old_Dict**, you don't actually have two dictionaries. The assignment just makes the two variables point to the one dictionary in memory.**



So, when you change new_Dict, old_Dict is also modified. If you want to change one copy without changing the other, use copy()**method.**

```
old_Dict = {'name': 'Bob', 'age': 25}

new_Dict = old_Dict.copy()

new_Dict['name'] = 'xx'

print(old_Dict)

# Prints {'age': 25, 'name': 'Bob'}

print(new_Dict)
# Prints {'age': 25, 'name': 'xx'}
```

Equivalent Method

You can copy dictionary using dictionary comprehension as well.

```
D = {'name': 'Bob', 'age': 25}

X = {k:v for k,v in D.items()}

print(X)
# Prints {'age': 25, 'name': 'Bob'}
```

Python Dictionary fromkeys() Method

Creates a new dictionary with default value

Usage

The fromkeys() **method creates a new dictionary with default** value **for all specified** keys**.**

If default value **is not specified, all keys are set to None.**

Syntax

```
dict.fromkeys(keys,value)
```

Parameter	Condition	Description
keys	**Required**	**An iterable of keys for the new dictionary**
value	**Optional**	**The value for all keys. Default value is None.**

Examples

```
# Create a dictionary and set default value 'Developer' for all keys
D = dict.fromkeys(['Bob', 'Sam'], 'Developer')
print(D)
# Prints {'Bob': 'Developer', 'Sam': 'Developer'}
```

If default value **argument is not specified, all keys are set to None.**

```
D = dict.fromkeys(['Bob', 'Sam'])
print(D)
# Prints {'Bob': None, 'Sam': None}
```

Equivalent Method

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the `fromkeys()` **method.**

```
# As if default value is specified
L = ['Bob', 'Sam']
D = {key:'Developer' for key in L}
print(D)
# Prints {'Bob': 'Developer', 'Sam': 'Developer'}

# As if default value is not specified
L = ['Bob', 'Sam']
D = {key:None for key in L}
print(D)
# Prints {'Bob': None, 'Sam': None}
```

Python Dictionary get() Method

Returns the value for key if exists

Usage

The get() **method returns the value for** key **if key is in the** **dictionary.**

You can also specify the default **parameter that will be returned if the specified key is not found. If** default **is not specified, it returns None. Therefore, this method never raises a** KeyError**.**

It's an easy way of getting the value of a key from a dictionary without raising an error.

Syntax

dictionary.get(key,default)

Parameter	Condition	Description
key	**Required**	**Any key you want to search for**
default	**Optional**	**A value to return if the specified key is not found. Default value is None.**

Basic Examples

get() **method is generally used to get the value for the specific** key**.**

```
D = {'name': 'Bob', 'age': 25}

print(D.get('name'))
# Prints Bob
```

If key is not in the dictionary, the method returns None.

```
D = {'name': 'Bob', 'age': 25}

print(D.get('job'))
# Prints None
```

Sometimes you want a value other than None to be returned, in which case specify the default parameter.

The default Parameter

If key is in the dictionary, the method returns the value for key (no matter what you pass in as default).

```
D = {'name': 'Bob', 'age': 25, 'job': 'Manager'}

print(D.get('job', 'Developer'))
# Prints Manager
```

But if key is not in the dictionary, the method returns specified default.

```
D = {'name': 'Bob', 'age': 25}

print(D.get('job','Developer'))
# Prints Developer
```

get() Method vs Dictionary Indexing

The `get()` method is similar to indexing a dictionary by key in that it returns the value for the specified key. However, it never raises a `KeyError`, if you refer to a key that is not in the dictionary.

```
# key present

D = {'name': 'Bob', 'age': 25}

print(D['name'])
```

```
# Prints Bob
print(D.get('name'))
# Prints Bob

# key absent
D = {'name': 'Bob', 'age': 25}
print(D['job'])
# Triggers KeyError: 'job'
print(D.get('job'))
# Prints None
```

Python Dictionary items() Method

Returns a list of key-value pairs in a dictionary

Usage

The `items()` method returns a list of tuples containing the key:value pairs of the dictionary. The first item in each tuple is the key, and the second item is its associated value.

