Machine Learning Diploma

Python4: Object Oriented and Python Exercises

AMIT

Agenda

- ➔ Procedural Programming vs Object Oriented Programming
- ➔ Features of OOP
- ➔ Class and Objects
- ➔ Methods in Classes
- ➔ Fundamentals concepts of OOP
- ➔ Exercises on Python concepts.

1. Procedural Programming vs Object Oriented Programming

AMIT

Procedural vs Object Oriented Programming:

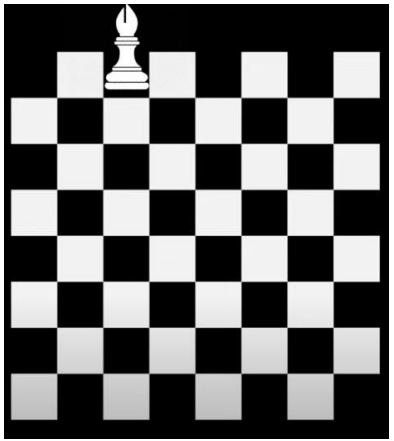
- ➔ Procedural Programming is a programming paradigm that follows a step-by-step approach to break down a task into a collection of variables and functions through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do.
- ➔ Object-oriented Programming is a programming paradigm that uses classes and objects to create models based on the real-world environment. In OOPs it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones. In OOPs concept of objects and classes is introduced and hence the program is divided into small chunks called objects which are instances of classes.

AMIT

Procedural vs Object Oriented Programming

Let’s imagine we will write a program to stimulate chess game and we need to write the code of the elephant piece. we will have a variable called position to indicate his position, another variable called color to indicate his white or black color and third value Boolean to indicate if the elephant is still alive. In procedural programming our code will be very long with 3 lines to each elephant only !! For example

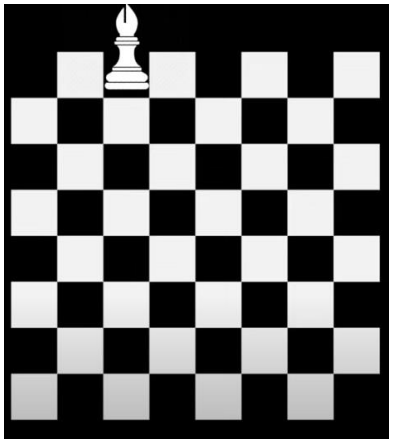
➔ Ele_1_poition = 3 , die_or_not = False , color= ‘white’



AMIT

Procedural vs Object Oriented Programming

And we will repeat these lines for each elephant in the game , and for each piece in the game. Let’s imagine if we can treat every element in the game as an object , the code will be more organized and easier.



2. Features of OOP

Features of OOP

- ➔ Ability to simulate real-world events much more effectively.
  - OOP languages allow you to break down your software into bite-sized problems that you then can solve — one object at a time.
- ➔ Code is reusable thus less code may have to be written.
  - Suppose that in addition to your Car object, one colleague needs a Race Car object, and another needs a Limousine object. Everyone builds their objects separately but discover commonalities between them
  - Create one generic class (Car), and then define the subclasses (Race Car and Limousine) that are to inherit the generic classes' traits.

