(7.1) Constructor, Destructor, __str__

1. Constructor

constructor is a magic method or special method which is called automatically whenever a new instance is created (Instantiation)
Constructor is optional and if we are not providing any constructor then python will provide default constructor.
The main purpose of constructor is to declare and initialize instance variables.
Per object constructor will be exeucted only once.
Constructor can take atleast one argument(atleast self)

In Pyhton we have two constructors

__new__ -> it creates object name space, used for memory allocation

__init__ -> it is used to set default attributes at object creation time

Note: object class is super class of all class

In [3]:
```
class A:

    #----------Constructor---------------class method-------------
    def __new__(cls,*args, **Kwargs):
        print("Creating an Instance / self")
        return super(A,cls).__new__(cls) #it will create self

    #---------Initilizer / Constructors----Instance Method--------
    def __init__(self, name):
        print("Initilizer is called")
        print("Setting name to current instance")
        self.name = name
        # name --> local variable
        # self --> instance
        # self.name --> instance variable / object variable

    #---------Normal Instance Methods----------------------------
    def get_name(self): # getters
        return self.name
    def set_name(self, new_name): # setters
        self.name = new_name
        # new_name - local variable
        # self.name - instance variable
```

Internally

step 1. A.__new__(A,'Pankaj')
step 2. (create self) --> A.__init__(self,'Pankaj')
step 3. self.name = 'Pankaj'

In [4]:
```
c1 = A('Pankaj')
```

```
Creating an Instance / self
Initilizer is called
```

```
Setting name to current instance
```

In [5]:
```
c2 = A('Sachin')
```

```
Creating an Instance / self
Initilizer is called
Setting name to current instance
```

In [8]:
```
print(c1.get_name()) # getter
print(c2.get_name())
```

```
Pankaj
Sachin
```

In [10]:
```
c1.set_name('pankaj yadav') # setter
print(c1.get_name())

c2.set_name('Sachin Yadav') # setter
print(c2.get_name())
```

```
pankaj yadav
Sachin Yadav
```

In [11]:
```
print(c1,id(c1))
print(c2,id(c2))
```

```
<__main__.A object at 0x0000025818DD7D90> 2577397546384
<__main__.A object at 0x0000025818E0ED60> 2577397771616
```

Calling (new) and (init) seperate

In [12]:
```
c3 = A.__new__(A) # c3 -> self
# c3.__new__(A) #self creation
```

```
Creating an Instance / self
```

In [13]:
```
c3.__init__('Rishi') # A.__init__(c3, 'Rishi')
```

```
Initilizer is called
Setting name to current instance
```

In [14]:
```
print(c3,id(c3))
```

```
<__main__.A object at 0x0000025818E0E100> 2577397768448
```

Class with Default constructor

In [29]:
```
class A:
    def get_name(self):
        return self.name
    def set_name(self, new_name):
        self.name = new_name

a = A()
# A.__new__(A) --> self --> A.__init__(A) # Default Constructors
a.set_name('Pankaj')
a.get_name()
```

Out[29]:
```
'Pankaj'
```

2. Destructor

Destructor is a special method and the name should be __del__

They are called whenever a object is deleted

In [34]:
```
class Person:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name.title()
    def __del__(self):
        print("I am destructors!! I will destroy you")
        del self.name
        del self
```

In [35]:
```
p1 = Person('Pankaj')
p2 = Person('Sachin')

print(p1)
print(p2)
```

```
Pankaj
Sachin
```

In [36]:
```
del p1
```

```
I am destructors!! I will destroy you
```

In [37]:
```
p2 = "Hello World"
```

```
I am destructors!! I will destroy you
```

In [38]:
```
p2
```

Out[38]:
```
'Hello World'
```

Garbage Collectors

it automatically deletes unused or orphan objects from main memory (global)

orphan objects are those objects which does not have any reference or variable

Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).Once destructor execution completed then Garbage Collector automatically destroys that object.

Note: The job of destructor is not to destroy object and it is just to perform clean up activities

In [33]:
```
x = 5
# x --> int instance --> (value = 5)
x = "hello world"
# x --> str instance --> (value = "hello world")
```

Note

If the object does not contain any reference variable then only it is eligible fo GC. ie if the reference count is zero then only object eligible for GC

In [44]:
```
import time

class Test:
    def __init__(self):
        print("Constructor Execution...")
    def __del__(self):
        print("Destructor Execution...")

t1=Test()
t2=t1
t3=t2
del t1
time.sleep(3)
print("object not yet destroyed after deleting t1")
del t2
time.sleep(3)
print("object not yet destroyed even after deleting t2")
print("I am trying to delete last reference variable...")
del t3
```

```
Constructor Execution...
object not yet destroyed after deleting t1
object not yet destroyed even after deleting t2
I am trying to delete last reference variable...
Destructor Execution...
```

How to find the number of references of an object

sys.getrefcount(objectreference)

In [45]:
```
import sys

class Test:
    pass

t1=Test()
t2=t1
t3=t1
t4=t1
print(sys.getrefcount(t1))
```

```
5
```

Note: For every object, Python internally maintains one default reference variable self

3. __str__ magic method

Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format

<__main__.classname object at 0x022144B0>

To return meaningful string representation we have to override __str__() method

In [50]:
```
class A:
    def __init__(self, name): # parameterized Constructors
        self.name = name
    def __str__(self):
        """
            __str__ magic method is responsible for printing your object
            it should always return a string
        """
        return self.name.title()
```

In [51]:
```
a = A('Pankaj yadav')
print(a) # __str__
```

```
Pankaj Yadav
```

In [55]:
```
repr(a) # default representation
```

Out[55]:
```
'<__main__.A object at 0x0000025818EF91C0>'
```

In [56]:
```
a
```

Out[56]:
```
<__main__.A at 0x25818ef91c0>
```

Difference between str() and repr() OR Difference between **str**() and **repr**()

str() internally calls __str__() function and hence functionality of both is same

Similarly,repr() internally calls __repr__() function and hence functionality of both is same.

str() returns a string containing a nicely printable representation object.

The main purpose of str() is for readability.It may not possible to convert result string to original object.

But repr() returns a string containing a printable representation of object

Note: It is recommended to use repr() instead of str()

In [58]:
```
import datetime
today=datetime.datetime.now()
s=str(today)#converting datetime object to str
print(s)
```

```
2022-01-27 01:21:56.066474
```

In [59]:
```
import datetime
today=datetime.datetime.now()
s=repr(today)#converting datetime object to str
print(s)
```

```
datetime.datetime(2022, 1, 27, 1, 22, 36, 550915)
```

In [ ]: