Object Oriented Programming

CONTENTS

- Class statement, class attributes, class namespace.
- Object, object attributes, object namespace.
- Writing constructor in Python. The __init__ method.
- Operator overloading.
    - Basic operators: Overloading usual operators defining special methods viz. __add__, __sub__, __mul__, __matmul__, __div__, __truediv__, __floordiv__, __mod__, __divmod__, __pow__, __lshift__, __rshihft__, __and__, __or__, __xor__.
    - Preparing user defined object for built-in methods: Defining special methods viz. __str__, __repr__ for printing object, __len__ for compatibility with len.
    - __getitem__, __setitem__ for subscripting object on left and right hand side.
    - __call__ to make a callable object.
- Inheritance:
    - Implementation and Interface inheritance in Python.
    - @staticmethod, @classmethod decorators for marking object and class methods.
    - Multiple inheritance, diamond problem, and MRO-method resolution order in Python.

Object-oriented Programming, or *OOP* for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modelling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

The key takeaway is that objects are at the centre of the object-oriented programming paradigm, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

Focusing first on the data, each thing or object is an instance of some *class*. The primitive data structures available in Python, like numbers, strings, and lists are designed to represent simple things like the cost of something, the name of a poem, and your favorite colors, respectively.

What if you wanted to represent something much more complicated?

For example, let's say you wanted to track a number of different animals. If you used a list, the first element could be the animal's name while the second element could represent its age.

How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals? This lacks organization, and it's the exact need for *classes*.

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age.

It's important to note that a class just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is.

It may help to think of a class as an *idea* for how something should be defined.

While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class. It's not an idea anymore; it's an actual animal, like a dog named Roger who's eight years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you. You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

```
#Defining a class is simple in Python:

class Car:
    pass
```

Instance Attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the __init__() method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the self variable, which refers to the object itself .

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
```

the self variable is also an instance of the class. Since instances of a class have varying values we could state Car.make = make rather than self.make = make. But since not all cars share the same make, we need to be able to assign different values to different instances. Hence the need for the special self variable, which will help to keep track of individual instances of each class.

Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances

```
class Dog:
    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Instantiating Objects

```
>>> class Dog:
...     pass
...
>>> Dog()
<__main__.Dog object at 0x1004ccc50>
>>> Dog()
<__main__.Dog object at 0x1004ccc90>
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

Instance, Class, and Static Methods

Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the `__init__` method, the first argument is always `self`:

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

- Instance methods need a class instance and can access the instance through `self`.
- Class methods don't need a class instance. They can't access the instance (`self`) but they have access to the class itself via `cls`.
- Static methods don't have access to `cls` or `self`. They work like regular functions but belong to the class's namespace.
- Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have maintenance benefits.

Getters and setters

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. Data encapsulation is seen as the bundling of data with the methods that operate on these data. These methods are of course the getter for retrieving the data and the setter for changing the data. According to this principle, the attributes of a class are made private to hide and protect them from other code.

Unfortunately, it is widespread belief that a proper Python class should encapsulate private attributes by using getters and setters. As soon as one of these programmers introduces a new attribute, he or she will make it a private variable and creates "automatically" a getter and a setter for this attributes. Such programmers may even use an editor or an IDE, which automatically create getters and setters for all private attributes. These tools even warn the programmer if she or he uses a public attribute! Java programmers will wrinkle their brows, screw up their noses, or even scream with horror when they read the following: The Pythonic way to introduce attributes is to make them public.

```
class P:
    def __init__(self,x):
        self.__x = x
    def get_x(self):
        return self.__x
```

```
    def set_x(self, x):
        self.__x = x
```

We can see in the following demo session how to work with this class and the methods:

```
>>> p1 = P(42)
>>> p2 = P(4711)
>>> p1.get_x()
42
>>> p1.set_x(47)
>>> p1.set_x(p1.get_x()+p2.get_x())
>>> p1.get_x()
4758
>>>
```

But what happens if we want to change the implementation in the future. Let's assume we want to change the implementation like this: The attribute x can have values between 0 and 1000. If a value larger than 1000 is assigned, x should be set to 1000. Correspondingly, x should be set to 0, if the value is less than 0.

It is easy to change our first P class to cover this problem. We change the set_x method accordingly:

