Object oriented programming in python:

Object-oriented Programming (OOP) is a model that uses objects and classes in programming.

It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.

Class:

Class represents the structure of something and it contain attributes and if we want to assign specific values to the attributes we create objects.

Example:

```
class Student:

   def __init__(self, name, age, number):
      self.name = name
      self.age = age
      self.number = number

c = Student("shaheer", "21", "03015916169")
print(c.name, c.number, c.age)

c1 = Student("khan", "21", "03015916169")
print(c.name, c.number, c.age)

c2 = Student("anas", "21", "03015916169")
print(c.name, c.number, c.age)
```

Another way by creating list and calling them by index

```
class Student:

   def __init__(self, name, age, number):
      self.name = name
```

```
        self.age = age
        self.number = number

students = [Student("shaheer", "22", "03015916169"), Student("khan", "22",
"03015916169")]
print(students[1].name)
print(students[0].name)
print(students[1].number)
```

Methods:

. init() method: Also known as initializer or constructor Initializer is a method or section code whenever we create a new class in python

Example:

```
class Student:

    def __init__(self, name, age, number):
        self.name = name
        self.age = age
        self.number = number
```

Parameters and Arguments:

When we are defining a method we give **parameters** to it basically the attributes that we want to have in the program

Example:

def __init__(self,name,membership):

Now here name and member ship are the parameters

But when we are assigning some values to these parameters then they become **arguments:**

```
c = Student("shaheer", "21", "03015916169")
print(c.name, c.number, c.age)
```

Creating our own method:

```
class Student:

    def __init__(self, name, age, number):
        self.name = name
        self.age = age
        self.number = number

    # creating our own method

    def update_number(self, up_number):
        self.number = up_number

students = [Student("shaheer", "22", "03015916169"), Student("khan", "22",
"03015916169")]
print(students[1].number)
students[1].update_number("03334245256")
print(students[1].number)
```

Some useful methods:

```
class Student:

    def __init__(self, name, age, number):
        self.name = name
        self.age = age
        self.number = number

    # creating our own method

    def update_number(self, up_number):
        self.number = up_number
```

```
    def __str__(self):
        return self.name

    def print_all_students(self):
        for student in self:
            print(student)



students = [Student("shaheer", "22", "03015916169"), Student("khan", "22",
"03015916169")]
print(students[1])
students[1].update_number("03334245256")
print(students[1].number)
Student.print_all_students(students)
```

Useful methods:

```
class Customer:
    def __init__(self,name,membership_type):
        self.name = name
        self.membership_type = membership_type

    def update_membership(self,new_membership):
        self.membership_type = new_membership

    def __str__(self):
        return self.name + " " + self.membership_type

    def print_all_customers(self):
        for customers in customer:
            print(customer)

    def __eq__(self, other):
        if self.name == other.name and self.membership_type ==
other.membership_type:
```

```
        return  True

    return  False
customer = Customer("Shaheer","Gold"), Customer("sherry","Luxury")
print(customer[0].membership_type,customer[1].membership_type)
customer[1].verified = True
print(customer[1].verified)  # Verification task
print(customer[1].membership_type) # old type of membership
customer[1].update_membership("Diamond")  # new type of membership
mantained
print(customer[1].membership_type) # display of new membership
print(customer[0] == customer[1])  # checking if first customer and second
customer's attribute is same or not
```

Principal of OOPs:

1. Encapsulation

   We can hide the data in the class only the data that is needed by the other class is given to it means that the whole class data is not shared with the other class only the data needed by the other class is given to it.

2. Inheritance

   Allows to provide certain attributes to objects from base class e.g. User base class has two derived classes. Customer. Teacher

3. Polymorphism

   In programming language theory and type theory, polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types

```
class User:
    def log(self):
        print(self)

class Teacher(User):
    def log(self):
        print("I m a teacher")

class Customer(User):
    def __init__(self,name,membership_type):
        self.name = name
        self.membership_type = membership_type

        @property
        def name(self):                    # Getter function
            print("Getting name")
            return self._name

        @name.setter
        def name(self,name):                # Setter function
            print("Setting name")
            self._name = name

    def update_membership(self,new_membership):
        self.membership_type = new_membership

    def __str__(self):
        return self.name + " " + self.membership_type

    def print_all_customers(self):
        for customers in self  :
            print(customers)

    def __eq__(self, other):
        if self.name == other.name and self.membership_type ==
other.membership_type:
            return  True
```

```
        return  False

    __hash__ = None
    __repr__ = __str__
users = [Customer("shaheer","bronze"), Customer("sherry","Diamond"), Teacher()]
for user in users:
    user.log()
```