

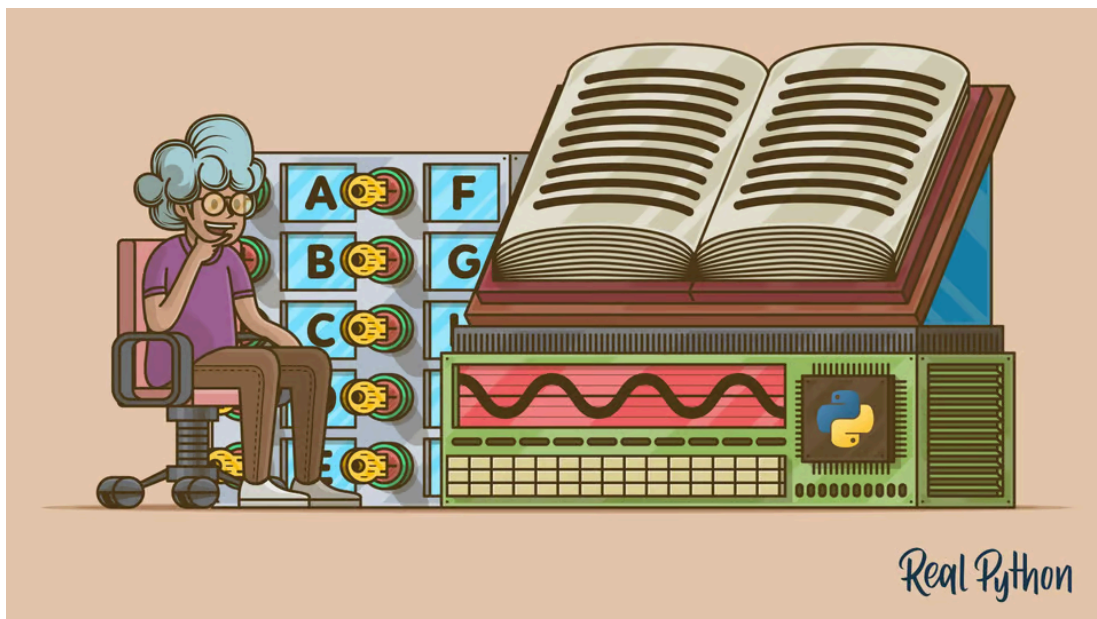
# 5. Dictionary

Let's dive into the world of dictionaries, a fundamental data structure in Python. We'll cover how to create, access, and manipulate key-value pairs within dictionaries.

A **dictionary** in Python is an unordered collection of key-value pairs. It's a data structure that allows you to store and retrieve data using unique keys. Each key is associated with a corresponding value.

We'll learn about the following topics:

- 5.1. Creating Dictionary
- 5.2. Built-in Dictionary Functions
- 5.3. Accessing Dictionary Values
- 5.4. Dictionary Properties
- 5.5. Dictionary Operators
- 5.6. Built-in Dictionary Methods
- 5.7. Nesting Dictionaries



Name	Type in Python	Description	Example
Dictionaries	dict	Unordered key:value pairs in { }.	{'key':10, 'word': 'hello'}

Dictionaries are a type of mapping. Mappings are collections of objects stored with unique keys, unlike sequences that store objects based on their relative positions. This distinction is significant because mappings do not preserve the order of objects, as they are defined by their keys.

```
In [1]: type({'key':10, 'word': 'hello'})
```

```
Out[1]: dict
```

## 5.1. Creating Dictionary:

Dictionaries in Python are created using curly braces `{}`. Each key-value pair within the dictionary is separated by a colon `:`.

```
In [2]: b = {'key1':'value1', 'key2':'value2', 'key3':'value3'}
```

It's important to note that dictionaries are very flexible in the data types they can hold.

```
In [3]: a = {'key1':123, 'key2':[12,23,33], 'key3':['item0','item1','item2']}
```

Please note that keys in a Python dictionary can be any immutable object, including numbers, strings, tuples, and booleans. Lists cannot be used as keys because they are mutable.

```
In [4]: my_dict = {1: "apple", (1, 2): "point A", True: "enabled"}
```

## 5.2. Built-in Dictionary Functions:

- **len()** : Python's built-in `len()` function returns the number of key-value pairs in the dictionary.

```
In [5]: len(a)
```

```
Out[5]: 3
```

## 5.3. Accessing Dictionary Values:

To access the value associated with a specific key in a dictionary, you use square brackets `[]` and provide the key as an argument.

