Constructor is a special method used to create and initialize an object of a class.

destructor is used to destroy the object.

we'll create a Class Student with an instance variable student name. we'll see how to use a constructor to initialize the student name at the time of object creation.

Using self, we can access the instance variable and instance method of the object.

class Student:

```
 def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')
def show(self):
        print('Hello, my name is', self.name)
# create object using constructor
s1 = Student('Emma')
s1.show()
```

Types of Constructors

In Python, we have the following three types of constructors.

- Default Constructor
- Non-parametrized constructor
- Parameterized constructor

- **Default Constructor: It does not perform any task but initializes the objects. It is an empty constructor without a body.**

```
class Employee:
    def display(self):
        print('Inside Display')
emp = Employee()
emp.display()
```

Non-Parametrized Constructor

A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.

```
class Company:
    # no-argument constructor
    def __init__(self):
        self.name = "PYnative"
        self.address = "ABC Street"
    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)
```

```
# creating object of the class
cmp = Company()
# calling the instance method using the object
cmp.show()
```

Parameterized Constructor

A constructor with defined parameters or arguments is called a parameterized constructor.
We can pass different values to each object at the time of creation using a parameterized constructor.

```
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
    # display object
    def show(self):
        print(self.name, self.age, self.salary)
# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()
kelly = Employee('Kelly', 25, 8500)
kelly.show()
```

Destructor is a special method that is called when an object gets destroyed
The special method __del__() is used to define a destructor.

For example, when we execute del object_name destructor gets called automatically and the object gets garbage collected.

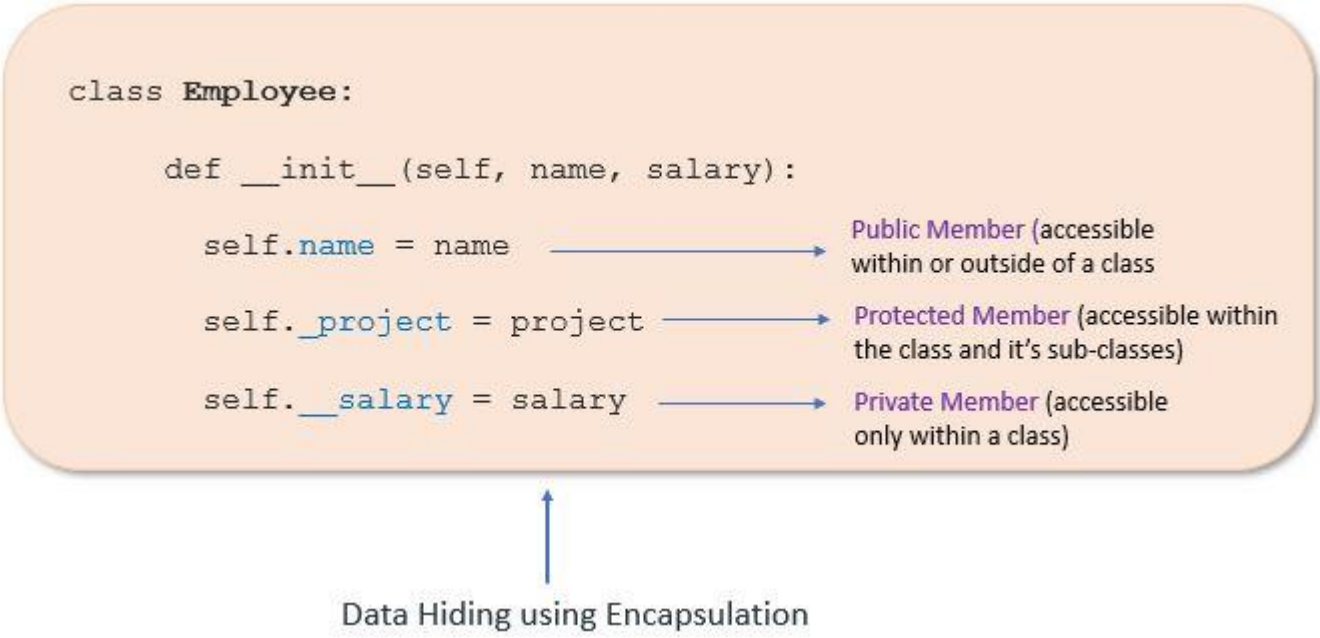
```
class Student:

    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)

    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Emma')
s1.show()

# delete object
del s1
```

Encapsulation in Python describes the concept of bundling data and methods within a single unit.

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double underscores**.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- **Public Member**: Accessible anywhere from otside oclass.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes



Data hiding using access modifiers

Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

Example:

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

Private Member

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.
Private members are accessible only within the class, and we can't access them directly from the class objects.

Example:

```
class Employee:
   # constructor
   def __init__(self, name, salary):
      # public data member
      self.name = name
      # private member
      self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

Protected Member

Protected members are accessible within the class and also available to its sub-classes.

To define a protected member, prefix the member name with a single underscore _.