Features of OOP

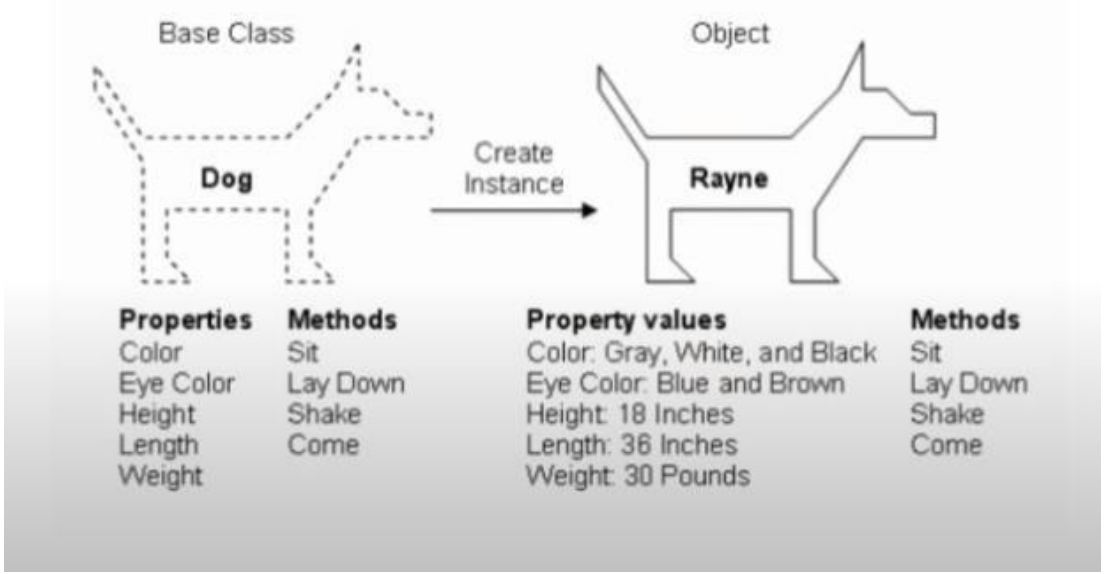
- ➔ Modularity for easier troubleshooting.
  - as you know exactly where to look. “Oh, the car object broke down? The problem must be in the Car class!” You don’t have to muck through anything else.

3. Class and Objects

Class and object

- ➔ A class is a special data type which defines how to build a certain kind of object.
- ➔ The class also stores some data items that are shared by all the instances of this class or will be specific to each object of this class.
- ➔ Instances are objects that are created which follow the definition given inside of the class.
- ➔ “Class is the blueprint of the object”

Class and object



AMIT

OOP Concepts (Attributes and Methods)

The attributes are the things that the object will have, and they are basically just variables that are associated with the final object. So for example, if you're constructing a car object, then one of those attributes could be the color it has, and may initialize it at the constructor to be blue.

Methods are the functions that belongs to an object , it defines what will the object be able to do

Ex: a Parrot class has attributes age, name, type and singing and dancing as methods.

OOP Concepts (Constructor)

Constructor is a part of the blueprint that allows us to specify what should happen when our object is being constructed, and this is also known in programming as initializing an object. When the object is being initialized, we can set variables or counters to their starting values.

In Python, the way that we would create the constructor is by using a special function, which is the init function.
The important thing to remember is that the init function is going to be called every time you create a new object from this class.

OOP Concepts (Constructor)

```
class Dog:


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


AMIT

How to create a class

- ➔ To define a class in Python, you can use the class keyword, followed by the class name and a colon. Inside the class, an __init__ method must be defined with def.
- ➔ This is the initializer that you can later use to instantiate objects. It's like a constructor in Java.
- ➔ __init__ must always be present! It takes one argument: self, which refers to the object itself. Inside the method, the pass keyword is used as of now, because Python expects you to type something there , self in Python is equivalent to this in C++ or Java.
- ➔ In this case, you have a (mostly empty) Dog class, but no object yet

How to create a class

➔ In this case, you have a (mostly empty) Dog class, but no object yet

```
class Dog:


    def __init__(self):
        pass
```

Instantiating objects

➔ To instantiate an object, type the class name, followed by two brackets. You can assign this to a variable to keep track of the object.

➔ Now we create an instance of the class dog as an object called ozzy.

```
ozzy = Dog()
```

```
print(ozzy)
```

```
<__main__.Dog object at 0x111f47278>
```


AMIT

Adding attributes to a class

➔ After printing ozzy, it is clear that this object is a dog. But you haven't added any attributes yet. Let's give the Dog class a name and age, by rewriting it:

```
class Dog:


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


AMIT

Adding attributes to a class

➔ You can see that the function now takes two arguments after self: name and age. These then get assigned to self.name and self.age respectively. You can now create a new ozzy object, with a name and age.

```
ozzy = Dog("Ozzy", 2)
```

➔ To access an object's attributes in Python, you can use the dot notation. This is done by typing the name of the object, followed by a dot and the attribute's name.

```
print(ozzy.name)

print(ozzy.age)
```

```
Ozzy

2
```


AMIT

Instance attributes and class attributes

- ➔ Class attributes are the variables defined directly in the class that are shared by all objects of the class, we can access it using class name as well as using object with dot notation, e.g. classname.class_attribute or object.class_attribute , Changing value by using classname.class_attribute = value will be reflected to all the objects.
- ➔ Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor using the self parameter, Accessed using object dot notation e.g. object.instance_attribute , Changing value of instance attribute will not be reflected to other objects.

