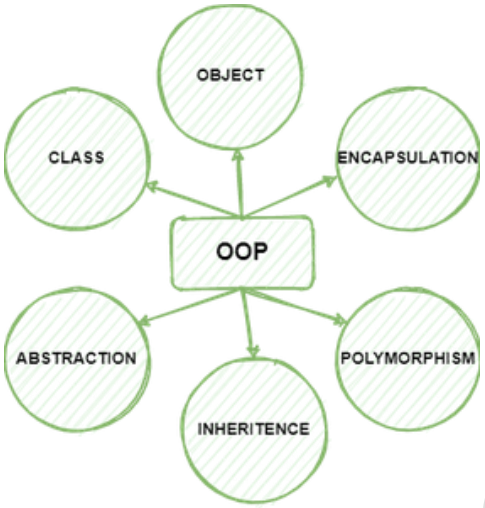
Table of Contents

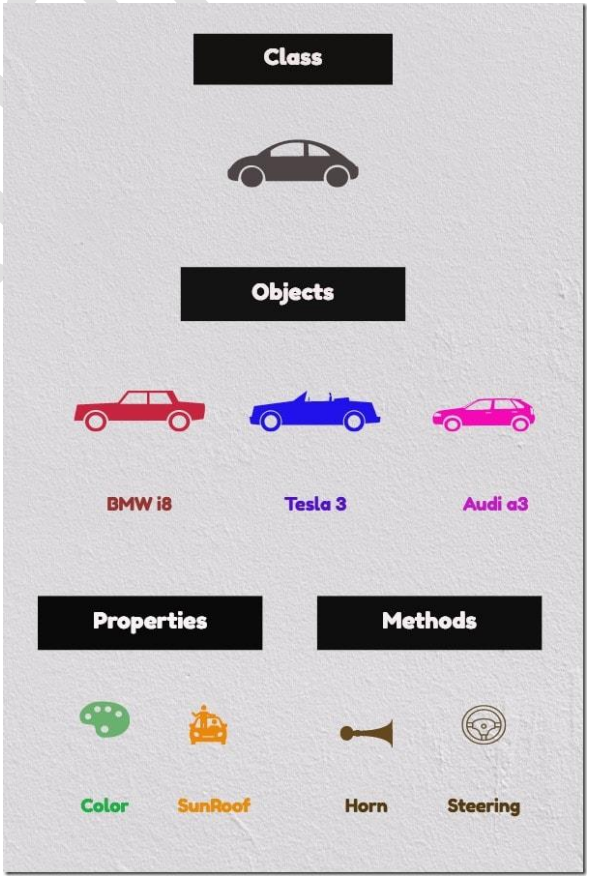
Chapter 12

OOP(Object Oriented Programming)



DRY-Don't Repeat Yourself

Python is an object oriented programming language.
Almost everything in Python is an object, with its properties and methods.
A Class is like an object constructor, or a "blueprint" for creating objects.
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.
Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. In this tutorial, you'll learn the basics of object-oriented programming in Python.



Concepts

1 Object - a variable or a method with attributes
2 Class - a definition of an object's attributes and methods
3 Polymorphism - having one common name and many forms
4 Encapsulation - hiding attributes or methods as needed
5 Inheritance - makes new object from parent attributes and methods

Some common relations:
- Inheritance makes code reuse possible i.e. create a child class
- Encapsulation - hides the internals of a class to make it 'simple' to use
- Polymorphism - makes it possible to use the 'same method name' while selecting a specific one based on context or arguments
- Concrete - a class with implementation (used for tight coupling(not recommended))
- Abstract - a class with no implementation (used for loose coupling)
- Abstraction - defines a method (category) with no implementation to allow for the needed 'loose coupling'
- Coupling - determined based on argument being abstract or concrete
- Tight Coupling - argument is a concrete class
- Loose Coupling - argument is an abstract class

Create a Class

To create a class, use the keyword class:
Ex.

```
# Create a Class
class perfect:
   name="Madi"
```

