09-OOP

September 16, 2022

1 Object oriented programming

1.1 Basics

- **Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of **attributes**, and code, in the form of functions known as **methods**.
- Object's method can access and often modify the data attributes of the object with which they are associated (objects have a notion of "self").
- In OO programming, computer programs are designed by making them out of objects that interact with one another.
- **Classes are used to create objects (objects are instances of the classes with which they were created), so we could see them as instance factories**.

[1]:
```
class Simplest:        # when empty, the braces are optional
    pass
```

we can create an instance of `Simplest: simp`

[2]:
```
simp = Simplest()

type(simp)   # what type is simp?
```

[2]:
```
__main__.Simplest
```

Is `simp` an instance of `Simplest`?

[3]:
```
print(type(simp) == Simplest)
```

```
True
```

There's a better way for this

[4]:
```
isinstance(simp, Simplest)
```

[4]:
```
True
```

what does `Simplest` "contain"?

[5]:
```
dir(Simplest)
```

[5]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

- After the **class object** has been created it **basically represents a namespace**.
- We can call that class to create its instances.
- Each instance **inherits the class attributes and methods** and **is given its own namespace**.
- We already know that, to walk a namespace, all we need to do is to use the dot (`.`) operator.
- **Class attributes** are shared amongst all instances, while **instance attributes** are not, they belong to the object;
- You should use **class attributes** to provide the states and behaviors to be shared by all instances, and use **instance attributes** for data that belongs just to one specific object.
- From within a class method we can refer to an instance by means of a special argument, called `self` *by convention*.
- `self` is always the first attribute of an instance method.
- While initializing an instance we have to assign values to the attributes. Other languages use a constructor but in Python we use an **initializer**, since it works on an already created instance, and therefore it's called `__init__`
- `__init__` is a "magic" method, which is run right after the object is created.

- Python classes also have a `__new__` method, which is the actual **constructor**.

[6]:
```
class Rectangle:
    def __init__(self, sideA, sideB):
        self.sideA = sideA
        self.sideB = sideB

    def area(self):
        return self.sideA * self.sideB
```

[7]:
```
r1 = Rectangle(10, 4)
```

[8]:
```
r1.sideA
```

[8]:
```
10
```

[9]:
```
f'Sides: {r1.sideA}, {r1.sideB}'
```

[9]:
```
'Sides: 10, 4'
```

[10]:
```
f'r1 area: {r1.area()}'
```

[10]:
```
'r1 area: 40'
```

[11]:
```
r2 = Rectangle(7, 3)
print('r2 area:', r2.area())
```

```
r2 area: 21
```

- Class attributes are shared amongst all instances, while instance attributes are not;
- You should use class attributes to provide the states and behaviors to be shared by all instances, and use instance attributes for data that belongs just to one specific object.

[12]:
```
class Square:

    numero_de_quadrados = 0          # numero_de_quadrados is a class attribute

    def __init__(self, side = 8):
        self.side = side              # self.side is an instance attribute
        Square.numero_de_quadrados += 1
        # self.__class__.numero_de_quadrados += 1   # or even better, since␣
↪like this you can change the name of the class

    def area(self): # self is a reference to an instance
        return self.side ** 2
```

[13]:
```
sq = Square()
print(sq.area()) # 64 (side is found on the class)
print(Square.area(sq)) # 64 (equivalent to sq.area())
```

```
64
64
```

[14]:
```python
sq.side = 10
sq.area()                  # 100 (side is found on the instance)
```

[14]: 100

How many Squares were created ?