Instance attributes and class attributes

```
class dog():
    num_dogs = 0 # num_dogs class attribute
    def __init__(self,name,age):
        dog.num_dogs +=1
        self.namee = name # namee is instance attribute
        self.agee = age
```

```
print(dog.num_dogs)

dog1 = dog("sokr",4)
print(dog.num_dogs)
print(dog1.num_dogs)

print(dog1.namee)
print(dog1.agee)
```

```
0
1
1
sokr
4
```


AMIT

Quiz:

➔ Create a class for students. With student attributes of name, age,number of enrolled course and a counter to count the number of added/stored students. Define a student and print his characteristics.

Quiz (Solution):

➔ Create a class for students. With student attributes of name, age,number of enrolled course and a counter to count the number of added/stored students. Define a student and print his characteristics.

```
class student:
    num_students = 0
    def __init__(self,name,age,courses):
        self.st_name = name
        self.st_age = age
        self.st_num_courses = courses
        student.num_students +=1

ahmed = student("ahmed",22,7)
print(ahmed.st_name)
print(ahmed.st_age)
print(ahmed.st_num_courses)
print(student.num_students)
```


AMIT

4. Methods in Classes

Define methods in a class

- ➔ Now that you have a Dog class, it does have a name and age which you can keep track of, but it doesn't actually do anything. This is where instance methods come in. You can rewrite the class to now include a bark() method. Notice how the def keyword is used again, as well as the self argument.
- ➔ The bark method can now be called using the dot notation, after instantiating a new ozzy object. The method should print "bark bark!" to the screen. Notice the parentheses (curly brackets) in .bark(). These are always used when calling a method. They're empty in this case, since the bark() method does not take any arguments.

Define methods in a class

```
class dog():
    num_dogs = 0 # num_dogs class attribute
    def __init__(self,name,age):
        dog.num_dogs +=1
        self.namee = name # namee is instance attribute
        self.agee = age

    def bark(self):
        print("bark bark")
```

```
dog1 = dog("sokr",4)
dog1.bark()
```

Output:

```
bark bark
```


AMIT

Practice

- ➔ Let’s implement the doginfo() method in the Dog class, that’s print the whole information of the object that belongs to class dog.
- ➔ Then create 3 objects of the class dog with different attributes and print their information using the new method

Practice

➔ Let's implement the doginfo() method in the Dog class, that's print the whole information of the object that belongs to class dog.

```
class dog():
    num_dogs = 0 # num_dogs class attribute
    def __init__(self,name,age):
        dog.num_dogs +=1
        self.namee = name # namee is instance attribute
        self.agee = age

    def bark(self):
        print("bark bark")

    def doginfo(self):
        print("this dog's name is", self.namee, "and age is", self.agee)
```


AMIT

Practice

➔ Let's implement the doginfo() method in the Dog class, that's print the whole information of the object that belongs to class dog.

➔ Then create 3 objects of the class dog with different attributes and print their information using the new method

```
dog1 = dog("sokr",4)
dog2 = dog("nemo", 2)
dog3 = dog("romeo", 3)
dog1.doginfo()
dog2.doginfo()
dog3.doginfo()
```

```
this dog's name is sokr and age is 4
this dog's name is nemo and age is 2
this dog's name is romeo and age is 3
```


AMIT

Quiz:

➔ Add to the created student class an instance method that prints out the student info stored.

Quiz (Solution):

➔ Add to the created student class an instance method that prints out the student info stored.

```
class student:
    def __init__(self,name,age,courses):
        self.st_name = name
        self.st_age = age
        self.st_num_courses = courses
    def info(self):
        print("student", self.st_name,"is", self.st_age,
        "years old and enrolled in", self.st_num_courses, "courses")

ahmed = student("ahmed",22,7)
ahmed.info()
```


