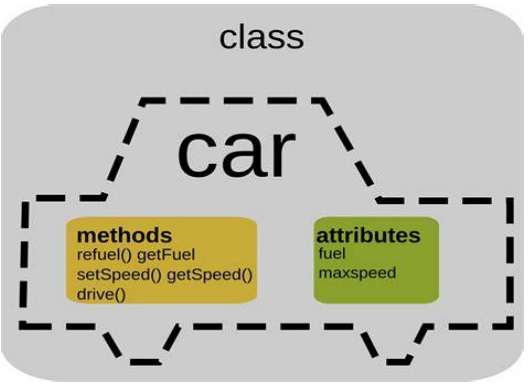
Object Oriented Programming

object oriented programming is a collection of methods and attributes of an object, object are nothing but they are an instance of a class



- **class :** are user-defined data types that act as the blueprint for individual objects, attributes and methods.
- **objects:** are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity. When class is defined initially, the description is the only object that is defined.
- **methods :** are functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for reusability or keeping functionality encapsulated inside one object at a time.

➔ **Special methods**

  - Special methods start and end with __.
  - Special methods have specific names, like __init__ for the constructor or __str__ for the conversion to string.

- **attributes :**are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field. Class attributes belong to the class itself.
- **Constructor Method:** constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the __init__() method is called the constructor and is always called when an object is created.

Syntax of constructor declaration :

```
def __init__(self):
# body of the constructor
```

RULE FOR CONSTRUCTORS

1. the constructor name should be the same as the class's name
2. the is no return type for constructors

Type of constructor

- **default constructor:** The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
- **Parameterized constructor:** constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Destructors methods : The __del__() method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.
Syntax of destructor declaration :

self in Python class

self represents the instance of the class. By using the "self" we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.

```
class Shirt:

    def __init__(self, shirt_color, shirt_size, shirt_style, shirt_price):
        self._price = shirt_price

    def get_price(self):
      return self._price

    def set_price(self, new_price):
      self._price = new_price

shirt_one = Shirt('yellow', 'M', 'long-sleeve', 15)
print(shirt_one.get_price())
shirt_one.set_price(10)
```

main principles of OOP

Object-oriented programming is based on the following principles:

- **Encapsulation:** This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.

  With data encapsulation in place, we can not directly modify the instance attribute by calling an attribute on the object. It will make our application vulnerable to hackers. However, we only change the instance attribute values by calling the specific method.

  ```
  class Product:
    def __init__(self):
      self.__maxprice = 1000
      self.__minprice = 1

    def sellingPrice(self):
  ```

```
        print('Our product maximum price is: {}'.format(self.__maxprice))
        print('Our product minimum price is: {}'.format(self.__minprice))

    def productMaxPrice(self, price):
        self.__maxprice = price

def productMinPrice(self, price):
self.__minprice = price

prod1= Product()
prod1.sellingPrice()

prod1.__max price = 1500
prod1.sellingPrice()
```

- **Abstraction**. Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.
- **Inheritance.** Classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.

```
>>> class Animal:
...     sound = ""
...     def __init__(self, name):
...         self.name = name
...     def speak(self):
...         print("{sound} I'm {name}! {sound}".format(
...             name=self.name, sound=self.sound))
...
>>> class Piglet(Animal):
...     sound = "Oink!"
...
```

```
>>> class Cow(Animal):
...     sound = "Moooo"
```

- **Polymorphisme.** Objects are designed to share behaviors and they can take on more than one form. The program will determine which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created, which extends the functionality of the parent class. Polymorphism allows different types of objects to pass through the same interface.

Creat a class

- calluses, instance and object in python
- create and instantiate class
- access variable and methods

instantiation process in Python with three key steps.

- class definition
- creating a new instance
- initializing the new instance

Classes and Instances

- Classes define the behavior of all instances of a specific class.
- Each variable of a specific class is an instance or object.
- Objects can have attributes, which store information about the object.
- You can make objects do work by calling their methods.
- The first parameter of the methods (self) represents the current instance.
- Methods are just like functions, but they can only be used through a class.

What are the benefits of OOP?