[15]:
```python
Square.numero_de_quadrados
```

[15]: 1

1.2 Dynamic attributes (optional)

[16]:
```python
class Person():
    species = 'Human'                        # class attributes

print('species? '+ Person.species)           # Human

Person.alive = True                          # Added dynamically!
print('alive? ' + str(Person.alive))         # True

man = Person()
print('species? '+ man.species)              # Human (inherited)
print('alive? ' + str(man.alive))            # True (inherited)

Person.alive = False
print('alive? ' + str(man.alive))            # False (inherited)
```

```
species? Human
alive? True
species? Human
alive? True
alive? False
```

Be aware of the attributes…

[17]:
```python
class Point():
    x, y = 10, 7
```

[18]:
```python
p = Point()

print('coordinates:', p.x, p.y) # 10 7 (from class attribute)

p.x = 12         # p gets its own 'x' attribute
print('p.x:', p.x)      # 12 (now found on the instance)
print('Point.x:', Point.x)  # 10 (class attribute still the same)
```

```
coordinates: 10 7
p.x: 12
Point.x: 10
```

Now, we have 2 "x"'s

[19]:
```python
id(Point.x)          # atributo da classe
```

[19]: 1882419325520

[20]:
```python
id(p.x)              # atributo da instância
```

[20]: 1882419325584

[21]:
```python
del p.x          # we delete instance attribute
print(p.x)       # 10 (now search has to go again to find class attr)
```

```
10
```

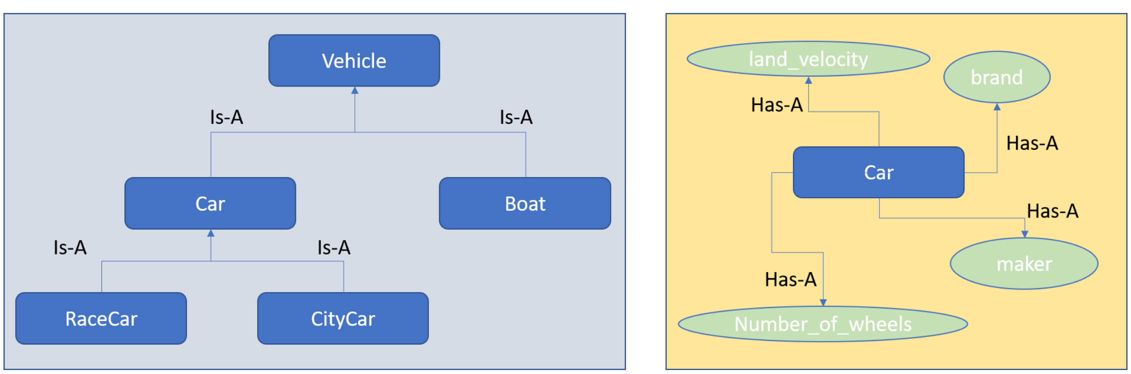
[22]:
```python
p.z = 3          # let's make it a 3D point
print('p.z:', p.z)      # 3
```

```
p.z: 3
```

[23]:
```python
print(Point.z)  # AttributeError: type object 'Point' has no att. 'z'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [23], line 1
----> 1 print(Point.z)

AttributeError: type object 'Point' has no attribute 'z'
```

1.3 Inheritance and composition



1.3.1 Inheritance

- **Inheritance** means that two objects are related by means of an *Is-A* type of relationship.

[ ]:
```
class Vehicle:
    def __init__(self, brand, model, number_of_passengers=0, owner=None):
        self.owner = owner
        self.brand = brand
        self.model = model
        self.number_of_passengers = number_of_passengers

    def vehicle_info(self):
        return f'''Vehicle of brand {self.brand}, model {self.model}, with␣
↪capacity for {self.number_of_passengers} passengers.\n The owner is {self.
↪owner}.'''
```

Now, a land vehicle is a vehiclecle, so it should have all the vehicle properties (even if they have to be redefined) and some other

[ ]:
```
class  LandVehicle(Vehicle):

    def __init__(self, land_velocity, wheels, number_of_wheels, brand, model,␣
↪number_of_passengers=0, owner=None):

        # call Vehicle initializer sending the Vehicle's attributes
        super().__init__(owner=owner, brand=brand, model=model,␣
↪number_of_passengers=number_of_passengers);

        self.land_velocity = land_velocity;
        self.wheels = wheels;
        self.number_of_wheels = number_of_wheels;

    def vehicle_info(self): # In some case, methods need to be redefined
        return  super().vehicle_info() + f''' \n It has {self.number_of_wheels}␣
↪wheels with the specifications {self.wheels}. The land velocity is {self.
↪land_velocity}.'''
```