Create Object

Memory is allocated when the object is created
Ex.

```
# Create an object
p = perfect()
print(p.name)
```

Learn with Grammar:

```
# Noun      ->  Class      -> Employee
# Adjective ->  Attributes -> name, age, salary
# Verbs     ->  Methods    -> getSalary(), getAge()
```

```
# Ex2
class flightForm:
   formType="flightForm"
   def printData(self):
      print(f"Name of the person is:{self.name}")
      print(f"Flight of the person is:{self.flight}")

madiApplication = flightForm()
madiApplication.name="Madi"
madiApplication.flight="AirIndia"
madiApplication.printData()
```

Class Attributes

```
# An attributes that belongs to a class rather than object
class clsAttri:
   company = "Wipro"
madi2 = clsAttri() # Wipro
print(madi2.company)
clsAttri.company = "Amazon"
print(madi2.company) # Amazon
```

Instance

```
#  An individual object of a certain class. An object obj that belongs to a class
Circle, for example, is an instance of the class Circle. Instantiation − The creation of
an instance of a class. Method − A special kind of function that is defined in a class
definition.
class employee:
   company = "Infosys"
   salary = 987556

omar = employee()
neuro = employee()
print(omar.company) # Infosys
print(neuro.company) # Infosys

# Create instance attribute
# An instance attribute is a Python variable belonging to only one object.
omar.company = "Google"
print(omar.company) # Google
```

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class

```
class employee1:
   company = "TCS"
   # def getSalary(self):
   #     print("Salary is 100k")
   def getSalary(self, signature):
      print(f"Salary of employee working in {self.company} is
{self.salary}\n{signature}")

madi1 = employee1()
# employee1.getSalary(madi1)
# madi1.getSalary()
madi1.salary = 1000000
madi1.getSalary("That's Ok")
# you can access the properties of that particular method

# Ex2.
class Cat:
   def __init__(self, name, age):
      self.name = name
      self.age = age

   def info(self):
      print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

   def make_sound(self):
      print("Meow")

cat1 = Cat('Andy', 2)
cat1.info()
cat2 = Cat('Phoebe', 3)

# The self keyword is used to represent an instance (object) of the given class. In this case, the two Cat objects cat1 and cat2 have their own name and age
```

attributes. If there was no self argument, the same class couldn't hold the information for both these objects.
However, since the class is just a blueprint, self allows access to the attributes and methods of each object in python. This allows each object to have its own attributes and methods. Thus, even long before creating these objects, we reference the objects as self while defining the class.

Static Method:

Static methods in Python are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class. This means that a static method can be called without an object for that class.
Ex.

```
class statMet:
  @staticmethod
  def test(status):
    print('static method',status)

# calling a static method without object
statMet.test("Done")
# calling using object
anm = statMet()
anm.test("Done")
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
To understand the meaning of classes we have to understand the built-in __init__() function.
All classes have a function called __init__(), which is always executed when the class is being initiated.
Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Ex.

```
# __init__
# Automatically called when the object is created
# Also known as constructor
```

```
class person:
    def __init__(self,name):
        print(f"I am created from {name}")

    @staticmethod
    def timeNow():
        print("The time is 9am")

madi3 = person("madi3")
person.timeNow()
# The __init__() function is called automatically every time the class is being used to create a new object.

Ex.
class employee3:
    company = "Google"
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary



    def getEmployeeDet(self):
        print(f"The employee working in {self.company} is {self.name} which is {self.age} years old who has salary of {self.salary}")

    @staticmethod
    def legalTerm(howMany):
        print(f"{howMany} the employee has accepted all the terms and policy")

# print(employee3.company)
employee3.legalTerm("All")
madi4 = employee3("Omar", 19 , 9876543)
print(madi4.name) # Omar
madi4.getEmployeeDet()
```

Delete Object Properties and Object

You can delete properties on objects by using the del keyword:

```
# delete properties
class forDel:
   todel="I will be going soon"
   @staticmethod
   def howTo():
      print("Use del operator and then write the property name you want to
delete")

forDel.howTo()
print(forDel.todel)
del forDel.todel
# print(forDel.todel) # error of no property found

# delete Object
print(forDel) #<class '__main__.forDel'>
del forDel
print(forDel) # not defined
```

The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Ex.

