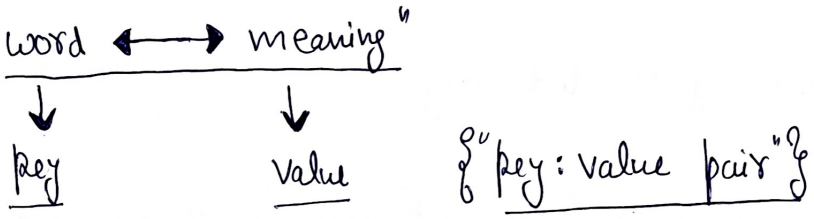
"Python Dictionary" :→

Dictionaries are used to store data values in key : value pairs.

It is similar to dictionary in real life, where we have "word-meaning pairs", we have a word, and all the meanings related to this particular word are present next to it.



In dictionary (real life) word acts as a key. It means whenever we need to find the meaning of a particular word, we will search for the word in the dictionary. Generally we donot search for the meaning, we start the search through "word".

So, word we say is the key of our pair, "key" means important thing. & meaning is basically the value of this key.

So, in order to store "key : value" information in python, we use dictionary datatype. It is a builtin datatype like lists and tuples.

Note: Dictionaries are used to store data values in "key : value" pairs. They are unordered, mutable (changeable) & donot allow "duplicate keys".

Now let's see how to create a dictionary in python.

P.T.O

```
dict = {
        "name"  : "vipul",
        "cgpa"  : 9.61,
        "marks" : [98, 97, 95],
}
```

It's name of the dictionary

There are key value pairs.

For eg:⇒ Let's create a dictionary to store some information

```
info = {
        "key" : "value",
        "name" : "SMVDU",
        "learning" : "coding"
}
```

Now let's print our dictionary info.

```
print(info)
```

O/P: {'key' : 'value', 'name' : 'SMVDU', 'learning' : 'coding'}

In O/P key value pairs are separated with a comma (,)

In key:value pair, value could be any datatype, It can be string, It can be a number, It can be a boolean value, It can be a floating value, list and tuples could also be stored.

[P.T.O]

Eg:→ info = {

"name" : "SMVDU",
"subjects" : ["Python", "C", "Java"], → list
"topics" : ("dict", "set"), → tuple
"age" : 25,
"is_adult" : True, → boolean
'marks' : 94.5 → float value
}

Note :→ Key can be a string, a number, a boolean value, a floating value, a tuple.

Note :→ Tuple is not mutable, & since tuple cannot be changed, hence key can be tuple as well.

To print the type of our dictionary, we the following command.

```
print(type(info))
```

O/p : <class 'dict'>

Note :→ Dictionaries are unordered, (there is no fix rule, which key: value pair appear first & which should appear last).

Note :→ In string, list & tuples, we have indices, so we know which value comes first & which value comes later. But in dictionary we do not have any such index. Hence there is no order.

Note :→ Tuple is immutable, It cannot be changed. List is mutable, it can be changed.

[P.T.O]

In dictionary we cannot have duplicate keys.

Now, let's see how to access the values of a dictionary using keys.

```
dict = {
        "name" : "Vipul",
        "cgpa" : 9.6,
        "marks" : [98, 97, 95],
       }
```

Using key name, we can access its value as shown:

```
dict["name"], dict["cgpa"], dict["marks"]

print (dict["name"])
```

O/p Vipul.

We can even change value of a key present in the dictionary as shown below:

```
dict["name"] = "Pankaj"
```

We can even add a new key:value pair in an already existing dictionary as shown:

```
dict["surname"] = "Sharma"
```

Note :- Dictionary in python are mutable, we can change the values & even add new values.

Note :⇒ If we try to create a new key with the same name as already existing, then python will overwrite the previous key.

Note :⇒ We can even create an empty dictionary. It is as shown below.

```
null_dict = { }
```

↓ name of dictionary, it could be anything.

Later, we can add key value pairs in this null dictionary.

→ Nested Dictionaries :⇒ In nested dictionary, we can create the value of a key as a dictionary itself.

So, dictionary within a dictionary is known as "nested dictionary".

Eg :⇒

```
student = {
        "name" : "vipul sharma",
        "subjects" : {
                "Physics" : 97,
                "Chemistry" : 98,
                "Maths" : 92,
                }
        }
print(student)
```

→ This is a nested dictionary, which is a dictionary within a dictionary.

o/p: {'name' : 'vipul sharma', 'subjects' : {'Physics' : 97, 'Chemistry' : 98, 'Maths' : 92 } }.