`Name_of_Dictionary[key] ---> value`

```
In [6]: a
```

```
Out[6]: {'key1': 123, 'key2': [12, 23, 33], 'key3': ['item0', 'item1', 'item2']}
```

```
In [7]: a['key3']
```

```
Out[7]: ['item0', 'item1', 'item2']
```

```
In [8]: #Can call an index on that value  
a['key2'][0]
```

```
Out[8]: 12
```

```
In [9]: a['key2'][-1]
```

```
Out[9]: 33
```

## 5.4. Dictionary Properties:

- **Unordered**

```
In [10]: a = {'key1':123, 'key2':[12,23,33], 'key3':['item0','item1','item2']}  
  
b = {'key2':[12,23,33], 'key3':['item0','item1','item2'], 'key1':123}
```

```
In [11]: a == b
```

```
Out[11]: True
```

This clearly shows that dictionaries are not ordered.

- **Mutability:** Dictionaries are mutable meaning that when a dictionary is created, the elements within it can be changed or replaced.

```
In [12]: a['key1'] = 125
```

```
In [13]: a
```

```
Out[13]: {'key1': 125, 'key2': [12, 23, 33], 'key3': ['item0', 'item1', 'item2']}
```

```
In [14]: #Can then even call methods on that value  
a['key3'][0].upper()
```

```
Out[14]: 'ITEM0'
```

We can affect the values of a key as well. For instance:

```
In [15]: #Subtract from the value  
a['key1'] = a['key1'] - 12
```

```
In [16]: a
```

```
Out[16]: {'key1': 113, 'key2': [12, 23, 33], 'key3': ['item0', 'item1', 'item2']}
```

Just a quick note: In Python, there's a convenient built-in method for performing self-subtraction, addition, multiplication, or division. Alternatively, we could use the += -= \*= /= etc. operators. For example:

```
In [17]: a['key1'] /= 2
```

```
In [18]: a
```

```
Out[18]: {'key1': 56.5, 'key2': [12, 23, 33], 'key3': ['item0', 'item1', 'item2']}
```

## 5.5. Dictionary Operators:

- **in** : Python also provides a membership operator that can be used with dictionaries. The in operator returns True if a key exists in the dictionary, and False otherwise.

```
In [19]: 'key1' in a
```

```
Out[19]: True
```

- **not in** : Python also provides a membership operator that can be used with dictionaries. The not in operator returns True if a key doesn't exist in the dictionary, and False otherwise.

```
In [20]: 'key1' not in a
```

```
Out[20]: False
```

## 5.6. Built-in Dictionary Methods:

Method	Description
keys()	return a list of all keys
values()	return a list of all values
items()	return tuples of all items
update(2nd dic)	Merge two dictionaries
get(key)	returns the value for a given key
pop(key)	removes the specified key-value pair from the dictionary and return the value of the removed key
popitem()	removes and returns an arbitrary key-value pair from the dictionary as a tuple
clear()	empties dictionary of all key-value pairs

```
In [21]: a.keys()
```

```
Out[21]: dict_keys(['key1', 'key2', 'key3'])
```

```
In [22]: a.values()
```

```
Out[22]: dict_values([56.5, [12, 23, 33], ['item0', 'item1', 'item2']])
```

```
In [23]: a.items()
```

```
Out[23]: dict_items([('key1', 56.5), ('key2', [12, 23, 33]), ('key3', ['item0', 'item1', 'item2'])])
```

```
In [24]: b = {'one':1, 'two':2}
c = {'three': 3, 'four': 4}

b.update(c)
b
```

```
Out[24]: {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Instead of using `.update()` method, you can add a new key-value pair in the following way:

```
In [25]: a['programming'] = 'Python'
```

```
In [26]: print(a)
```

```
{'key1': 56.5, 'key2': [12, 23, 33], 'key3': ['item0', 'item1', 'item2'], 'programming': 'Python'}
```

We can't directly concatenate dictionaries using the `+` operator.

```
In [27]: a.get('key1')
```

```
Out[27]: 56.5
```

`.get()` is equivalent to `dictionary_name[key_name]`.

```
In [28]: a['key1']
```

```
Out[28]: 56.5
```

```
In [29]: a.pop('programming')
```

```
Out[29]: 'Python'
```

```
In [30]: a
```

```
Out[30]: {'key1': 56.5, 'key2': [12, 23, 33], 'key3': ['item0', 'item1', 'item2']}
```

```
In [31]: key_pop, value_pop = a.popitem()

print('Removed Key:', key_pop)
print('Removed Value:', value_pop)
print('Updated Dictionary:', a)
```

```
Removed Key: key3  
Removed Value: ['item0', 'item1', 'item2']  
Updated Dictionary: {'key1': 56.5, 'key2': [12, 23, 33]}
```

```
In [32]: a.clear()
```

```
In [33]: a
```

```
Out[33]: {}
```

## 5.7. Nesting Dictionaries:

```
In [34]: dic1 = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

```
In [35]: #Keep calling the keys  
dic1['key1']['nestkey']['subnestkey']
```

```
Out[35]: 'value'
```

Dictionaries and lists share the following characteristics:

- Both are mutable.
- Both are dynamic: They can grow and shrink as needed.
- Both can be nested: A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing.
- Dictionary elements are accessed via keys.