Attributes?

# What is OOP?

Object oriented programming that attempts to model to model some process or thing in the world as a class or object.

**Class –** A blueprint for objects. Classes can contain methods (functions) and attributes (similar to keys in a dict).

**Instance –** Objects that are constructed from a class blueprint that contain their class’s methods and properties.

# Why OOP?

With object-oriented programming, the goals is to encapsulate your code into **logical, hierarchical groupings using classes** so that you can reason about your code at a higher level.

# Encapsulation and Abstraction

**Encapsulation –** The grouping of public and private attributes and methods into a programmatic class, making abstraction process.

**Abstraction –** Exposing only “relevant” data in a class interface, hiding private attributes and methods (“the inner workings”) from users.

Example

* Designing the Deck class, we make the **cards** a private attribute (a list)
* We decide the length of the cards should be accessed via a public method called count (), ie. Deck.count()

# Creating a Class

-Classes are created using the class method

Classes in python can have a special \_\_init\_\_ method, which gets called every time you create an instance of the class (instantiate).

# Self

class Vehicle:

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

This self keyword refers to the current class instance.

Self-instance always be the first parameter to \_\_init\_\_ and any methods and properties on class instances.

# Attribute Name (Underscores)

1. **\_attribute** – **A Private attribute** (only be convention and not enforced). We use this when we want to convey that this attribute is meant to be private, i.e. only accessible from inside the class definition.
2. **\_\_attribute – Name mangling** Python will internally mangle (change) the name and add the class name in front of it, so it’s not directly accessible. Its sole purpose is to make it unique to the class it’s defined in and not subclasses (inheritance).
3. **\_\_method\_\_ - Magic methods /dunder methods.** Special methods that aren’t meant to be invoke directly by you, but invocation happens internally from within the class definition.

# Instance attributes

Classes can have variables defined inside of them. These are called attributes.

They can be set on the \_\_init\_\_() constructor and are unique to their instances.

We can access these attributes using the **dot notation** syntax on a class instance. For example

For example, **Instance.attribute\_name**

# Instance Methods

Classes can have functions defined inside them.

They are called methods and will only work on its instance unlike normal functions.

We use the **dot notation** syntax style to use it.

Example, **instance. methon\_name ()**

class User:

def \_\_init\_\_ (self, first, last):

self.first = first

self.last = last

def full\_name(self):

return f"{self.first} {self.last}"

def likes(self, thing):

return f"{self.first} likes {thing}"

user1 = User ("Jack", "Shephard")

user. full\_name ()

user. Likes ("Python")

# Inheritance

A core feature of OOP, inheritance allows us to define a class which inherits from another class (a “base” or “parent”)

**Child/Sub Classes** will inherit both attributes and methods from it’s **Base/Parent** classes. Child/sub classes can also overwrite attributes and methods of its parent classes.

# Using Inheritance

To create and use this inheritance relationship, we pass the base class **as an argument** to the definition of the sub class.

#Inheritance

#Base Class

class User:

#constructor

def \_\_init\_\_(self, name, email):

self.name = name

self.email = email

def login(self):

return f"{self.name} has logged in"

#child class Admin inherits from User class

class Admin(User):

def \_\_init\_\_(self, name, email, password):

self.name = name

self.email = email

self.password = password

def delete\_user(self):

return f"You have deleted a user"

piyush = User("Piyush", "bhavsarp456@gmail.com")

paresh = Admin("Paresh", "bparesh13@gmail.com",12354)

print(piyush.name)

print(paresh.email)

print(paresh.name,paresh.email,paresh.password)

print(piyush.login())

print(paresh.login())

print(paresh.delete\_user())

# Properties

We can turn a regular attribute to a property to make getting and setting it much easier.

We have to use the @property decorator on the getter method to achieve this

class Animal:

    def \_\_init\_\_(self, species, name):

        self.\_species = species

        self.name = name

    @property

    def species(self):

        return self.\_species

    @species.setter

    def species(self, new\_species\_name):

        self.\_species = new\_species\_name

beerus = Animal("Tiger", "Berus the distroyer")

print(beerus.species)

beerus.species = "Human"

print(beerus.species)

# Super()

Super() refers to the base/parent class.

We could also use the **Base** class directly and pass self instead of Using super().

class Animal:

    def \_\_init\_\_(self, species, name):

        self.species = species

        self.name = name

class Dog(Animal):

    def \_\_init\_\_(self, name, breed, age):

        super().\_\_init\_\_("dog", name)

        self.breed =breed

        self.age = age

dexter = Dog("Dexter", "hound", 30)

print(dexter.species)

print(dexter.breed)

We could have used

BASECLASS.\_\_init\_\_(self,…)

Don’t use this

# Multiple Inheritance

**Sub** classes can inherit from multiple **base** classes.

Not recommended as this can introduce a lot of unpredictable and hard to find bugs.

The **method resolution order (MRO)** decides where to look for attributes and methods if not present in the sub class.

class Human:

    #code

class Animal:

    #code

class Mutant(Human, Animal):

    #code

    #inherits attributes and methods

    #from both of its base classes