If let's say we want to print only subjects, it can be done as shown below:

```
print(student["subjects"])
```

O/p: {'Physics': 97, 'Chemistry': 98, 'Maths': 92}

Now, let's say we want to print marks of chemistry only. It could be done like, as shown:

```
print(student["subjects"]["Chemistry"])
```

O/p: 98

In nested dictionaries, relevant information can be retrieved using series of square brackets.

Now let's understand the dictionary methods :→

① .keys() # returns all keys.

It is used to fetch all the keys. It returns a collection of all the keys in our dictionary

Eg:>

```
student = {
        "name": "Vipul Sharma",
        "subjects": {
            "Physics": 97,
            "Chemistry": 98,
            "Maths": 95
        }
}
```

[P.T.O]

```
print (student.keys())
```

O/p: dict_keys(['name', 'subjects'])

↙ In the form of a list format.

Note :→ Nested keys are not returned.
It returns only outer layer keys.

This datatype dict_keys can be converted into a normal tuple or list using typecasting.

It can be done as :-

```
print ( list (student.keys()))
```

O/p: ['name', 'subjects'].

→ In order to get total no. of keys, we can do it like :-

```
print (len(student))
```

O/p : 2

or

```
print ( len(list (student.keys())))
```

O/p : 2

Note :→ We can call one function inside 2nd function, 2nd function inside 3rd function & so-on.

[P.T.O]

② .values() # returns all values in our dictionary

It returns a collection of all the values in the dictionary.

Eg:→ print (student.values())

O/p: dict_values (['vipul sharma', {'Physics': 97, 'Chemistry': 98, 'Maths': 95}])

It can also be converted into a list using typecasting.

print(list (student.values()))

O/p: ['vipul sharma', {'Physics': 97, 'Chemistry': 98, 'Maths': 95}]

Note :→ We can store one datatype inside another datatype, we can store list inside a dictionary & vice versa.

③ .items() # returns all (key, value) pairs as tuples.

It returns all 'key-value' pairs in a dictionary in the form of "tuples"

Eg:→ print(student.items())

O/p: dict_items([('name', 'vipul sharma'), ('subjects', {'Physics': 97, 'Chemistry': 98, 'Maths': 95})])

Let's typecast it into a list :

```
print (list (student.items()))
```

O/p: [('name', 'vipul Sharma'), ('subjects', {'Physics': 97, 'Chemistry': 98, 'Maths': 95})]

We can even access these individual tuples as shown below :

```
pairs = list(student.items())
print(pairs[0])
```

→ accessing first tuple, as index is "0"

O/p: ('name', 'vipul Sharma')

(4) .get(key) # returns the key according to value

By passing the key as an argument in this method, the value of that particular key will be returned.

There are two ways to return the value of a particular key from the dictionary.

1) d["key"] ⟶ Value

It returns the value of that particular key.

2) d.get("key") ⟶ Value

It also returns the value of that particular key.

Now let's understand, why we have two ways to fetch the value of a particular key.

Let's say we want to print the value of key "name",

```
print(student["name"])          # first-method.
print(student.get("name"))      # second-method.
```

Difference b/w these two is, if let's say we use a key which doesn't exist, then first method will give us an error & second method will return "None"

Eg:→ `print(student["name1"])` → It will give error.
`print(student.get("name1"))` → It will return none.

The problem with first method is that, if it gives an error, then the lines of code written after this, even if they are correct cease to run, whereas no such problem will occur with the second method.

---

(5) `.update( )` # inserts the specified items to the dictionary.
↳ we can all together pass a "key : value pair" or even a new dictionary.

If we pass a new key value pair as an argument, it will get added in already created dictionary.

P.T.O

Let's try to add age of the student in already existing 'student dictionary' or city name

It could be done as shown:

```
student.update({"city": "Jammu"})
```

Let's print it.

```
print(student)
```

O/p: {'name': 'Vipul Sharma', 'Subjects': {'Physics': 97, 'Chemistry': 98, 'Maths': 95}, 'city': 'Jammu'}

We can also do it like

```
new_dict = {"city": "Jammu"}
```
→ we created new dictionary

```
student.update(new_dict)
```
→ using new dictionary, we updated the previous one.

multiple key-value pairs can also be added using the 'update function'.

