L20: Object-oriented programming

```
In [1]: # Let's start by defining an empty class!
        # The pass keyword indicates an empty block (that will be possibly filled with something later on)
        # or just indicates that no actions are designed for that scope
        #
        class Robot:
            pass

        # We can create an instance of the class
        x = Robot()
        # ... there's are neither methods nor attributes, therefore it's just a namholder at the moment
```

```
In [2]: # Can we add class attributes "on-the-fly"? Yes! (but it's not a good practice!)
        Robot.type = "arm manipulator"
        Robot.brand = "Kuka"

        # These statements set both the name and the default value of a new attributed of the class
        print(x.type, x.brand)
```

```
arm manipulator Kuka
```

```
In [3]: # Interesting ... we can dynamically adapt the structure of a class, if we want,
        # but how does this happen? Because behind the scenes classes are implemented using dictionaries,
        # such that adding an attribute is similar to add a key-value pair.
        # Invoking the magic method __dict__ reveals the structure
        #
        Robot.__dict__
```

```
Out[3]: mappingproxy({'__module__': '__main__',
                      '__dict__': <attribute '__dict__' of 'Robot' objects>,
                      '__weakref__': <attribute '__weakref__' of 'Robot' objects>,
                      '__doc__': None,
                      'type': 'arm manipulator',
                      'brand': 'Kuka'})
```

```
In [4]: # The values to the attributes are just the default, call-level, values. They can be overridden
        # inside an object
        #
        x.brand = "SkyNet"
        x.type = "humanoid"
        print(x.type, x.brand)
```

```
humanoid SkyNet
```

```
In [5]: # Can we also add methods dynamically? Yes!
        def display_info(self, msg):
            print(msg, 'type:', self.type + ',', 'brand:', self.brand)

        Robot.display_info = display_info
        x.display_info('Robot -')
```

```
Robot - type: humanoid, brand: SkyNet
```

```
In [6]: # We have highlighted a connection between classes and dictionaries data structures,
        # let's see how classes can be effectvely used  to organize data.
        # A class has attributes + methods, but methods are optional, in a sense
        #
        class Robot:
            type  = "wheeled"
            brand = None
            price = 0
            since = 0

        x = Robot()
        print(x.type)

        y = Robot()
        print(y.type)


        print(x == y)
        print(x,y)
        # They are different, being different objects! We need to define a content-based equality
        # operator for the specific classes!
```

```
wheeled
wheeled
False
<__main__.Robot object at 0x1101dd710> <__main__.Robot object at 0x1101acf60>
```

```
In [7]: # We can override the values of the default attributes and set them to whatever appropriate,
        # one by one
        y.brand = 'Kuka'
        y.type = 'arm manipulator'
        x.brand = 'SkyNet'
```

```
In [8]: # A class with attributes only is however unnecessarily limiting: let's make little step further
        # and let's add an initializer, such that we can initialize class object with the right values
        # whenever a new class object is being created
        #
        class Robot:
            def __init__(self, robot_type='wheeled', brand=None, model= None, price=0, since=0):
                self.type  = robot_type
                self.brand = brand
                self.model = model
                self.price = price
                self.since = since

        # This is much better, we have the same result as before, but now we also have an initializer
        #
        x = Robot('humanoid', 'SkyNet', 'Conqueror', 10000000, 2020)
        y = Robot(brand='Kuka', price=50000)
        print(x.__dict__, '\n', y.__dict__)
```

```
{'type': 'humanoid', 'brand': 'SkyNet', 'model': 'Conqueror', 'price': 10000000, 'since': 2020}
 {'type': 'wheeled', 'brand': 'Kuka', 'model': None, 'price': 50000, 'since': 0}
```

```
In [9]: # We can access the individual attributes of a specific class object by 'name'/ label
        #
        y.price = y.price * 1.2
        y.since = 2015

In [10]: # Indeed this is "similar" to using dictionaries, when an entry was accessed by a label
         ydict = {'type': 'wheeled', 'brand': 'Kuka', 'model': '007', 'price': 50000, 'since': 0}
         print(ydict)
         print(ydict['price'], y.price)
         # Maybe the class notation is cleaner ... and with classes with the do more ...

{'type': 'wheeled', 'brand': 'Kuka', 'model': '007', 'price': 50000, 'since': 0}
50000 60000.0


In [11]: # If we have many robots (e.g., we are selling them) we might need to organize and search data
         # We can define a dictionary, or a list, or a set of Robot objects
         robots = [Robot('humanoid', 'SkyNet', 'Conqueror', 10000000, 2020),
                   Robot(brand='Kuka', price=50000)]

         robots.append(Robot(brand="Pioneer"))

         robots[2].model = 'Nav01'
         robots[1].model = 'AgileX'
         robots[1].since = 2016

In [12]: # We can perform operations on list
         for r in robots:
             print('Robot model and price:', r.model, r.type)

Robot model and price: Conqueror humanoid
Robot model and price: AgileX wheeled
Robot model and price: Nav01 wheeled


In [13]: # How do we sort the class objects? We need a criterion.
         # At this aim we can define a method that does the job!
         #
         def compare_obj(self):
             return self.since * self.price  # an arbitrary criterion here...

         # Let's add it to the class definition
         Robot.compare_obj = compare_obj

In [14]: # Let's sort the robot by the given criterion.
         # Watch out: we are passing the class method as key function.
         #
         robots.sort(key = Robot.compare_obj, reverse = True)
         for r in robots:
             print(r.brand, r.model, r.price, r.since)

SkyNet Conqueror 10000000 2020
Kuka AgileX 50000 2016
Pioneer Nav01 0 0


In [15]: # Maybe we can also add a method to have a nice print out of the info about a robot
         #
         def get_info(self):
```

```
    info = 'Model {} made by {} is a {} robot, price is {}, we have it since {}'.format(
            self.model, self.brand, self.type, self.price, self.since)
    return info
Robot.get_info = get_info
```