[ ]:
```
lv = LandVehicle(land_velocity=200,
                 wheels='225/55 R 17 97 W',
                 number_of_wheels=4,
                 owner='Margarida',
                 brand='Fiat',
                 model='500',
                 number_of_passengers=4)

print(lv.vehicle_info())
```

[ ]:
```
class Car(LandVehicle):

    def __init__(self, engine, number_of_doors, land_velocity, wheels,␣
↪number_of_wheels, brand, model, number_of_passengers=0, owner=None):
```

```python
        # call LandVehicle contructor
        super().__init__(land_velocity=land_velocity, wheels=wheels,␣
↪number_of_wheels=number_of_wheels, owner=owner, brand=brand, model=model,␣
↪number_of_passengers=number_of_passengers)

        self.engine = engine
        self.number_of_doors = number_of_doors
        self.kms = 0
        self.filled_fuel = 0

    def vehicle_info(self): # redefinição do método
        return  super().vehicle_info() + f''' Also has an engine with {self.
↪engine}cc and {self.number_of_doors} doors.'''

    def add_kms(self, kms):
        self.kms += kms

    def add_filled_fuel(self, filled_fuel):
        self.filled_fuel += filled_fuel

    def consumption(self):
        return self.filled_fuel / self.kms * 100
```

[ ]:
```python
c = Car(
    engine='1500',
    number_of_doors=5,
    land_velocity=200,
    wheels='225/55 R 17 97 W',
    number_of_wheels=4,
    owner='Margarida',
    brand='Fiat',
    model='500',
    number_of_passengers=4
)

print(c.vehicle_info())
```

[ ]:
```python
c.add_kms(1823)
c.add_filled_fuel(100)

c.consumption()
```

1.3.2 More about super() (optional)

The attributes of the super class can be called in a distinct number of ways

[ ]:
```python
class Book:
    def __init__(self, title, publisher, pages):
        print('Book')
        self.title = title
        self.publisher = publisher
        self.pages = pages
```

By calling them directly (not advisable)

[ ]:
```python
class Ebook1(Book): # is a Book
    def __init__(self, title, publisher, pages, format_):
        self.title = title          # not advisable
        self.publisher = publisher  # not advisable
        self.pages = pages          # not advisable
        self.format_ = format_
```

Using the super's class name, which is better but can lead to problems if the class changes its name

[ ]:
```python
class Ebook2(Book): # is a Book
    def __init__(self, title, publisher, pages, format_):
        # If we modify the logic within the __init__ method of Book,
        # we don't need to touch Book, it will auto adapt to the change.
        Book.__init__(self, title, publisher, pages)
        # But if we change the name of the Book class...
        self.format_ = format_
```

or using `super()`, as already seen.

[ ]:
```python
class Ebook3(Book): # is a Book
    def __init__(self, title, publisher, pages, format_):
        # now we can change the name of the class
        super().__init__(title, publisher, pages)

        # Another way to do the same thing is:
        # super(Book, self).__init__(title, publisher, pages)

        self.format_ = format_
```

1.3.3 Composition

- On the other hand, **composition** means that two objects are related by means of a *Has-A* type of relationship.

[ ]:
```python
class Engine():
    def start(self):
        print(f'Engine {self.__class__.__name__} started.')

    def stop(self):
        print(f'Engine {self.__class__.__name__} stopped.')
```

[ ]:
```python
class ElectricEngine(Engine):        # Is-A Engine
    pass

class V8Engine(Engine):              # Is-A Engine
    pass
```

So, a `Car` **Has-A** `Engine`

[24]:
```python
class Car:
    def __init__(self, engine):
        self.engine = engine      # Has-A Engine

    def start(self):
        print('Start engine {0} for car {1}... Wroom!'.format(self.engine.
↪__class__.__name__, self.__class__.__name__))
        self.engine.start()

    def stop(self):
        self.engine.stop()
```

