Python OOPS

OOPS concepts in python are very closely related to our real world, where we write programs to solve our problems. Solving any problem by creating objects is the most popular approach in programming.

A class is a blueprint or a template for creating objects while an object is an instance or a copy of the class with actual values.

Creating a Class

Classes in Python can be defined by the keyword class, which is followed by the name of the class and a colon.

In [1]:
```
class Info:
    pass
```

In [2]:
```
class Info:
    name = "Simran"
    age = 20
```

Creating an Object

An object is an instance of a class. It is a collection of attributes (variables) and methods. We use the object of a class to perform actions.

Every object has the following property.

- **Identity:** Every object must be uniquely identified.
- **State:** An object has an attribute that represents a state of an object, and it also reflects the property of an object.
- **Behavior:** An object has methods that represent its behavior.

In [3]:
```
obj = Info()
print(obj.name)
print(obj.age)
```

```
Simran
20
```

self method

It is a reference to the current instance of the class, and is used to access variables that belongs to the class.

In [4]:
```
class Info:
    name = "Simran"
    age = 20

    def desc(self):
        print("My name is", self.name, "and my age is", self.age)

obj = Info()
obj.desc()
```

```
My name is Simran and my age is 20
```

init method (Constructor)

The **init** method is used to initialize the object's state and contains statements that are executed at the time of object creation.

In [8]:
```
class Info:
    # constructor
    # initialize instance variable
    def __init__(self, name, age,height):
         # data members (instance variables)
        print('Inside Constructor')
        self.name = name
        self.age = age
        self.height=height
        print('All variables initialized')

obj = Info("Simran", 20, 5)
# accessing instance variables
print(obj.name, "is", obj.age, "years old", obj.height)
```

```
Inside Constructor
All variables initialized
Simran is 20 years old 5
```

Destructor

A destructor is called when an object is deleted or destroyed. Destructor is used to perform the clean-up activity before destroying the object, such as closing database connections or filehandle.

In [9]:
```
class Info:

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

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

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

# create object
s1 = Info('Simran',20)
s1.show()

# delete object
del s1
```

```
Inside Constructor
Hello, my name is Simran
Inside destructor
Object destroyed
```

Creating Class with its methods and calling it using objects

In [10]:
```
class Info:
    def __init__(self, name, age, profession):
        # data members (instance variables)
        self.name = name
        self.age = age
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Age:', self.age, 'Profession:', self.profe

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'working as a', self.profession)

# create object of a class
obj = Info('Ashish', 30, 'Software Engineer')

# call methods
obj.show()
obj.work()
```

```
Name: Ashish Age: 30 Profession: Software Engineer
Ashish working as a Software Engineer
```

In [11]:
```
# modify objects and their properties

class Info:
    def __init__(self, name, age):
        self.name = name
        self.age = age

obj = Info("Simran", 20) #object created
obj.name = "Raj" # variable assigned value
print(obj.name, "is", obj.age, "years old")
```

```
Raj is 20 years old
```

In [15]:
```
# delete objects and their properties

class Info:
    def __init__(self, name, age):
        self.name = name
        self.age = age

obj = Info("Simran", 20) #object created
#print(obj.name, "is", obj.age, "years old")
del obj #object deleted
print(obj.name, "is", obj.age, "years old")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [15], in <cell line: 11>()
      9 #print(obj.name, "is", obj.age, "years old")
     10 del obj #object deleted
---> 11 print(obj.name, "is", obj.age, "years old")

NameError: name 'obj' is not defined
```

Question

Write a Python program to create a Courses class with course_name and total_students instance attributes.

Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

In [16]:
```
# base class
class Info:
    def eat(self):
        print( "I can eat!")
# derived class
class Profession(Info):
    def work(self):
        print("I am working in IT Company")

#object created
d = Profession()
# Calling members of the base class
d.eat()
# Calling member of the derived class
d.work()
```

```
I can eat!
I am working in IT Company
```

Question

Create a Bus class that inherits from the Vehicle class. The seating capacity is method of Bus class and color of bus inherits for vehicle class

Encapsulation

Encapsulation is one of the key features of oops. Encapsulation refers to the bundling of attributes and methods inside a single class.

It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve data hiding.

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.

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

In [17]:
```
class Info:

    def __init__(self,name,age,profession):
        self.name = name # Public Member
        self._age = age # Protected Member
        self.__profession = profession # Private Member

# creating object of a class
d = Info('Himani', 20, "Writer")

# accessing public member
print('Name:', d.name)

# accessing protected data members
print('Age:', d._age)

# accessing private data members
print('Profession:', d.__profession)
```

```
Name: Himani
Age: 20

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [17], in <cell line: 18>()
     15 print('Age:', d._age)
     17 # accessing private data members
---> 18 print('Profession:', d.__profession)

AttributeError: 'Info' object has no attribute '__profession'
```

Name Mangling to access private members

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this _classname__dataMember, where classname is the current class, and data member is the private variable name.

In [18]:
```
class Info:

    def __init__(self,name,age,profession):
        self.name = name # Public Member
        self._age = age # Protected Member
        self.__profession = profession # Private Member

# creating object of a class
d = Info('Himani', 20, "Writer")

# accessing public member
print('Name:', d.name)

# accessing protected data members
print('Age:', d._age)

# accessing private data members
print('Profession:', d._Info__profession)
```

```
Name: Himani
Age: 20
Profession: Writer
```

Question

Create a class name **reactangle**, the breath of the reactangle is public and length is private. Print the value of breath and length by creating object **rect**

Polymorphism

Polymorphism in Python is the ability of an object to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

In [19]:
```
s = ['ABC', 'BHG', 'CDH']
t = 'ABC School'
d = {'k':1,'p':2}

# calculate count
print(len(s))
print(len(t))
print(len(d))
```

```
3
10
2
```

Polymorphism in class methods

In [20]:
```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()

# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
    car.fuel_type()
    car.max_speed()
```

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

Polymorphism with Function and Objects

In [21]:
```
# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

Question

Give example of Polymorphism in addition operator