In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
  # Statement-1
  .
  .
  # Statement-N
```

Creating an Empty Class in Python

```
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

Note: # is the beginning of a single line comment. Python doesn't support multi-line comments. The next way is by using string literals but not assigning them to any variables. If you do not assign a string literal to a variable, the Python interpreter ignores it. Use this to your advantage to write multi-line comments. You can either use a single (‘’) quotation or double (“”) quotation.
Eg:

‘Hello This is the value not assigned to any variable and you can also use this as a comment.’

Python Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.

- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that we will use while working with objects and classes.

Eg:

```
Obj1 = Dog()

Vehicle2 = Car()
```

The Python “self”

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to the “this” pointer in C++ and the “this” reference in Java.

==========

```
class HelloWorld {

        int Salary;

        String Name;

        HelloWorld(int consSalary, String consName){

        this.Salary = consSalary;

        this.Name = consName;

        }

        void showDetails(){

        System.out.println(this.Name+ " earns Rs."+this.Salary);

        }


        public static void main(String[] args) {

        System.out.println("Hello, World!/n");
```

```
        System.out.println();
        HelloWorld emp1 = new HelloWorld(30000, "Shyam");
        emp1.showDetails();
        }
}
```

==========

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

```
#it is clearly seen that self and obj is referring to the same
object
class check:
    def __init__(self):
        print("Address of self = ",id(self))

obj =check()
print("Address of class object = ",id(obj))
```

==========

```
class Employee:
      def setSalary(self, salary):
      self.salary = salary
      def showSalary(self):
      print("The salary is ", self.salary)

emp1 = Employee()
emp1.setSalary(1500)
```

```
emp1.showSalary()
```

```
======================================================================
```

The Python ___init___ Method

The ___init___ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Now let us define a class and create some objects using the self and ___init___ method.

Creating a class and object with class and instance attributes

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
       self.name = name

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

```
======================================================================
```

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.

- ___init___ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.
  The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The ___init___ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
       self.name = name

    def speak(self):
       print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

==========

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Inheritance in Python

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
       self.name = name
       self.idnumber = idnumber
       print("The parent code run!")

    def display(self):
       print(self.name)
       print(self.idnumber)

    def details(self):
```

```
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post
        print("The code from Child is run here!")

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()

=====================================================================
```

Python Polymorphism

Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

Polymorphism in Python

This code demonstrates the concept of inheritance and method overriding in Python classes. It shows how subclasses can override methods defined in their parent class to provide specific behavior while still inheriting other methods from the parent class.

```
class Bird:

    def intro(self):
      print("There are many types of birds.")

    def flight(self):
      print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
      print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
      print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()

=================================================================
```

Python Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Encapsulation in Python

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can't even change its value.

```
# Python program to

# demonstrate private members

# Creating a Base class

class Base:

    def __init__(self):

            self.a = "GeeksforGeeks"

            self.__c = "GeeksforGeeks"


# Creating a derived class

class Derived(Base):

    def __init__(self):


            # Calling constructor of

            # Base class

            Base.__init__(self)

            print("Calling private member of base class: ")

            print(self.__c)



# Driver code

obj1 = Base()

print(obj1.a)
```

```
# Uncommenting print(obj1.c) will

# raise an AttributeError


# Uncommenting obj2 = Derived() will

# also raise an AttributeError as

# private member of base class

# is called inside derived class

================================================================
```

Download and Install python interpreter from :
https://www.python.org/downloads/

I suggest using different environments for your projects to maintain compatibility, so use “virtualenvwrapper-win” to isolate libraries.

```
c:\> pip install virtualenvwrapper-win
```

See this page for more info:
https://sachinjose31.medium.com/virtual-environments-with-virtualenvwrapper-for-windows-c535c2a0de8c

https://virtualenvwrapper.readthedocs.io/en/latest/

https://stackoverflow.com/questions/36491776/virtualenvwrapper-creating-project-in-wrong-directory

Creating an app:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "hello there"

if __name__ == "__main__":
    app.run(debug=True)
```

If ___name___ = “___main___”In Short: It Allows You to Execute Code When the File Runs as a Script, but Not When It’s Imported as a Module.

To change port and Host in the app:

```
if ___name___ == '___main___':
   app.run(debug=True, host='localhost', port=port_number)
```

Create folder named “static” and “templates” in the same folder to use template and javascripts, css and import

```
from flask import Flask, render_template

@app.route(“/”)
def index():
        return render_template(‘index.html’)
```

Template Inheritance:: Code resume, use the template for all pages but just change content, apply for dynamic websites.

Create a template and use {% block head%}{% endblock %} to write the codes.