Eg:-
```
new_dict = {"city": "Jammu", "age": 25}
student.update(new_dict)
```

Note:- if let's say, we try to enter a key-value pair, such that the key already existed in the dictionary, then in that case update function will overwrite the old value of that particular key. New key with the same name will not be created.

[P.T.O]

Note:→ Duplicate keys are not allowed in a dictionary.

→ "Methods to remove a key from the dictionary":

① Using pop() :→ pop() method removes a specific key from the dictionary & returns its corresponding value.

Eg:→
```
a = {"name": "vipul", "age": 25, "city": "Jammu"}
rv = a.pop("age")
print(a)
print(rv)
```

O/p:
```
{'name': 'vipul', 'city': 'Jammu'}
25
```

② Using del() :→ del() statement deletes a key-value pair from the dictionary directly.

Eg:→
```
a = {"name": 'vipul', "age": 25, "city": "Jammu"}
del a["city"]
print(a)
```

O/p: {'name': 'vipul', 'age': 25}

Note:→ del() statement directly removes the key-value pair for "city" & does not return the value making it ideal when the value is not needed.

—

③ Using dictionary comprehension :⇒

This method allows us to create a new dictionary without the key, we want to remove.

Eg :⇒

```
a = {"name": "vipul", "age": 25, "city": "Jammu"}
a = {k:v for k, v in a.items() if k != "name"}
print(a)
```

O/p: {'age': 25, 'city': 'Jammu'}

---

④ Using popitem() for Last Key Removal :⇒

if we want to remove the last key-value pair in the dictionary, we can use popitem() method.

Eg :⇒

```
a = {"name": "vipul", "age": 25, "city": "Jammu"}
c = a.popitem()
print(a)
print(c)
```

O/p :- {'name': 'vipul', 'age': 25}
('city', 'Jammu')

---

⑤ Using pop() with Default :- We can also provide a default value to the pop() method, ensuring no error occurs if the key doesn't exist.

[P.T.O]

Eg:→ a = {"name": "vipul", "age": 25, "city": "Jammu"}

rv = a.pop("country", "Key not found")

↓ key name

↓ Value we want to print, in case key is not present.

```
print(a)
print(rv)
```

O/p: {'name': 'vipul', 'age': 25, 'city': 'Jammu'}
Key not found.

Note :→ if the key exists, pop() removes it & returns its value & if the key does not exist, the default value ("key not found") is returned, preventing errors.

"Set in Python":

Set is the collection of the unordered items, Each element in the set must be unique and immutable.

In mathematics, set is a collection of unique values.

Similarly in python set is a collection of unordered items. (no index is present in sets.)

Note → boolean, int, float, string, tuples can be stored in a set, becoz they are immutable, whereas list & dictionary cannot be stored.

Now let's see how to create a set.

Eg:⇒ nums = {1, 2, 3, 4}

Collection = {1, 2, 3, 4}

print(collection)
print(type(collection))

O/p: {1, 2, 3, 4}
<class 'set'>

Note:⇒ String values can also be added in a set.

If let's say we create a set with duplicate values, then in that case python set will automatically remove the duplicate values.

Eg:⇒ Collection = {1, 2, 2, 2, "hello", "world", "world"}

print(collection)
print(type(collection))

O/p: {'world', 1, 2, 'hello'} ← duplicate values are automatically removed.
<class 'set'>

Note:⇒ "set is always unordered."

Note:⇒ if we use length function len() on sets, it will also not consider duplicate values.

[P.T.O]

Let's now see, how to create an "empty set":

```
Collection = set()
print(type(collection))
```

O/p: <class 'set'>

"Some commonly used set methods":→

① set.add(el) # adds an element

It is used for adding an element in a set.

"set name" "element to be inserted."

Note :→ Sets are "mutable", it means we can add or remove the elements from a set;
but set elements are "immutable".
Values of set elements cannot be changed.

Note :→ In add() function, we can pass a tuple, but we cannot pass a "list" or a "dictionary".

Eg:→ Collection = set() → It is an empty set.

Now let's add some values in this "empty set",

```
Collection.add(1)
Collection.add(2)
Collection.add(3)
Collection.add(2)
```

→ only one value will be inserted.