Syntax

```
dictionary.items()
```

Examples

```
# Print all items from the dictionary
```

```
D = {'name': 'Bob', 'age': 25}
L = D.items()
print(L)
# Prints dict_items([('age', 25), ('name', 'Bob')])
```

`items()` method is generally used to iterate through both keys and values of a dictionary. The return value is the tuples of `(key, value)`.

```
# Iterate through both keys and values of a dictionary
D = {'name': 'Bob', 'age': 25}
for x in D.items():
    print(x)
# Prints ('age', 25)
# Prints ('name', 'Bob')
```

items() Returns View Object

The object returned by `items()` is a view object. It provides a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

```
D = {'name': 'Bob', 'age': 25}

# Assign dict items to L
L = D.items()

# modify dict D
D['name'] = 'xx'

# L reflects changes done to dict D
print(L)
# Prints dict_items([('age', 25), ('name', 'xx')])
```

Python Dictionary keys() Method

Returns a list of keys from a dictionary

Usage

The keys() **method returns a list of keys from a dictionary.**

Syntax

```
dictionary.keys()
```

Examples

```
# Print all keys from the dictionary
D = {'name': 'Bob', 'age': 25}
L = D.keys()
print(L)
# Prints dict_keys(['age', 'name'])
```

keys() **method is generally used to iterate through all the keys from a dictionary.**

```
# Iterate through all the keys from a dictionary
D = {'name': 'Bob', 'age': 25}
for x in D.keys():
    print(x)
# Prints age name
```

keys() Returns View Object

The object returned by `items()` is a view object. It provides a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

```
D = {'name': 'Bob', 'age': 25}

# Assign dict keys to L
L = D.keys()

# modify dict D
D['job'] = 'Developer'

# L reflects changes done to dict D
print(L)
# Prints dict_keys(['job', 'age', 'name'])
```

Python Dictionary pop() Method

Removes a key from the dictionary

Usage

If specified `key` is in the dictionary, the `pop()` method removes it and returns its value. You can also specify the `default` parameter that will be returned if the specified key is not found.

If `default` is not specified and `key` is not in the dictionary, a KeyError is raised.

Syntax

```
dictionary.pop(key,default)
```

Parameter	Condition	Description
key	**Required**	**Any key you want to remove**
default	**Optional**	**A value to return if the specified key is not found.**

Examples

pop() **method is generally used to remove a key from the dictionary.**

```
D = {'name': 'Bob', 'age': 25}
D.pop('age')
print(D)
# Prints {'name': 'Bob'}
```

This method not only removes the specified key, but also returns its value.

```
D = {'name': 'Bob', 'age': 25}
v = D.pop('age')
print(v)
# Prints 25
```

If key **is not in the dictionary, the method raises KeyError exception.**

```
D = {'name': 'Bob', 'age': 25}
D.pop('job')
# Triggers KeyError: 'job'
```

To avoid such an exception, you need to specify the default **parameter.**

The default Parameter

If `key` is in the dictionary, the `pop()` method removes it and returns its value (no matter what you pass in as `default`).

```
D = {'name': 'Bob', 'age': 25}

v = D.pop('age', 0)

print(D)

# Prints {'name': 'Bob'}

print(v)
# Prints 25
```

But if `key` is not in the dictionary, the method returns specified `default`.

```
D = {'name': 'Bob', 'age': 25}

v = D.pop('job', 'Developer')

print(v)
# Prints Developer
```

Python Dictionary popitem() Method

Removes a key-value pair from a dictionary

Usage

The `popitem()` method removes and returns the last inserted `key:value` pair from the dictionary. Pairs are returned in Last In First Out (LIFO) order.

In versions before 3.7, `popitem()` would remove and return a random item.