In the main ‘index.html’ web file use :

{% extends ‘base.html’ %}

{% block head%} <your code here>{% endblock %}

{% block body%}{% endblock %}

Example: base.html

```
    <!DOCTYPE html>

    <html lang="en">

    <head>

      <meta charset="UTF-8" />

      {% block head%}{% endblock %}

      <meta name="viewport"
content="width=device-width,initial-scale=1" />
```

```
      <meta name="description" content="" />



    </head>

    <body>

      <h1>Hello, world! Welcome to Flask Learning</h1>
      {% block body %}{% endblock %}

    </body>

    </html>
```

Index.html

```
{% extends 'base.html' %}

{% block head%}{% endblock %}

{% block body%}<h1>Template</h1>{% endblock %}
```

Example to use css in the app,

Create a folder called “css” inside a static folder and create a file called main.css inside it and write your css code there, also link that in your base.html file.

```
<link rel = "stylesheet" href = "{{url_for('static', filename
='css/main.css')}}">
```

Path looks like this static> css>main.css

```
body {
    margin:0ch;
    font-family: Verdana, Geneva, Tahoma, sans-serif;
}
```

SQLAlchemy

To use SQL in the app, import the sqlalchemy:

Code: from flask_sqlalchemy import SQLAlchemy
Now configure app for the sqlalchemy

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'Ssqlite:///test.db'
```

3-/// is relative path - if you want to reside in project location
4-//// is absolute path
Everything will be stored in test.db file

Initialize the database now:

```
db = SQLAlchemy(app)
```

Now Create the model for the Database:

```
class Students(db.Model):
    id = db.Column(db.Integer, primary_key= True)
    fname = db.Column(db.String(200), nullable=False)
    lname = db.Column(db.String(200), nullable=False)
    age = db.Column(db.Integer, nullable=False)
    faculty = db.Column(db.String(200), nullable=False)
    date_created = db.Column(db.DateTime, default = datetime.utcnow)

    def __retStudent__(self):
        return '<Student %r>' % self.id
```

now run “flask shell” in terminal and type:

```
>> from app import db
>> app.create_all()
```

Now your code in “app.py” looks something like this:

```
from flask import Flask, render_template, url_for

from flask_sqlalchemy import SQLAlchemy

from datetime import datetime


app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
db = SQLAlchemy(app)

class Students(db.Model):
```

```
    id = db.Column(db.Integer, primary_key= True)
    fname = db.Column(db.String(200), nullable=False)
    lname = db.Column(db.String(200), nullable=False)
    age = db.Column(db.Integer, nullable=False)
    faculty = db.Column(db.String(200), nullable=False)
    date_created = db.Column(db.DateTime, default = datetime.utcnow)

    def __retStudent__(self):
        return '<Student %r>' % self.id

@app.route('/')

def index():
    return render_template('index.html')

if __name__ == "__main__":
    app.run(debug=True)

#app.run(debug=True, host='localhost', port=port_number) to change port
and host
```

Now give list of methods to your app to accept :: POST and GET
@app.route(‘/’, methods = [‘POST’, ‘GET’])

and import request:

from flask import request, redirect # after success use this.

This means the route (“/”) can now accept two methods i.e POST and GET methods.
Now add if statement for each methods in the route:

```
@app.route('/', methods = ['POST', 'GET'])
def index():
    if request.method=='POST':
        new_student = Students(fname=request.form['fname'],
lname=request.form['lname'], age=request.form['age'],
faculty=request.form['faculty'])

        try:
```

```
            db.session.add(new_student)
            db.session.commit()
            return redirect('/')

        except:
            return 'Something Error.'
    else:
        students = Students.query.order_by(Students.date_created).all()
        return render_template('index.html', students=students)
```

and your forms looks like this:

```
{% extends 'base.html' %}

{% block head%}<h1>The name of the students</h1>{% endblock %}

{% block body%}
<div class = "content">
    <h2>Students</h2>
    <table>
        <tr>
            <th> ID</th>
            <th> Students Name</th>
            <th> Students Age</th>
            <th>Faculty</th>
            <th>Date Created</th>
            <th> Actions</th>
        </tr>
        {% for student in students %}
        <tr>
            <td>{{student.id}}</td>
            <td>{{student.fname, student.lname}}</td>
            <td>{{student.age}}</td>
            <td>{{student.faculty}}</td>
            <td>{{student.date_created}}</td>
            <td>
                <a href="">Delete</a>
                <br>
                <a href="">Edit</a>
            </td>
        </tr>
```

```
        {%endfor%}

    </table>
    <br>
    <form action="/" method="POST">
        <label for="fname">First name</label>
        <input type="text" name = "fname" id = "fname"><br>
        <label for="lname">Last name</label>
        <input type="text" name = "lname" id = "lname"><br>
        <label for="age">Age</label>
        <input type="text" name = "age" id = "age"><br>
        <label for="faculty">Faculty</label>
        <input type="text" name = "faculty" id = "faculty">
        <input type="submit" value = "Add New Student">
    </form>
</div>

{% endblock %}
```