- **Modularity.** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.

- **Reusability.** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
- **Productivity.** Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
- **Easily upgradable and scalable.** Programmers can implement system functionalities independently.
- **Interface descriptions.** Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.
- **Security.** Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
- **Flexibility.** Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

Python MRO (Method Resolution Order)

Multiple inheritances mean that a single subclass can inherit more than one class, and the subclass will be authorized to access the attributes and functions unless they aren't private to that specific class. The MRO technique is used there to search the order of the classes being executed.

MRO is the order in Python where a method is looked for in the class hierarchy. Mostly it is used to look for methods and attributes in the parent classes of a subclass.

First, the attribute or method is searched within the subclass, following the specified order while inheriting. This order is also known as the Linearization of a class, and a set of rules is applied while checking the order.

While inheriting from other classes, the compiler needs a proper way to resolve the methods called via an instance of the class. It plays an important role in multiple inheritances when we have the same function in more than one parent of base classes.

Python help() Method

The Python **help()** function invokes the interactive built-in help system. If the argument is a string, then the string is treated as the name of a module, function, class, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is displayed.

```
SYNTAX : help(object)
```

object: (Optional) The object whose documentation needs to be printed on the console.

Docstring Patterns in Python

Documenting code is a good habit, and aspiring developers and programmers should develop a habit of documenting their code in the early phases of their coding journey.

Documenting a source code improves the readability and management of the source code and makes it extremely easy for new contributors to the source code to understand it.

Docstrings are string literals written inside source codes. They act as comments or documentation for pieces of code. Docstrings are used to describe classes, functions, and sometimes, even the files.

In order words, a docstring acts as metadata about the code snippets. A docstring contains all the relevant information about what they are describing. For a class, it holds information about:

- the class
- class functions
- classes attributs.

For functions, it holds details about:

- parameters
- data types of parameters
- default values of parameters
- short descriptions about parameters
- what the function returns
- data type of what is returned by the function
- errors and exceptions that the function raises and short descriptions about

There are several docstring patterns that professional Python developers use to document their code.

Instead of using the existing ones, one can create their docstring pattern. Still, this decision solely depends on the individual developer or the team of developers.

This article will tackle the best docstring patterns for the Python programming language.

1. Epytext Pattern
2. reSt Pattern
3. Google Pattern
4. Numpydoc Pattern

the best one i use it all the time is Google Pattern

Google Pattern

Technically, its name is not Google's pattern, but it is a pattern that Google developed.

It is a clean pattern that organizes details under headings. The Sphinx tool is capable of recognizing this pattern too and generating documentation.

This pattern is one of the most docstrings patterns too. Following is an example of the Google pattern.

```
"""
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore e

Args:
    parameter1: this is the first parameter.
    parameter2: this is the second parameter.

Returns:
    this is a description of what is returned by the function.

Raises:
    KeyError: raises an exception.
    TypeError: raises an exception.
"""
```

The description is followed by heading such as **Args**, **Returns**, and **Raises**. Under the **Args** heading, all the parameters and details, such as their type and default values, are placed.

A description of what is returned by the function is placed under the **Returns** heading. Lastly, errors or exceptions and their details are written under the **Raises** heading.

example 👍

```
class Pants:
    """The Pants class represents an article of clothing sold in a store
    """

    def __init__(self, color, waist_size, length, price):
        """Method for initializing a Pants object

        Args:
            color (str)
            waist_size (int)
            length (int)
            price (float)

        Attributes:
            color (str): color of a pants object
            waist_size (str): waist size of a pants object
            length (str): length of a pants object
            price (float): price of a pants object
        """

        self.color = color
        self.waist_size = waist_size
        self.length = length
        self.price = price

    def change_price(self, new_price):
        """The change_price method changes the price attribute of a pants
object

        Args:
            new_price (float): the new price of the pants object

        Returns: None

        """
        self.price = new_price

    def discount(self, percentage):
        """The discount method outputs a discounted price of a pants object

        Args:
            percentage (float): a decimal representing the amount to
discount

        Returns:
```

```
    float: the discounted price
"""
return self.price * (1 - percentage)
```