Syntax

dictionary.popitem()

Examples

```
# Remove the last inserted item from the dictionary
D = {'name': 'Bob', 'age': 25}
D.popitem()
print(D)
# Prints {'name': 'Bob'}
```

`popitem()` returns key:value pair of removed item as a tuple.

```
D = {'name': 'Bob', 'age': 25}
v = D.popitem()
print(v)
# Prints ('age', 25)
```

popitem() on Empty Dictionary

calling `popitem()` on an empty dictionary, raises a `KeyError` exception.

```
D = {}
D.popitem()
# Triggers KeyError: 'popitem(): dictionary is empty'
```

To avoid such exception, you must check if the dictionary is empty before calling the `popitem()` method.

```
D = {}
if D:
    D.popitem()
```

Python Dictionary setdefault() Method

Returns the value for key if exists, else inserts it

Usage

The `setdefault()` **method returns the value for** `key` **if key is in the dictionary. If not, it inserts** `key` **with a value of** `default` **and returns default.**

Syntax

```
dictionary.setdefault(key,default)
```

Parameter	Condition	Description
key	**Required**	**Any key you want to return value for**
default	**Optional**	**A value to insert if the specified key is not found. Default value is None.**

Basic Example

`setdefault()` **method is generally used to insert a key with a default value.**

```
# Insert a key 'job' with default value 'Dev'
D = {'name': 'Bob', 'age': 25}
v = D.setdefault('job', 'Dev')
print(D)
# Prints {'job': 'Dev', 'age': 25, 'name': 'Bob'}
print(v)
# Prints Dev
```

setdefault() Method Scenarios

The method's output depends on input parameters. Here are three scenarios for different input parameters.

Key Present

If `key` **is in the dictionary, the method returns the value for key (no matter what you pass in as** `default`**)**

```
# without default specified
D = {'name': 'Bob', 'age': 25}
v = D.setdefault('name')
print(v)
# Prints Bob

# with default specified
D = {'name': 'Bob', 'age': 25}
v = D.setdefault('name', 'Max')
print(v)
# Prints Bob
```

Key Absent, Default Specified

If key is not in the dictionary, the method inserts key with a value of default and returns default.

```
D = {'name': 'Bob', 'age': 25}

v = D.setdefault('job', 'Dev')

print(D)

# Prints {'job': 'Dev', 'age': 25, 'name': 'Bob'}

print(v)
# Prints Dev
```

Key Absent, Default Not Specified

If key is not in the dictionary and default is not specified, the method inserts key with a value None and returns None.

```
D = {'name': 'Bob', 'age': 25}

v = D.setdefault('job')

print(D)

# Prints {'job': None, 'age': 25, 'name': 'Bob'}

print(v)
# Prints None
```

Python Dictionary update() Method

Updates/Adds multiple items to the dictionary

Usage

The `update()` method updates the dictionary with the key:value pairs from element.

- **If the key is already present in the dictionary, value gets updated.**
- **If the key is not present in the dictionary, a new key:value pair is added to the dictionary.**

element **can be either another dictionary object or an iterable of key:value pairs (like list of tuples).**

Syntax

dictionary.update(element)

Parameter	Condition	Description
element	**Optional**	**A dictionary or an iterable of key:value pairs**

Examples

update() **method is generally used to merge two dictionaries.**

```
D1 = {'name': 'Bob'}

D2 = {'job': 'Dev', 'age': 25}

D1.update(D2)

print(D1)
# Prints {'job': 'Dev', 'age': 25, 'name': 'Bob'}
```

When two dictionaries are merged together, existing keys are updated and new key:value pairs are added.

```
D1 = {'name': 'Bob', 'age': 25}

D2 = {'job': 'Dev', 'age': 30}

D1.update(D2)

print(D1)
# Prints {'job': 'Dev', 'age': 30, 'name': 'Bob'}
```

Note that the value for existing key ‘age’ is updated and new entry ‘job’ is added.