```
class P:


    def __init__(self,x):
        self.set_x(x)


    def get_x(self):
        return self.__x


    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

The following Python session shows that it works the way we want it to work:

```
>>> from mutators import P
>>> p1 = P(1001)
>>> p1.get_x()
1000
>>> p2 = P(15)
>>> p2.get_x()
```

```
15
>>> p3 = P(-1)
>>> p3.get_x()
0
```

But there is a catch: Let's assume we have designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

```
p1 = P(42)
p1.x = 1001
```

Our new class means breaking the interface. The attribute x is not available anymore. That's why in Java e.g. people are recommended to use only private attributes with getters and setters, so that they can change the implementation without having to change the interface.

But Python offers a solution to this problem. The solution is called properties!

The class with a property looks like this:

```
class P:


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


    @property
    def x(self):
        return self.__x


    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

A method which is used for getting a value is decorated with "@property", i.e. we put this line directly in front of the header. The method which has to function as the setter is decorated with "@x.setter". If the function had been called "f", we would have to decorate it with "@f.setter".
Two things are noteworthy: We just put the code line "self.x = x" in the __init__ method and the property method x is used to check the limits of the values. The second interesting thing is that we wrote "two" methods with the same name and a different number of parameters "def x(self)" and "def x(self,x)". We have learned in a previous chapter of our course that this is not possible. It works here due to the decorating:

```
>>> from p import P
>>> p1 = P(1001)
>>> p1.x
1000
>>> p1.x = -12
```

```
>>> p1.x
0
```

Alternatively, we could have used a different syntax without decorators to define the property. As you can see, the code is definitely less elegant and we have to make sure that we use the getter function in the __init__ method again:

```
class P:

    def __init__(self,x):
        self.set_x(x)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    x = property(get_x, set_x)
```

There is still another problem in the most recent version. We have now two ways to access or change the value of x: Either by using "p1.x = 42" or by "p1.set_x(42)". This way we are violating one of the fundamentals of Python: "There should be one-- and preferably only one --obvious way to do it."

We can easily fix this problem by turing the getter and the setter method into private methods, which can't be accessed anymore by the users of our class P:

```
class P:

    def __init__(self,x):
        self.__set_x(x)

    def __get_x(self):
        return self.__x

    def __set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
```

```
            self.__x = 1000
        else:
            self.__x = x

    x = property(__get_x, __set_x)
```

Even though we fixed this problem by using a private getter and setter, the version with the decorator "@property" is the Pythonic way to do it!

The following example shows a class, which has internal attributes, which can't be accessed from outside. These are the private attributes self.__potential_physical and self.__potential_psychic. Furthermore we show that a property can be deduced from the values of more than one attribute. The property "condition" of our example returns the condition of the robot in a descriptive string. The condition depends on the sum of the values of the psychic and the physical conditions of the robot.

```
class Robot:

    def __init__(self, name, build_year, lk = 0.5, lp = 0.5 ):
        self.name = name
        self.build_year = build_year
        self.__potential_physical = lk
        self.__potential_psychic = lp

    @property
    def condition(self):
        s = self.__potential_physical + self.__potential_psychic
        if s <= -1:
           return "I feel miserable!"
        elif s <= 0:
           return "I feel bad!"
        elif s <= 0.5:
           return "Could be worse!"
        elif s <= 1:
           return "Seems to be okay!"
        else:
           return "Great!"

if __name__ == "__main__":
    x = Robot("Marvin", 1979, 0.2, 0.4 )
    y = Robot("Caliban", 1993, -0.4, 0.3)
    print(x.condition)
    print(y.condition)
```

Magic Methods and Operator Overloading

Binary Operators

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Extended Assignments

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Unary Operators

Operator	Method

-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Comparison Operators

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called *child classes*, and the classes that child classes are derived from are called *parent classes*.

It's important to note that child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an `object`, which generally all other classes inherit as their parent.

```
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print(jim.description())

# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))
```