[25]:
```python
e = Engine()
normal = Car(e)
normal.start()
normal.stop()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [25], line 1
----> 1 e = Engine()
      2 normal = Car(e)
      3 normal.start()

NameError: name 'Engine' is not defined
```

[ ]:
```python
sport = Car(V8Engine())
sport.start()
sport.stop()
```

Other ways to implement an engine

[26]:
```python
class Car():
    engine_cls = Engine
    def __init__(self):
        self.engine = self.engine_cls()      # Has-A Engine

    def start(self):
```

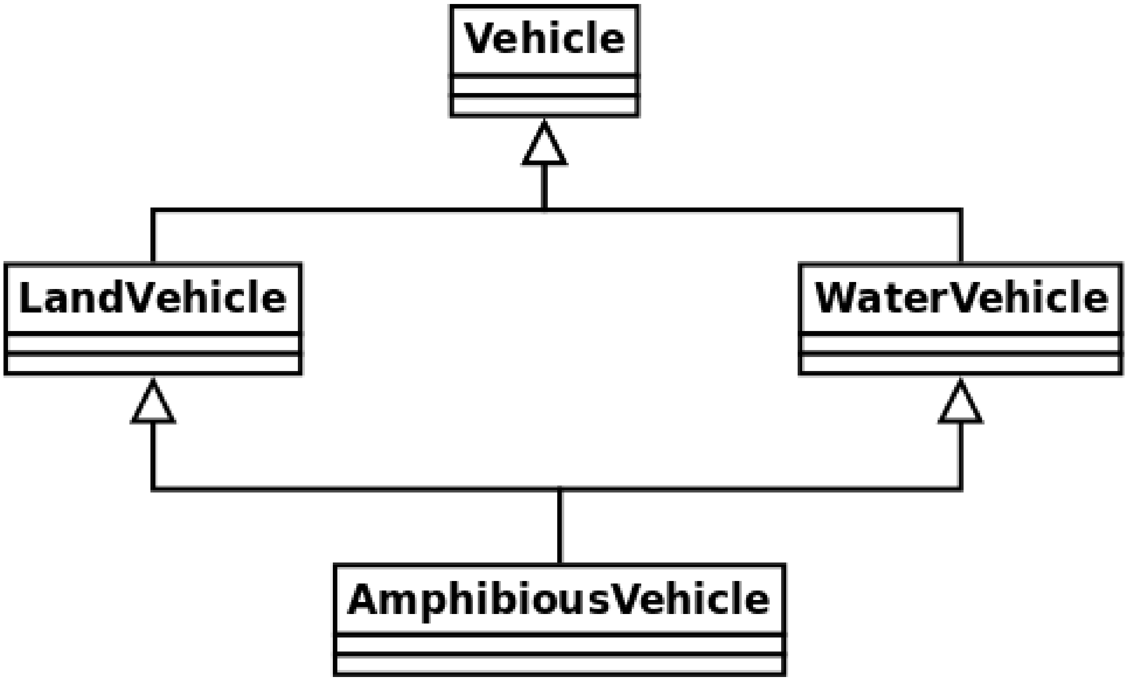
```python
        print(f'Start engine {self.engine.__class__.__name__} for car {self.
↪__class__.__name__}... Wroom!')
        self.engine.start()

    def stop(self):
        self.engine.stop()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [26], line 1
----> 1 class Car():
      2     engine_cls = Engine
      3     def __init__(self):

Cell In [26], line 2, in Car()
      1 class Car():
----> 2     engine_cls = Engine
      3     def __init__(self):
      4         self.engine = self.engine_cls()     # Has-A Engine

NameError: name 'Engine' is not defined
```

[ ]:
```python
class RaceCar(Car):                   # Is-A Car
    engine_cls = V8Engine             # Has-A Engine

class CityCar(Car):                   # Is-A Car
    engine_cls = ElectricEngine       # Has-A Engine

class F1Car(RaceCar):                 # Is-A RaceCar and also Is-A Car
    pass                              # engine_cls = V8Engine
```

