

Object Oriented Programming in Python

Classes and objects



CentraleSupélec

Outline

- 1 Object-oriented programming: basics
- 2 Classes in Python

Outline

1 Object-oriented programming: basics

2 Classes in Python

Object-oriented programming (OOP)

OOP: a programming paradigm for directly **mapping real-life problems into a program**

- it is based on the notion of **class** (a user-defined data type)
- and **objects** (instances of a given class)

an **object** is a data structure that contains:

- **data**: in form of variables called **attributes** or **fields**
- **behaviour**: in form of procedures called **methods**

Real-world objects

real-world objects share two characteristics: they all have a **state** and a **behaviour**

examples of real-world objects

- **Dog:**
 - **state:** name, color, breed, hungry, ...
 - **behaviour:** barking, fetching, wagging tail, eating, ...
- **Bicycle:**
 - **state:** current gear, current pedal cadence, current speed, ...
 - **behaviour:** changing gear, changing pedal cadence, applying brakes, ...

Example: class “Bicycle” and class “Rider”

class name	Bicycle
attributes (state variables)	int gear; float speed;
methods (class interface)	void upshift(); void downshift(); void increase_speed(); void decrease_speed();

	Rider	class name
	int age; float energy;	attributes (state variables)
	void upshift(); void downshift(); void pedal_faster(); void pedal_slower();	methods (class interface)

What is a (software) class ?

class: the *blueprint* characterising a category of objects

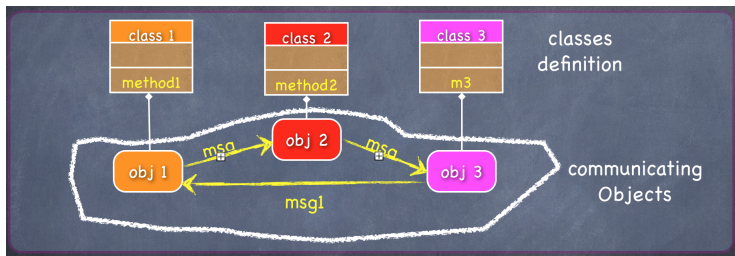
- defines the **attributes** representing the **state** of objects
- defines the **methods** representing the **behaviour** of objects

several **objects** can be instantiated from a given **class**

What an Object-Oriented program looks like?

an Object-Oriented program consists of:

- a collection of **classes definitions**
- a collection of **objects' instances**



computation: instantiated objects perform the desired computation by invoking each other methods (i.e. by exchanging messages)

Outline

1 Object-oriented programming: basics

2 Classes in Python

Classes in Python

- **Class:** bundle together *data* and *functionalities*
 - defining a Class defines a *new data type* allowing new *instances* (objects) of that data type to be made
- **Object:** Objects are an encapsulation of variables and functions into a single entity.
 - Objects get their variables and functions from classes.

Class definition syntax

in Python a class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example:

```
class MyClass:
    # this is a comment
    """ A simple example class """

    # this is an attribute
    i = 12345

    # this is a method
    def f(self):
        return 'hello world'
```

defines a class called `MyClass` with one attribute named `i` and one function named `f`

Object instantiation

```
class MyClass:  
    i = 12345  
  
    def f(self):  
        return 'hello world'
```

instantiation of an object of a class: uses *function call notation*

```
myobject = MyClass() #
```

myobject is an object of type MyClass

Accessing object's variables and functions

To access a variable or a function of an object you use the `.` operator

```
class MyClass:
    i = 12345

    def f(self):
        print('hello world')

myobject = MyClass()
print(myobject.i) # access the attribute 'i' of 'myobject' prints 12345
myobject.f(); # execute function f() of 'myobject' hence prints "hello world"
```

Object's initialisation: the `__init__()` function

All classes have a function called `__init__()`, which is always executed when the class is being initiated.

`__init__()` function is used to assign values to object's attributes and perform operations necessary when the object is being created:

```
class Person:
    def __init__(self, name, age):
        self.name = name # declares and initialise an attribute called 'name'
        self.age = age # declares and initialise an attribute called 'age'

p1 = Person("John", 36) # create a Person object with name "John" and age 36
p2 = Person("Mary", 23) # create a Person object with name "Mary" and age 23

print(p1.name)
print(p1.age)
print(p2.name)
print(p2.age)
```

Remark: the `__init__()` function is automatically called whenever an object is created

The self parameter (of a class method)

Every method defined in a class must have at least one parameter (which refers to the object of the class)

such parameter is conventionally denoted `self` and MUST BE the first parameter of a method

```
class Person:
    def __init__(self, name, age): # 'self' is used to refer to the 'name' and 'age' of the object being created
        self.name = name
        self.age = age

    def myfunc(abc): # in this case 'abc' is used in place of 'self'
        print("Hello my name is " + abc.name)

p1 = Person("John", 36) # create a Person object with name "John" and age 36
p1.myfunc()
```

Inheritance: parent-class and child-class

Inheritance allows to define a class as a **child-class** of a **parent-class**

- **Parent-class**: the class being inherited from,
- **Child-class**: the class that inherits from another class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class

```
class Student(Person): # Student is a child-class of Person
    def __init__(self, name, age): # this __init__ overrides the __init__ in Person
        self.name = name
        self.age = age
```

Remark: the `__init__()` in the child-class **overrides** that in the parent class

when you create a `Student` object the `__init__()` in `Student` is executed not that in `Person`

The `super()` function (inheriting from parent)

`super()` allows to accessing inherited methods that have been overridden in a class.

commonly used in child-class `__init__()` to delegate to the parent-class the initialisation of inherited attributes

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname # the first name of a person
        self.lastname = lname # the last name of a person

    def printname(self):
        print(self.firstname, self.lastname) # prints the person's first and last name

class Student(Person):
    def __init__(self, fname, lname, year): #overrides the __init__ of Person
        super().__init__(fname, lname) # call __init__ of Person to initialise first and last name of Student
        self.graduationyear = year # year of graduation

    def welcome(self): # print a welcome message for a Student
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

x = Student("Mike", "Olsen", 2019) # create a Student object
x.welcome() # print welcome message for Student x
```

Class versus instance variables

Instance variables: used for data unique to each instance object

Class variables: used for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance

d = Dog('Fido') # create a dog
e = Dog('Buddy') # create a dog
print(d.kind) # prints 'canine' which is shared by all dogs
print(e.kind) # prints 'canine' which is shared by all dogs
print(d.name) # prints 'Fido' which is unique to d
print(e.name) # prints 'Buddy' which is unique to e
```