(P.T.O)

print (collection)

O/p:- {1, 2, 3}

(2) set.remove(el) # remove the passed element.

set.remove (el)

↓ "name of set" ↓ "element to be removed from set"

Eg:- Collection.remove(1)
print (collection)

O/p:- {2, 3}

Note:- If let's say we want to remove a value which doesn't exist in set, then in that case we will get an error.

Note :- We can add a tuple, a string to a set, but adding a list or dictionary is not allowed, as they are mutable.

Note :- "Hashable values" mean "immutable values".

(3) set.clear() # empties the set.

It empties the set.

Eg:- Collection = set()
Collection.add(1)
Collection.add(2)
Collection.add(3)
Collection.add(4)

P.T.O.

```
print(len(collection))
```

O/p: 4.

Now let's try to clear the collection set.

```
collection.clear()
print(len(collection))
```

O/p: 0.

(4) set.pop() # removes a random value.

It will remove a random value from the set.

(5) set.remove() # removes a specified value from the set

Eg:- fruits = {'apple', 'banana', 'mango'}

Note :- set can also be created as shown above.

```
fruits.remove('banana')
print(fruits)
```

O/p: {'mango', 'apple'}.

---

(6) set1.union(set2) # Combines both set values & returns new. Duplicate values are counted only once.

Eg:- set1 = {1, 2, 3}
set2 = {2, 3, 4}

```
print(set1.union(set2))
```

O/p: {1, 2, 3, 4}

[set 1 & set 2 still retain their original values]

⑦ set.intersection(set2) # Combines common values & returns new.

Eg:-
```
set1 = {1, 2, 3}
set2 = {2, 3, 4}
print(set1.intersection(set2))
print(set1)
print(set2)
```

O/P:-
```
{2, 3}
{1, 2, 3}
{2, 3, 4}
```

⑧ set.copy() # This method copies the set.

Eg:-
```
fruits = {'apple', 'banana', 'cherry'}
x = fruits.copy()
print(x).
```

⑨ set.discard(value) # This method removes the specified item from the set.

Note:- This method is different from the remove() method, beoz. the remove() method will raise an error, if the specified item does not exist, and the discard() method will not.

Eg:-
```
fruits = {'apple', 'banana', 'cherry'}
fruits.discard('banana')
print(fruits).
```

(10) set.update(set2) :⇒

The update() method updates the current set, by adding items from another set (or any other iterable).

Note :⇒ If an item is present in both the sets, only one appearance of this item will be present in the updated set.

Note :⇒ as a shortcut we can also use |= operator.

Eg:⇒
```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.update(y)
print(x)
```
O/p: {'microsoft', 'apple', 'cherry', 'google', 'banana'}

```
x |= y   (same output will be generated)
```

We can even insert more than one set.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = {"cherry", "micra", "bluebird"}
x.update(y, z)
print(x)

x |= y | z
print(x)
```

set1.difference(set2) :⇒

Returns a set that contains the items that only exist in set1 & not in set2.:

Eg:-

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.difference(y)
print(z)
```

o/p: {'cherry', "banana"}

It can also be written as :⇒

```
z = x - y
print(z).
```

Joining more than two sets :⇒

```
a = {"apple", "banana", "cherry"}
b = {"google", "microsoft", "apple"}
c = {"cherry", "micra", "bluebird"}
myset = a.difference(b,c).        or myset = a - b - c
print(myset)
```

o/p = {"banan"}

It means elements which are present in "a" but not in "b" or "c"

(12) isdisjoint() :→ This method returns True if none of the items are present in both sets, otherwise it returns false

Or

Returns True, if no items in set x is present in set y:

Eg:→ x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "facebook"}

z = x.isdisjoint(y)

print(z)

O/p. True.

(13) issubset() :→ Returns True if all the items in the set exists in the specified set, otherwise it returns False.

Eg:→ x = {"a", "b", "c"}

y = {"f", "e", "d", "c", "b", "a"}

z = x.issubset(y)

print(z).

O/p. True

(14) issuperset() :→ Returns True if all the items in the specified set exists in the original set, otherwise returns False

Eg:→ x = {"f", "e", "d", "c", "b", "a"}

y = {"a", "b", "c"}

P.T.O

```
z = x.issuperset(y)
print(z)
```

O/p True

Note ⇒ Returns True if all the items in set y are present in set x.

—— o ——