[ ]:
```python
cars = [Car(), RaceCar(), CityCar(), F1Car()]
for car in cars:
    car.start()

for car in cars:
    car.stop()
```

1.4 Multiple Inheritance



[27]:
```
class Vehicle:
    def __init__(self, owner, brand):
        self.owner = owner
        self.brand = brand

    def vehicle_info(self):
        raise NotImplementedError("vehicle_info: não implementado")
```

[28]:
```
class LandVehicle(Vehicle):
    def __init__(self, owner, brand, land_velocity):
        print(super(LandVehicle, self))
        super().__init__(self, owner, brand)
        self.land_velocity = land_velocity

    @property
    def land_velocity(self):
        return self.__land_velocity

    @land_velocity.setter
    def land_velocity(self, lv):
        self.__land_velocity = lv
```

[29]:
```
class WaterVehicle(Vehicle):
    def __init__(self, owner, brand, water_velocity):
        print(super(WaterVehicle, self))
```

```
        Vehicle.__init__(self, owner, brand)
        self.water_velocity = water_velocity

    @property
    def water_velocity(self):
        return self.__water_velocity

    @water_velocity.setter
    def water_velocity(self, wv):
        self.__water_velocity = wv
```

[30]:

```
class AmphibiousVehicle(LandVehicle, WaterVehicle):
    def __init__(self, owner, brand, land_velocity, water_velocity):
        LandVehicle.__init__(self, owner, brand, land_velocity)
        WaterVehicle.__init__(self, owner, brand, water_velocity)

    def print_info(self):
        print(f'''This is an AmphibiousVehicle owned by {self.owner}
        from {self.brand} with velocity {self.land_velocity} km/h in land and␣
 ↪{self.water_velocity} Knot  in the water''')
```

[31]:

```
a = AmphibiousVehicle('Margarida', 'rinspeed splash', 199, 38)
a.print_info()
```

```
<super: <class 'LandVehicle'>, <AmphibiousVehicle object>>
<super: <class 'WaterVehicle'>, <AmphibiousVehicle object>>
<super: <class 'WaterVehicle'>, <AmphibiousVehicle object>>
This is an AmphibiousVehicle owned by Margarida
        from rinspeed splash with velocity 199 km/h in land and 38 Knot  in the
water
```

The methods resolution order is

[32]:

```
AmphibiousVehicle.__mro__
```

[32]:

```
(__main__.AmphibiousVehicle,
 __main__.LandVehicle,
 __main__.WaterVehicle,
 __main__.Vehicle,
 object)
```

1.4.1 Another multiple inheritance example (optional)

[33]:

```
class Shape(object):
    geometric_type = 'Generic Shape'
    def area(self):     # This acts as placeholder for the interface
        raise NotImplementedError
```

```python
    def get_geometric_type(self):
        return self.geometric_type

    def f(self):
        print("Shape")

class Plotter:
    def plot(self, ratio, topleft):
        # Imagine some nice plotting logic here...
        print('Plotting at {}, ratio {}.'.format( topleft, ratio))

    def f(self):
        print("Plotter")

class Polygon(Shape, Plotter):          # base class for polygons
    geometric_type = 'Polygon'

    f = Plotter.f

class RegularPolygon(Polygon):          # Is-A Polygon
    geometric_type = 'Regular Polygon'
    def __init__(self, side):
        self.side = side
```

[34]:

```python
p = Polygon()
p.plot(0, (0,0))
```

```
Plotting at (0, 0), ratio 0.
```

[35]:

```python
p.get_geometric_type()
```

[35]: 'Polygon'

[36]:

```python
p.f()
```

```
Plotter
```

[37]:

```python
try:
    p.cor()
except AttributeError as e:
    print("it hasn't that method")
    print(e)
```

```
it hasn't that method
'Polygon' object has no attribute 'cor'
```