```
# The pass statement
class willUse:
   pass
```

All together:

```
# Noun      ->  Class      -> Employee
# Adjective ->  Attributes -> name, age, salary
# Verbs     ->  Methods    -> getSalary(), getAge()

# Create a class
class perfect:
   name="Madi"
# Create an object
p = perfect()
print(p.name)

# Ex2
class flightForm:
   formType="flightForm"
   def printData(self):
      print(f"Name of the person is:{self.name}")
      print(f"Flight of the person is:{self.flight}")

madiApplication = flightForm()
madiApplication.name="Madi"
madiApplication.flight="AirIndia"
madiApplication.printData()


# Class Atrributes
# An attributes that belongs to a class rather than object
class clsAttri:
   company = "Wipro"
madi2 = clsAttri()
print(madi2.company)
clsAttri.company = "Amazon"
print(madi2.company)

# Instance
#  An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle. Instantiation − The creation of an instance of a class. Method − A special kind of function that is defined in a class definition.
```

```
class employee:
    company = "Infosys"
    salary = 987556

omar = employee()
neuro = employee()
print(omar.company) # Infosys
print(neuro.company) # Infosys

# Create instance attribute
# An instance attribute is a Python variable belonging to only one object.
omar.company = "Google"
print(omar.company) # Google

# Self Parameter:
class employee1:
    company = "TCS"
    # def getSalary(self):
    #     print("Salary is 100k")
    def getSalary(self, signature):
        print(f"Salary of employee working in {self.company} is
{self.salary}\n{signature}")

madi1 = employee1()
# employee1.getSalary(madi1)
# madi1.getSalary()
madi1.salary = 1000000
madi1.getSalary("That's Ok")
# you can access the properties of that particular method

# Ex2.
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")
```

```
    def make_sound(self):
        print("Meow")

cat1 = Cat('Andy', 2)
cat1.info()
cat2 = Cat('Phoebe', 3)

# The self keyword is used to represent an instance (object) of the given class. In this case, the two Cat objects cat1 and cat2 have their own name and age attributes. If there was no self argument, the same class couldn't hold the information for both these objects.
# However, since the class is just a blueprint, self allows access to the attributes and methods of each object in python. This allows each object to have its own attributes and methods. Thus, even long before creating these objects, we reference the objects as self while defining the class.

# Static Method:
# Static methods in Python are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class. This means that a static method can be called without an object for that class.
class statMet:
    @staticmethod
    def test(status):
        print('static method',status)

# calling a static method without object
statMet.test("Done")
# calling using object
anm = statMet()
anm.test("Done")

# __init__
# Automatically called when the object is created
# Also known as constructor
class person:
    def __init__(self,name):
        print(f"I am created from {name}")
```

```
    @staticmethod
    def timeNow():
        print("The time is 9am")

madi3 = person("madi3")
person.timeNow()
# The __init__() function is called automatically every time the class is being used to create a new object.

class employee3:
    company = "Google"
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary



    def getEmployeeDet(self):
        print(f"The employee working in {self.company} is {self.name} which is {self.age} years old who has salary of {self.salary}")

    @staticmethod
    def legalTerm(howMany):
        print(f"{howMany} the employee has accepted all the terms and policy")

# print(employee3.company)
employee3.legalTerm("All")
madi4 = employee3("Omar", 19 , 9876543)
print(madi4.name) # Omar
madi4.getEmployeeDet()

# delete properties
class forDel:
    todel="I will be going soon"
    @staticmethod
    def howTo():
        print("Use del operator and then write the property name you want to delete")
```

```
forDel.howTo()
print(forDel.todel)
del forDel.todel
# print(forDel.todel) # error of no property found

# delete Object
print(forDel) #<class '__main__.forDel'>
del forDel
print(forDel) # not defined

# The pass statement
class willUse:
    pass
```