AMIT

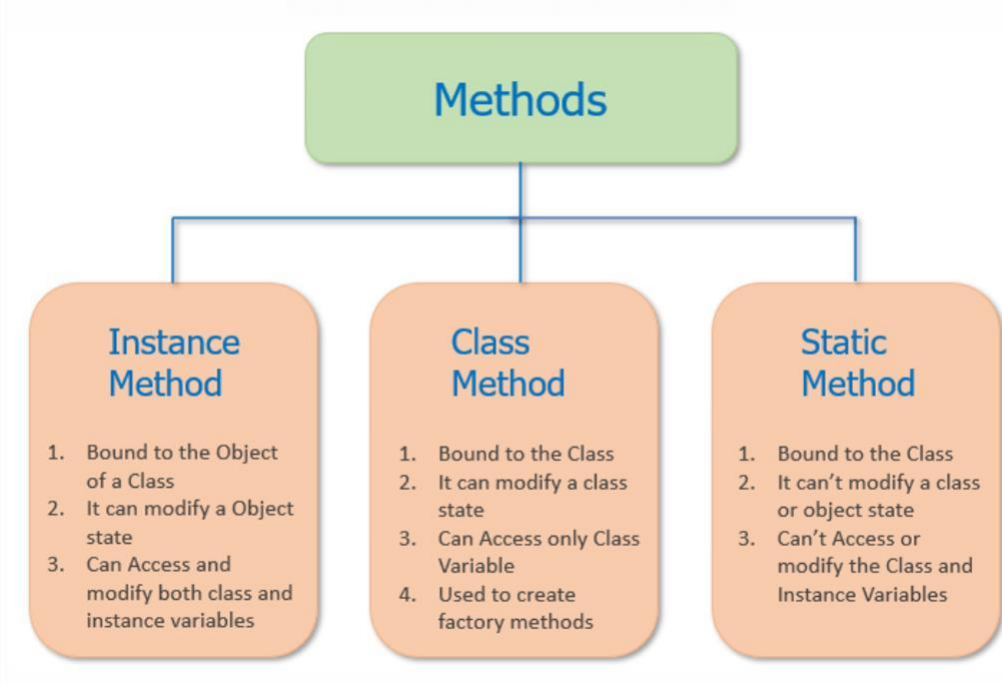
Instance Methods vs Class Methods vs Static Methods

➔ Instance method performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods.

➔ Class method is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.

➔ Static method is a general utility method that performs a task in isolation. This method doesn't have access to the instance and class variable.

Instance Methods vs Class Methods vs Static Methods



AMIT

Instance Methods vs Class Methods vs Static Methods (Usage)

➔ The instance method acts on an object's attributes. It can modify the object state by changing the value of instance variables.

➔ Class method Used to access or modify the class state. It can modify the class state by changing the value of a class variable that would apply across all the class objects.

➔ Static methods have limited use because they don't have access to the attributes of an object (instance variables) and class attributes (class variables). However, they can be helpful in utility such as conversion form one type to another.

Instance Methods vs Class Methods vs Static Methods (definition)

- ➔ All three methods are defined inside a class, and it is pretty similar to defining a regular function.
- ➔ Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method or static method.
- ➔ Use the @classmethod decorator or the classmethod() function to define the class method
- ➔ Use the @staticmethod decorator or the staticmethod() function to define a static method.

Instance Methods vs Class Methods vs Static Methods (definition)

- ➔ Use self as the first parameter in the instance method when defining it. The self parameter refers to the current object.
- ➔ On the other hand, Use cls as the first parameter in the class method when defining it. The cls refers to the class.
- ➔ A static method doesn't take instance or class as a parameter because they don't have access to the instance variables and class variables.

```python
class Student:
    # class variables
    school_name = 'ABC School'

    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age

    # instance variables
    def show(self):
        print(self.name, self.age, Student.school_name)

    @classmethod
    def change_School(cls, name):
        cls.school_name = name

    @staticmethod
    def find_notes(subject_name):
        return ['chapter 1', 'chapter 2', 'chapter 3']
```


AMIT

Quiz:

➔ Add to the created student class:

- an instance method that prints out the student info stored.
- A class method to print the number of students enrolled.
- A static method to print a given list of courses.