[38]:

```python
class RegularHexagon(RegularPolygon):   # Is-A RegularPolygon
    geometric_type = 'RegularHexagon'
```

```python
    def area(self):
        return 1.5 * (3 ** .5 * self.side ** 2)

class Square(RegularPolygon):            # Is-A RegularPolygon
    geometric_type = 'Square'
    def area(self):
        return self.side * self.side
```

[39]:
```python
hexagon = RegularHexagon(10)
print(hexagon.area())                # 259.8076211353316
print(hexagon.get_geometric_type()) # RegularHexagon
hexagon.plot(0.8, (75, 77))
```

```
259.8076211353316
RegularHexagon
Plotting at (75, 77), ratio 0.8.
```

[40]:
```python
square = Square(12)
print(square.area())                 # 144
print(square.get_geometric_type())  # Square
square.plot(0.93, (74, 75))          # Plotting at (74, 75), ratio 0.93.
```

```
144
Square
Plotting at (74, 75), ratio 0.93.
```

1.5 Static methods (optional)

- When you create a class object, Python assigns a name to it. That name acts as a namespace, and sometimes it makes sense to group functionalities under it.
- Static methods are perfect for this use case since unlike instance methods, they are not passed any special argument.
- Static methods are created by applying the `@staticmethod` decorator to them.
- The class, acts as a container for functions.
- Another approach would be to have a separate module with functions inside.

[41]:
```python
class String:

    @staticmethod           # decorator
    def is_palindrome(s, case_insensitive=True):
        s = ''.join(c for c in s if c.isalnum()) # Study this!
        if case_insensitive:
            s = s.lower()
        for c in range(len(s) // 2):
            if s[c] != s[-c - 1]:
                return False
```

```python
        return True

    @staticmethod
    def get_unique_words(sentence):
        return set(sentence.split())
```

[42]:
```python
print(String.is_palindrome('Radar', case_insensitive=False)) # False
print(String.is_palindrome('A nut for a jar of tuna'))       # True
print(String.is_palindrome('Never Odd, Or Even!'))           # True

print(String.get_unique_words('I love palindromes. I really really love them!
 ↪'))   # {'them!', 'really', 'palindromes.', 'I', 'love'}
```

```
False
True
True
{'really', 'them!', 'love', 'palindromes.', 'I'}
```

1.6 Class methods (optional)

- Class methods are slightly different from instance methods in that they also take a special first argument, but in this case, it is the class object itself.
- Two very common use cases for coding class methods are to provide
  - factory capability to a class
  - allow breaking up static methods (which you have to then call using the class name) without having to hardcode the class name in your logic.

[43]:
```python
class Pizza:

    # area of pizza per person
    area_by_person = 750.

    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarela', 'tomate'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarela', 'tomate', 'fiambre'])

    @staticmethod
```

```python
    def how_many_person(radius):
        area_pizza = 3.14 * radius ** 2
        return area_pizza / Pizza.area_by_person

    @staticmethod
    def which_radius(number_of_persons):
        area_total = number_of_persons * Pizza.area_by_person
        return (area_total / 3.14) ** .5
```

[44]:
```python
four_cheeses = Pizza(['mozzarela', 'gorgonzola', 'requeijão', 'parmesão'])
four_cheeses
```

[44]: `Pizza(['mozzarela', 'gorgonzola', 'requeijão', 'parmesão'])`

[45]:
```python
margherita = Pizza.margherita()
margherita
```

[45]: `Pizza(['mozzarela', 'tomate'])`

[46]:
```python
r = 30
f'a pizza with {r}cm² is enough for {Pizza.how_many_person(r)} person'
```

[46]: `'a pizza with 30cm² is enough for 3.768 person'`

[47]:
```python
p = 4
f'for {p} person you shoud order a pizza with {Pizza.which_radius(p)} cm radius'
```

