Python: An introduction to classes

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts of Southern Switzerland

What is an object?

In computer science, the concept of object is equivalent to that of the real world (a car, a tree, a telephone, . . .).

Often, objects are even computer representations of the objects of the real world (think about video games).

We therefore need a new paradigm that allows us to run a sequence of instructions on computer objects rather than on simple values.

What is an object?

If we want to describe an object, both in the real and digital world, we need basically two important characteristics:

- State: it describes the current situation of an object
- Behavior: it describes a set of functionality or actions that the object is capable of providing

Object example

An object of type `Car` could for example have the following characteristics.

State:

- the engine is running
- the driver is seated
- the doors are closed
- internal temperature
- external temperature
- fuel level

Behavior:

- start up the engine
- open the doors
- refill the fuel tank
- accelerate
- brake
- travel in time ;-)

Object-Oriented Programming

When the complexity of the programs increases, the manageability of the source code is of fundamental importance. Solutions are needed that foster modularity and re-usability for both data structures and algorithms, such as for example lists and functions.

Object-Oriented Programming is one of these solutions, particularly useful for medium or large programs.

The emphasis is on developing independent program portions easily reusable: objects.

Object-Oriented Programming

Object-Oriented Programming is possible also with procedural programming tools, such as the C language.

However, it is complex because the developer has to make sure that it is applied consistently

Other languages, which offer a specific support for object-oriented programming, provide dedicated tools that ease the word, such as:

- classes and interfaces
- methods
- encapsulation
- inheritance and polymorphism

Procedural vs. object-oriented

Procedural programming does not provide specific solutions to associate the behavior to the objects.

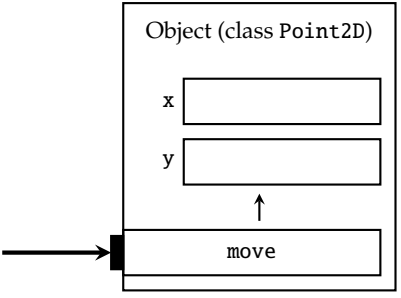
It is possible to collect and aggregate the information of the states in variable and, separately, implement the behavior in functions.

The state and behavior of the program parts remain separate.

A new paradigm

In Object-oriented programming what we want to obtain is the opposite:

- explicitly and indivisibly associate the state and behavior of each object of the program



Classes

In the real world, similar objects are categorized into object classes.

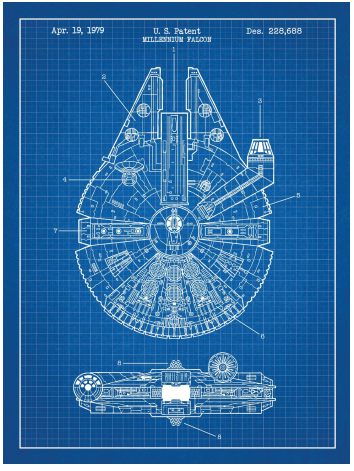
In computer science it is often used the same concept of "class".

With classes it is possible to specify the common features of multiple objects of the same kind.

Classes

If we compare to the real world, a class is a "blueprint" to create objects.

Class



Object

Object and classes, instances

For example, Luca's car and Francesco's car are different objects of the car kind. They are both part of the `Car` class.

Objects that are part of a certain kind of class are said to be instances of a class. Therefore, Luca's and Francesco's car are instances of the `Car` class.

Object and classes

Summing up:

- A class represent a data type
- An object member of a class is an instance of that data type

As it happens with each variable containing a built-in data type, each object is stored in a memory space with the data contained in the object.

The class defines which are the instance variables, the actual values are contained into objects, not in classes.

What we can put inside a class?

As discussed before, one of the principles behind OOP is to model the state and behavior of real world elements.

In order to express this two concepts inside a class we can use respectively attributes and methods.

- Attributes are the characteristics (state) associated with a class type.
- Methods are the functions (behavior) that operates with on an instance of a class.

To go back to the car example, the color, model, make and year are some of the possible attributes associated with each car, whereas the ability to start the engine, accellerate or brake would be methods.

Class definition

A new class can be defined by the special keyword `class` followed by a `class name` and a colon.

The convention for the naming is that the class name should start with a capital letter (e.g. Car, Person)

```
class Car:
```

Inside the body of a class, which needs to be indented, it is possible to define methods and instance variables.