Quiz (Solution):

➔ Add to the created student class:

- an instance method that prints out the student info stored.
- A class method to print the number of students enrolled.
- A static method to print a given list of courses.

```
class student:
    no_of_students = 0
    def __init__(self,name,age,courses):
        self.st_name = name
        self.st_age = age
        self.st_num_courses = courses
        student.no_of_students += 1

    def show_info(self):
        print(f"student name is {self.st_name} his age is {self.st_age} and he have {self.st_num_courses} and his id is {student.no_of_stud

    @classmethod
    def tot_num_std(cls):
        print(f"the total number of students is {student.no_of_students}")

    @staticmethod
    def courses_list(list_courses):
        for i in list_courses:
            print (f"i'm learning {i}")
```


AMIT

5. Fundamentals concepts of OOP

Four Major Concepts of OOP

- ➔ Encapsulation
- ➔ Data Abstraction
- ➔ Inheritance

Encapsulation

- ➔ Encapsulation in Python is the process of wrapping up variables and methods into a single entity. In programming, a class is an example that wraps all the variables and methods defined inside it.
- ➔ Encapsulation acts as a protective layer by ensuring that, access to wrapped data is not possible by any code defined outside the class in which the wrapped data are defined. Encapsulation provides security by hiding the data from the outside world.
- ➔ In Python, Encapsulation can be achieved by declaring the data members of a class either as private or protected.

Encapsulation (private)

- ➔ If we declare any variable or method as private, then they can be accessed only within the class in which they are defined.
- ➔ In Python, private members are preceded by two underscores.
- ➔ In the example, 'length' and 'breadth' are the two private variables declared and can be accessed within the class 'Rectangle'.

```
class Rectangle:

    __length = 0 #private variable
    __breadth = 0#private variable
    def __init__(self):
        #constructor
        self.__length = 5
        self.__breadth = 3
        #printing values of the private va
        print(self.__length)
        print(self.__breadth)

rec = Rectangle() #object created for the
#printing values of the private variable o
print(rec.length)
print(rec.breadth)
```

```
5
3

---------------------------------------------------------------------------
AttributeError                            Traceback (most rec
~\AppData\Local\Temp/ipykernel_15288/4290515323.py in <module
     13 rec = Rectangle() #object created for the class 'Rect
     14 #printing values of the private variable outside the
lass 'Rectangle'
---> 15 print(rec.length)
     16 print(rec.breadth)

AttributeError: 'Rectangle' object has no attribute 'length'
```

Encapsulation (protected)

- ➔ Protected members can be accessed within the class in which they are defined and also within the derived classes.
- ➔ In Python, protected members are preceded by a single underscore.
- ➔ In the example, 'length' and 'breadth' are the two protected variables defined inside the class 'Shape'.

```
class Shape:#protected variable
    _length = 10
    _breadth = 20

class Circle(Shape):
    def __init__(self):
        #printing protected var
        print(self._length)
        print(self._breadth)

cr = Circle()
#printing protected variablesou
print(cr.length)
print(cr.breadth)
```

```
10
20

---------------------------------------------------------------
AttributeError                            Traceback (most r
~\AppData\Local\Temp/ipykernel_15288/3998163607.py in <modu
     11 cr = Circle()
     12 #printing protected variablesoutsidethe class 'Shap
---> 13 print(cr.length)
     14 print(cr.breadth)

AttributeError: 'Circle' object has no attribute 'length'
```

Abstraction

➔ Abstraction in Python is the process of hiding the real implementation of an application from the user and emphasizing only on usage of it.

➔ Through the process of abstraction in Python, a programmer can hide all the irrelevant data/process of an application in order to reduce complexity and increase efficiency.

Inheritance

- ➔ Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more.
- ➔ Inheritance is a powerful feature in object oriented programming by defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.
- ➔ Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.
- ➔ To override a method defined on the parent class, you define a method with the same name on the child class.

Inheritance

```
class person:
    def __init__(self,first,last):
        self.first_name = first
        self.last_name = last
    def print_name(self):
        print(self.first_name + " " + self.last_name)

x = person("ahmed",'mohamed')
x.print_name()

class student(person):
    def __init__(self,first,last,year):
        super().__init__(first,last)
        self.graduation_year = year
    def welcome(self):
        print("Welcome", self.first_name, self.last_name, "to the class of", self.graduation_year)

z = student("ahmed","mohamed",2019)
z.welcome()
```


AMIT

6. Exercises on Python concepts

Exercises Links:

- ➔ https://www.w3resource.com/python-exercises/
- ➔ https://www.practicepython.org/
- ➔ https://www.geeksforgeeks.org/python-exercises-practice-questions-and-solutions/

AMIT

Any Questions?

THANK YOU!

AMIT