[47]: `'for 4 person you shoud order a pizza with 30.909772123696634 cm radius'`

1.7 Private methods and name mangling

- In OOP, **public** attributes are accessible from any point in the code, while **private** ones are accessible only within the scope they are defined in.
- In Python, there is no such thing: ** everything is public**
- Programmers rely on
  - Convention
    - If an attribute's name has no leading underscores it is considered **public**. This means you can access it and modify it freely.
    - When the name has one leading underscore, the attribute is considered **private**, (probably used internally and you should not use it or modify it from the outside)
  - mangling
    - Any attribute name that has at least two leading underscores and at most one trailing underscore, like `__my_attr`, is replaced with a name that includes an underscore and the class name before the actual name, like `_ClassName__my_attr`

1.7.1 The _ (underscore) convention

[48]:
```
class A:
    def __init__(self, factor):
        self._factor = factor

    def op1(self):
        print('Op1 with factor {}...'.format(self._factor))

class B(A):     # derived from A
    def op2(self, factor):
        self._factor = factor # you can do this but you probably shouldn't␣
 ↪(_factor is private in the mother class)
        print('Op2 with factor {}...'.format(self._factor))

obj = B(100)
obj.op1()
obj.op2(42)
obj.op1()
```

```
Op1 with factor 100...
Op2 with factor 42...
Op1 with factor 42...
```

[49]:
```
obj._factor = 1290     # definetly , you shouldn't do this. _factor should be␣
 ↪treated as private!
obj.op1()
```

```
Op1 with factor 1290...
```

[50]:
```
dir(obj)
```

[50]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
```

```
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_factor',
 'op1',
 'op2']
```

1.7.2 Mangling

[51]:
```python
class A:
    def __init__(self, factor):
        self.__factor = factor    # a double underscore

    def op1(self):
        print('Op1 with factor {}...'.format(self.__factor))

class B(A):
    def op2(self, factor):
        self.__factor = factor
        print('Op2 with factor {}...'.format(self.__factor))
```

[52]:
```python
obj = B(100)
obj.op1()       # Op1 with factor 100...
obj.op2(42)     # Op2 with factor 42...
obj.op1()       # Op1 with factor 100... <- Now you did not change the __factor␣
 ↪of class A
```

```
Op1 with factor 100…
Op2 with factor 42…
Op1 with factor 100…
```

[53]:
```python
dir(obj) # look for ['_A__factor', '_B__factor', ... , 'op1', 'op2']
```

[53]:
```
['_A__factor',
 '_B__factor',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
```

```
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'op1',
 'op2']
```

[54]:
```
print(obj._A__factor)   # 100
print(obj._B__factor)   # 42
```

```
100
42
```

[55]:
```
try:
    print(obj.__factor)     # AttributeError: 'B' has no attr '__factor'
except:
    print("there is no obj.__factor")
```

```
there is no obj.__factor
```

1.7.3 property decorator

Imagine that you have an age attribute in a Person class and at some point you want to make sure that when you change its value, you're also checking that age is within a proper range, like [18, 99].

You can write accessor methods, like `get_age()`and `set_age()` (also called getters and setters) and put the logic there.

Getters and setters are used in many object oriented programming languages to ensure the principle of data **encapsulation**. They are known as mutator methods as well. According to this principle, the attributes of a class are made private to hide and protect them from other code.

But the Pythonic way to introduce attributes is to make them public.

getters and setters In short, you can use getters and setters, like this

[56]:
```
class PersonNonPythonic:
    def __init__(self, age):
        self.set_age(age)

    def get_age(self):
        return self._age

    def set_age(self, age):
        assert 18 <= age <= 99, 'Age must be within [18, 99]'
        self._age = age

p = PersonNonPythonic(20)
p.set_age(21)
```

[57]:
```
p.set_age(11)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [57], line 1
----> 1 p.set_age(11)

Cell In [56], line 9, in PersonNonPythonic.set_age(self, age)
      8 def set_age(self, age):
----> 9     assert 18 <= age <= 99, 'Age must be within [18, 99]'
     10     self._age = age

