In Python Everything is Object i.e. the instance of any class, whether it’s a number, string, list, tuple, function.

“ Python is an object-oriented programming language which means that it supports user-defined classes and objects. What you might not know is that every pre-existing thing available in Python is already an object, whether it is strings, numbers, functions, or even classes. In this video, you will build a solid knowledge of Python objects by learning to check the type of objects, listing their attributes & methods, and understanding what actually goes under the hood. “

type() method

It use to return the type of the objects

type.py

```
number1 = 5
string1 = "Hello! Test type()"
list1 = [1,2,3]
dic1 = {'a': 2, 'b':5}
def my_function():
    pass

list_all = [number1,string1, list1, dic1, my_function]
print(list_all)
print(*[type(x) for x in list_all],sep='\n')
```

Result:

```
[5, 'Hello! Test type()', [1, 2, 3], {'a': 2, 'b': 5}, <function my_function at 0x00000131CF2F7040>]

<class 'int'>

<class 'str'>

<class 'list'>

<class 'dict'>

<class 'function'>
```

This shows all of them are the instance of a class i.e. object.

dir() method

We can list out all the attributes & methods of a given object by using dir() function

#dir.py

```
number_list = [2,3]
print(dir(number_list))
```

Result:

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

id() method

In Python, every object has an id for identity. The id of an object is always unique and constant for this object during the lifetime.

#id.py

```python
number1 = 5
print(id(number1))

number2 = 6
print(id(number2))
```

Result:

```
1440120596912

1440120596944
```

Obs: The id of two different objects (number1 and number2) are different.

Modification: *If we assign number1 to number2*

```python
number1 = 5
print(id(number1))

number2 = number1
print(id(number2))
```

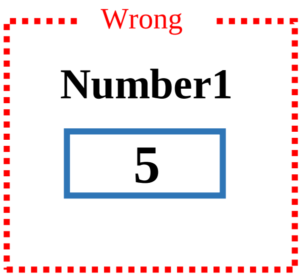
Result:

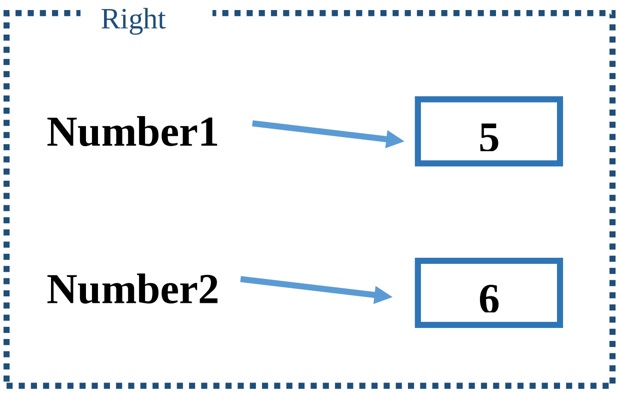
```
1771266861488

1771266861488
```

Obs: Both the ids are same as both the variables are pointing towards same object. **Python does this for memory optimisation.**

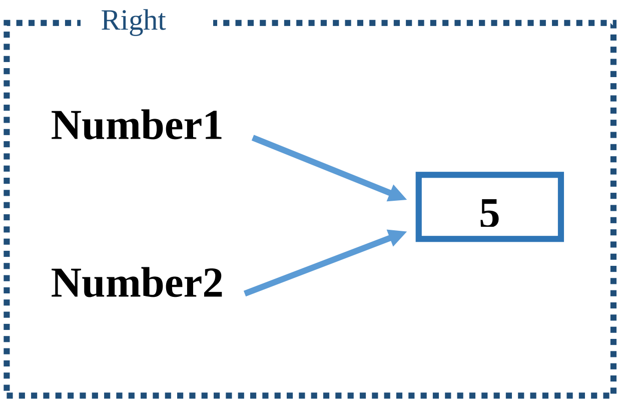
How variables actually works?

We say for previous example that “5 is stored in number1”. That’s wrong! Actually variable (number1 in this case) are **NAME_Tags only** that refers to the object whose id is – **1771266861488** and value is **5.**





And when Number1= Number2



List.copy() method

```
a = [1,2,3]
b=a    # expection to store value of a into b

a.append(4) # expection to change a only

print("a = ",a)
print("b = ",b)
```

Result:

a = [1, 2, 3, 4]

b = [1, 2, 3, 4]

Obs: Though it is expected to change only list a, b also gets modified as they are referring to the same object. That's why we use **list.copy()** method to avoid this behavior

```python
a = [1,2,3]
b=a.copy()

a.append(4)

print("a = ",a)
print("b = ",b)
```

Result:

```
a =  [1, 2, 3, 4]

b =  [1, 2, 3]
```