Passing Different Arguments

`update()` **method accepts either another dictionary object or an iterable of** `key:value` **pairs (like tuples or other iterables of length two).**

```
# Passing a dictionary object

D = {'name': 'Bob'}

D.update({'job': 'Dev', 'age': 25})

print(D)
# Prints {'job': 'Dev', 'age': 25, 'name': 'Bob'}

# Passing a list of tuples

D = {'name': 'Bob'}

D.update([('job', 'Dev'), ('age', 25)])

print(D)
# Prints {'age': 25, 'job': 'Dev', 'name': 'Bob'}

# Passing an iterable of length two (nested list)

D = {'name': 'Bob'}

D.update([['job', 'Dev'], ['age', 25]])

print(D)
# Prints {'age': 25, 'job': 'Dev', 'name': 'Bob'}
```

`key:value` **pairs can be also be specified as keyword arguments.**

```
# Specifying key:value pairs as keyword arguments

D = {'name': 'Bob'}

D.update(job = 'Dev', age = 25)

print(D)
```

```
# Prints {'job': 'Dev', 'age': 25, 'name': 'Bob'}
```

Python Dictionary values() Method

Returns a list of values from a dictionary

Usage

The `values()` method returns a list of values from a dictionary.

Syntax

```
dictionary.values()
```

Examples

```
# Print all values from the dictionary
D = {'name': 'Bob', 'age': 25}
L = D.values()
print(L)
# Prints dict_values([25, 'Bob'])
```

`values()` method is generally used to iterate through all the values from a dictionary.

```
# Iterate through all the values from a dictionary
D = {'name': 'Bob', 'age': 25}
for x in D.values():
    print(x)
```

```
# Prints 25 Bob
```

values() Returns View Object

The object returned by `items()` is a view object. It provides a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

```
D = {'name': 'Bob', 'age': 25}


# Assign dict values to L
L = D.values()


# modify dict D
D['name'] = 'xx'


# L reflects changes done to dict D
print(L)
# Prints dict_values([25, 'xx'])
```

Built-in Functions with Dictionary

Python also has a set of built-in functions that you can use with dictionary objects.

Method	Description
all()	**Returns True if all list items are true**
any()	**Returns True if any list item is true**
len()	**Returns the number of items in the list**
sorted()	**Returns a sorted list**

Python all() Function

Determines whether all items in an iterable are True

Usage

The all() function returns True if all items in an iterable are True. Otherwise, it returns False.

If the iterable is empty, the function returns True.

Syntax

```
all(iterable)
```

Parameter	Condition	Description
iterable	**Required**	**An iterable of type (list, string, tuple, set, dictionary etc.)**

Falsy Values

In Python, all the following values are considered False.

- **Constants defined to be false:** None **and** False**.**
- **Zero of any numeric type:** 0**,** 0.0**,** 0j**,** Decimal(0)**,** Fraction(0, 1)
- **Empty sequences and collections:** ''**,** ()**,** []**,** {}**,** set()**,** range(0)

Basic Examples

```
# Check if all items in a list are True
```

```
L = [1, 1, 1]
print(all(L))   # Prints True


L = [0, 1, 1]
print(all(L))   # Prints False
```

Here are some scenarios where `all()` returns False.

```
L = [True, 0, 1]
print(all(L))   # Prints False


T = ('', 'red', 'green')
print(all(T))   # Prints False


S = {0j, 3+4j}
print(all(S))   # Prints False
```

all() on a Dictionary

When you use `all()` function on a dictionary, it checks if all the keys are true, not the values.

```
D1 = {0: 'Zero', 1: 'One', 2: 'Two'}
print(all(D1))   # Prints False


D2 = {'Zero': 0, 'One': 1, 'Two': 2}
print(all(D2))   # Prints True
```

all() on Empty Iterable

If the iterable is empty, the function returns True.

```
# empty iterable
```

```
L = []
print(all(L))   # Prints True


# iterable with empty items
L = [[], []]
print(all(L))   # Prints False
```

Python any() Function

Determines whether any item in an iterable is True

Usage

The `any()` function returns True if any item in an iterable is True. Otherwise, it returns False.

If the iterable is empty, the function returns False.

Syntax

```
any(iterable)
```

Parameter	Condition	Description

iterable	**Required**	**An iterable of type (list, string, tuple, set, dictionary etc.)**

Falsy Values

In Python, all the following values are considered False.