Instance variables

As already mentioned, it is possible to define inside a class some attributes, also called instance variables.

In the following example we defined a `Car` class containing the attributes make, color and year.

```
class Car:
    def __init__(self, make, color, year):
        self.make = make
        self.color = color
        self.year = year
```

Instance variables needs to be defined inside a special method named `__init__`

The value of instance variables will be accessible from a `Car` instance using the variable name (e.g. `my_car.color`).

Constructor

The `__init__` method is a special method that is run when a new instance (a new concrete object) is created.

This special method is known with the name `constructor`, because is the one actually involved with the creation of a new object.

It is possible to define the `__init__` method so that it takes from the external world the needed data to populate internal attributes as seen in the previous slide.

Constructor

The constructor takes as first parameter the keyword `self` which refers to the instance being created. The naming of this variable is a convention in python, but it would work also with other names.

If a class has no constructor defined, it is still possible to create an instance since a default empty constructor is available out of the box.

```
class Car:
    # definition of an empty class
    pass

# usage of the default empty constructor
car = Car()
```

If we define a custom `__init__` method, then we loose the possibility to call the default empty constructor.

Creating a class instance

To actually create a class instance by running its constructor method, we use the name of the class, followed by a set of parentheses ().

It is possible to pass arguments to the constructor, exactly how we would do with normal functions.

```
my_car = Car('mercedes', 'black', 2020)

# accessing instance variables
print(my_car.color)
# prints 'black'
```

Changing instance variables values

Once we have created an instance of a class, it is possible not only to read the content of a certain instance variable (using the dot notation as in previous examples), but also to modify the values assigned.

To do that, we just use the dot notation on the instance name, followed by the name of the instance variable (attribute) we want to modify. We can then for example assign new values as if it where a normal variable.

```
my_car = Car('mercedes', 'black', 2020)

# accessing instance variables
print(my_car.color)
# prints 'black'

# changing value of an instance variable
my_car.color = 'red'
print(my_car.color)
# prints 'red'
```

Class definition and instantiation: example Car

```python
# definition of a Car class
class Car:
    def __init__(self, make, color, year)
        self.make = make
        self.color = color
        self.year = year

# usage of the Car class to create instances
car1 = Car(make='Lamborghini', color='green', year=2019)
car2 = Car(make='McLaren', color='black', year=2015)

print(car1.make)
# prints 'Lamborghini'

print(car2.make)
# prints 'McLaren'
```

Class definition and instantiation: example Dog

```
# definition of a Dog class
class Dog:
    def __init__(self, name, kind, color)
        self.name = name
        self.kind = kind
        self.color = color

# usage of the Dog class to create instances
dog1 = Dog(name='Mikey', kind='Labrador', color='beige')
dog2 = Dog(name='Willie', kind='Rottweiler', color='black')

print(dog1.kind)
# prints 'Labrador'

print(dog2.kind)
# prints 'Rottweiler'
```

Instance methods

So far we learned how to define classes and how to add some attributes.

This means we are just managing the state of our objects but we still need something different in order to also describe the behavior of our objects.

This is done with so called instance methods.

Instance methods

The definition of an instance method follows the same syntax already used to define functions.

We use the `def` keyword, we define a function name and a list of parameters.

The only special thing is that each method, exactly as the constructor, requires as first parameter the `self` keyword.

The `self` keyword is a representation of the instance on which the method is executed. In other languages like Java, it is implicit (no need to define a parameter) and is referred as `this` instead of `self`.

```
class Car:

    def start_engine(self):
        print("wroom!")
```

Instance methods

Once we define an instance method in our class, is then possible to call the method on an instance of that class.

Method call is achieved using the name of the variable holding an instance, followed by a dot, the name of the method and a set of parentheses.

```
# definition of the class
class Car:

    # definition of an instance method
    def start_engine(self):
        print("wroom!")

# creation of a Car instance
car = Car()

# call of an instance method
car.start_engine()
# prints 'wroom!'
```

Putting state and behavior together: example

```python
# definition of the class
class Car:

    # constructor
    def __init__(self, make, color, year, is_radio_on):
        # instance variables
        self.make = make
        self.color = color
        self.year = year
        self.is_radio_on = is_radio_on

    # instance method
    def start_engine(self):
        print("wroom!")

    # instance method
    def toggle_radio(self):
        self.is_radio_on = not self.is_radio_on
        print(f'radio is now {"on" if self.is_radio_on else "off"}')
```

Putting state and behavior together: example

```
# creation of a Car instance
car = Car('Honda', 'red', 2020, False)

# call of an instance method
car.start_engine()
# prints 'wroom!'

# access to an instance variable
print(car.is_radio_on)
# prints False

# call of an instance method
car.toggle_radio()
# prints 'radio is now on'
```

More on classes...