AssertionError: Age must be within [18, 99]
```

"properties" you can also use "properties" as follows

[58]:
```
class PersonPythonic:
    def __init__(self, age):
        self.age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        assert 18 <= age <= 99, 'Age must be within [18, 99]'
        self.__age = age
```

[59]:
```python
person = PersonPythonic(39)

person.age
```

[59]: 39

[60]:
```python
person.age = 18      # Notice we access as data attribute
person.age
```

[60]: 18

[61]:
```python
try:
    person.age = 100    # ValueError: Age must be within [18, 99]
except AssertionError as e:
    print(e)
```

```
Age must be within [18, 99]
```

Returning to the car example

[62]:
```python
class Car:

    def __init__(self, color, brand):
        self.color = color  # calls the property
        self.brand = brand # chama a propriedade (valida dados).E guarda o␣
 ↪valor em self.__marca

    @property
    def color(self):
        return self.__color

    @color.setter
    def color(self, color):
        print('debug: setting a color')
        if color.lower() in ['red', 'white', 'yellow']:
            self.__color = color
        else:
            raise BaseException('invalid color')

    @color.deleter
    def color(self):
        print('debug: setting color to none')
        self.__color = None

    @property
    def brand(self):
        return self.__brand
```

```
    @brand.setter
    def brand(self, brand):
        print('debug: setting brand')
        if brand.lower() in ['audi', 'fiat', 'seat', 'ferrari']:
            self.__brand = brand
        else:
            raise
```

[63]:
```
c = Car('red', 'fiat')
```

```
debug: setting a color
debug: setting brand
```

[64]:
```
c.color='white'
```

```
debug: setting a color
```

[65]:
```
try:
    c.color = 'azul'
except:
    print("you are smart!")
```

```
debug: setting a color
you are smart!
```

1.7.4 Operator overloading

To overload an operator means to give it a meaning according to the context in which it is used. For example, the + operator means addition when we deal with numbers, but concatenation when we deal with sequences.

[66]:
```
class OverloadingExamples:
    def __init__(self, s):
        self._s = s

    def __len__(self):
        return len(self._s.replace(' ', '')) # strip all spaces

    def __bool__(self):
        return 'year' in self._s

    def __add__(self, other):
        return OverloadingExamples(self._s + other._s)

    def __repr__(self):
        return "--" + self._s + "--"

    def __str__(self):
        return "++" + self._s + "++"
```

```python
    def __eq__(self, other):
        return self._s == other._s
```

The following will call `__repr__`

[67]:
```python
obj = OverloadingExamples('Hello! My dog is called Olivia and she is 3 months␣
 ↪old!')
obj  # this will ask for the object's representation (__repr__)
```

[67]:
```
--Hello! My dog is called Olivia and she is 3 months old!--
```

The following will call `__str__` (it ask for a string representation of `obj`)

[68]:
```python
print(obj)
```

```
++Hello! My dog is called Olivia and she is 3 months old!++
```

[69]:
```python
str(obj)
```

[69]:
```
'++Hello! My dog is called Olivia and she is 3 months old!++'
```

[70]:
```python
obj.__len__()
```

[70]:
```
44
```

The following will call `__len__`

[71]:
```python
len(obj)
```

[71]:
```
44
```

The following will call `__bool__`

[72]:
```python
bool(obj)
```

[72]:
```
False
```

[73]:
```python
obj2 = OverloadingExamples('Hello! I am 42 years old!')
len(obj2)
```

[73]:
```
20
```

[74]:
```python
bool(obj2)
```

[74]:
```
True
```

The following comparison (`==`) will call `__eq__`

[75]:
```python
obj3 = OverloadingExamples('Hello! I am 42 years old!')
obj4 = OverloadingExamples('Hello! I am 43 years old!')
```

```
obj3 == obj4 # obj3.__eq__(obj4)
```

[75]: False

The sum ('+') will call `__add__`

[76]:
```
obj = obj3 + obj4
# obj3.__add__(obj4)
obj
```

[76]: --Hello! I am 42 years old!Hello! I am 43 years old!--

2 Exercises

Go here...