- **Constants defined to be false:** `None` **and** `False`**.**
- **Zero of any numeric type:** `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- **Empty sequences and collections:** `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Basic Examples

```
# Check if any item in a list is True

L = [0, 0, 0]
print(any(L))   # Prints False

L = [0, 1, 0]
print(any(L))   # Prints True
```

Here are some scenarios where `any()` **returns True.**

```
L = [False, 0, 1]
print(any(L))   # Prints True

T = ('', [], 'green')
print(any(T))   # Prints True

S = {0j, 3+4j, 0.0}
print(any(S))   # Prints True
```

any() on a Dictionary

When you use `any()` function on a dictionary, it checks if any of the keys is true, not the values.

```
D1 = {0: 'Zero', 0: 'Nil'}
print(any(D1))   # Prints False


D2 = {'Zero': 0, 'Nil': 0}
print(any(D2))   # Prints True
```

any() on Empty Iterable

If the iterable is empty, the function returns False.

```
L = []
print(any(L))   # Prints False
```

Python len() Function

Returns the number of items of an object

Usage

The `len()` function returns the number of items of an object.

The object may be a sequence (such as a string, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

Syntax

```
len(object)
```

Parameter	Condition	Description
object	**Required**	**A sequence or a collection.**

len() on Sequences

```
# number of characters in a string
S = 'Python'
x = len(S)
print(x)
# Prints 6
# number of items in a list
L = ['red', 'green', 'blue']
x = len(L)
print(x)
# Prints 3
# number of items in a tuple
T = ('red', 'green', 'blue')
x = len(T)
print(x)
# Prints 3
```

len() on Collections

```
# number of key:value pairs in a dictionary
D = {'name': 'Bob', 'age': 25}
x = len(D)
print(x)
```

```
# Prints 2

# number of items in a set

S = {'red', 'green', 'blue'}

x = len(S)

print(x)
# Prints 3
```

Python sorted() Function

Sorts the items of an iterable

Usage

The `sorted()` **method sorts the items of any** iterable

You can optionally specify parameters for sort customization like sorting order and sorting criteria.

Syntax

```
sorted(iterable,key,reverse)
```

The method has two optional arguments, which must be specified as keyword arguments.

Parameter	Condition	Description

iterable	**Required**	**Any iterable (list, tuple, dictionary, set etc.) to sort.**
key	**Optional**	**A function to specify the sorting criteria. Default value is None.**
reverse	**Optional**	**Settting it to True sorts the list in reverse order. Default value is False.**

Return Value

The method returns a new sorted list from the items in iterable**.**

Sort Iterables

`sorted()` **function accepts any** iterable **like list, tuple, dictionary, set, string etc.**

```
# strings are sorted alphabetically
L = ['red', 'green', 'blue', 'orange']
x = sorted(L)
print(x)
# Prints ['blue', 'green', 'orange', 'red']

# numbers are sorted numerically
L = [42, 99, 1, 12]
x = sorted(L)
print(x)
```

```
# Prints [1, 12, 42, 99]
```

If you want to sort the list in-place, use built-in sort() method.

sort() **is actually faster than** sorted() **as it doesn't need to create a new list.**

```
# Sort a tuple

L = ('cc', 'aa', 'dd', 'bb')

x = sorted(L)

print(x)
# Prints ['aa', 'bb', 'cc', 'dd']
```

sorted() **function sorts a dictionary by keys, by default.**

```
D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}

x = sorted(D)

print(x)
# Prints ['Bob', 'Max', 'Sam', 'Tom']
```

To sort a dictionary by values use the sorted() **function along with the values() method.**

```
D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}

x = sorted(D.values())

print(x)
# Prints [20, 25, 30, 35]
```

Sort in Reverse Order

You can also sort an iterable **in reverse order by setting** reverse **to true.**

```
L = ['cc', 'aa', 'dd', 'bb']

x = sorted(L, reverse=True)

print(x)
# Prints ['dd', 'cc', 'bb', 'aa']
```

Sort with Key

Use key parameter for more complex custom sorting. A key parameter specifies a function to be executed on each list item before making comparisons.

For example, with a list of strings, specifying key=len (the built-in len() function) sorts the strings by length, from shortest to longest.

```
L = ['orange', 'red', 'green', 'blue']

x = sorted(L, key=len)

print(x)
# Prints ['red', 'blue', 'green', 'orange']
```