So far we have just scratched the surface on basic stuff you can do, however, there is still much more to classes and OOP to say:

- Class variables
- Class methods
- Static methods
- Encapsulation
- Inheritance
- Polymorphism
- ...

You will learn more on this topics in later semesters.

Where and how should we declare a class?

Classes can be defined in the same file as the program using them, outside the main context;

```
class Car:

    # constructor
    def __init__(self, make, color, year):
        self.make = make
        self.color = color
        self.year = year

if __name__ == '__main__':

    car = Car('McLaren', 'black', 2020, True)
```

When we develop more complex programs, it is however better to create classes in separate modules (files) and import them when needed, so that the code become easier to maintain.

Importing classes: single class

Let's assume that we wrote our `Car` class in a file named `car.py`.

We now want to write a program that actually uses the class is in another module named `my_program.py`

To import in our program an externally defined class we use the following syntax:

```
from module_name import class_name
```

Example:

```
from car import Car

if __name == '__main__':

    # use the imported Car class
    my_car = Car('audi', 'grey', 2018)
```

Importing classes: multiple classes

It is possible to define multiple classes in the same file. If we do that, we should take care that this are classes somehow related to each other.
We could for example create a module named `formula1` containing the classes `Car` and `Driver`.

The syntax to import in our program multiple externally defined classes is:

```
from module_name import class_name_1, class_name_2
```

Example:

```
from formula1 import Car, Driver

if __name == '__main__':

    # use the imported classes
    my_car = Car('McLaren', 'black', 1998)
    my_driver = Driver('Mika', 'Hakkinen', 'Finland')
```

Importing classes: all classes from a module

It is possible also possible to import all the classes from a certain module.

The syntax to import in our program all the classes defined in a module is:

```
from module_name import *
```

Example:

```
from formula1 import *

if __name == '__main__':

    # use the imported classes
    my_car = Car('McLaren', 'black', 1998)
    my_driver = Driver('Mika', 'Hakkinen', 'Finland')
```

Using other classes

So far we have seen the basics on how to create and use our own classes.

As discussed previously, we usually write classes in order to model real world objects that we need to solve our problems.

Another point is that we want to maximize the code reuse. If we implemented a class, we can import and use it whenever needed.

Speaking about reuse... sometimes we can also look around and see if someone already did the job for us.

Remember that we saw that in Python standard library we have modules to work with different file formats (json, csv, ...) ?

The `Path` class: example

Let's imagine that we need to build a class to represent and manipulate file paths with all their properties. We would have to implement functionalities to get the absolute and relative paths, to know if the path provide is a directory or not and many other stuff.

There is no need to write our own implementation from scratch. Since it is a common developer problem, someone already gave us something we can use.

The `pathlib` module from the Python standard library for instance, provides us with a `Path` class that fits perfectly our needs!

The Path class: example

If we have a file named `abc.txt` and we want to use path related we could take advantage of the `Path` class as follows:

```
from pathlib import Path

    if __name__ == '__main__':
        # create an instance of the Path class
        p = Path('abc.txt')

        # check if the file exists
        print(p.exists())
        # prints True

        # get the absolute path of the file
        print(p.absolute())
        # prints /absolute/path/to/the/file (e.g /home/rob/abc.txt)

        # check if the file is a directory
        print(p.is_dir())
        # prints False
```

The Path class: example

```
# check if it is a file
print(p.is_file())
# prints True

# get the name of the file owner
print(p.owner())
# rob

# deletes a file
p.unlink()
# file abc.txt has been deleted
```

The `Path` class

The methods in the example above are just some of those available on the Path class.

Please refer to the `pathlib` module official documentation for the complete list of methods and attributes available.

Summary

- OOP
- Classes
- Class definition
- Instance variables
- Constructor
- Instance methods
- Class imports
- Class usages
- The `Path` class

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapters 26, 27
- Python Crash Course, no starch press - Eric Matthes: Chapters 9
- Python Official Documentation: https://docs.python.org/3/tutorial/classes.html
- LearnByExample: https://www.learnbyexample.org/python/