In [16]:
```
# Let's use it on the sorted robot list
#
for r in robots:
    print(r.get_info())
```

```
Model Conqueror made by SkyNet is a humanoid robot, price is 10000000, we have it since 2020
Model AgileX made by Kuka is a wheeled robot, price is 50000, we have it since 2016
Model Nav01 made by Pioneer is a wheeled robot, price is 0, we have it since 0
```

In [17]:
```
# We see how naturally our class has been growing, and it's an extremely flexible tool to
# represent and deal with information data. It provides a 'label-based' access to information,
# and it pairs the information with the methods to manipulate it.
#
# Let's write down the full class so far.
#
class Robot:
    def __init__(self, robot_type='wheeled', brand=None, model= None, price=0, since=0):
        self.type  = robot_type
        self.brand = brand
        self.model = model
        self.price = price
        self.since = since

    def compare_obj(self):
        return self.since * self.price  # an arbitrary criterion here...

    def get_info(self):
        info = 'Model {} made by {} is a {} robot, price is {}, we have it since {}'.format(
            self.model, self.brand, self.type, self.price, self.since)
        return info
```

In [18]:
```
# One question: is it correct / appropriate / safe that the user of a class object
# gets direct access to the attribute variables?
# First, notice that in order to manipulate a class variable (e.g., by changing its value),
# the user must be aware of the data type of the variable, and also how this is used (e.g.,
# it might have some restrictions on the values it takes). E.g., is price supposed to be
# integer or float? Is since an integer or a date format?
#
# A good practice (information hiding) consists in hiding the details to the user, and providing
# instead so-called getter and setter methods!
# In our example it may seem a bit redundant, but it's the way to go to ensure code transparency,
# reusability, and safety.
#
# Let's add one setter and one getter for the price attribute
#
def set_price(self, price):
    self.price = price

def get_price(self):
    return self.price

Robot.set_price = set_price
Robot.get_price = get_price
```

```
In [19]: x = Robot()
         x.set_price(20500)
         print(x.get_price())
```

```
20500
```

```
In [20]: # With setters we can do more things (e.g., checks), and ensure code safety
         #
         def set_price(self, price):
             if price < 0:
                 print('Given price is negative!')
                 return -1
             else:
                 self.price = price
                 return price

         Robot.set_price = set_price

         price = -100
         if x.set_price(price) == -1:
             x.set_price(-price)
```

```
Given price is negative!
```

```
In [21]: # Let's move to a more complex scenario a robot shop
         # Now the robot list is an attribute of the class
         # The shop starts with robot_num robots
         #
         class RobotShop():
             def __init__(self, robot_num=0, model=None, robot_type=None ):
                 self.robots = []
                 for r in range(robot_num):
                     self.robots.append(Robot(robot_type, model, None, 0, 0))

         robot_shop = RobotShop(3, model='SkyNet', robot_type='humanoid')
```

```
In [22]: # Let's add a method for gathering info about the robots in the shop
         #
         def get_info(self):
             models = set()
             for r in self.robots:
                 models.add(r.model)

             msg = 'Robot shop features {} robots, for a total of {} different models\n'.format(
                     len(self.robots), len(models))

             for r in self.robots:
                 msg += "Model {}, Brand {}, Type {}, Price {}, Since {}\n".format(r.model, r.brand,
                                                                              r.type, r.price, r.since)
             return msg

         RobotShop.get_info = get_info

         print(robot_shop.get_info())
```

```
Robot shop features 3 robots, for a total of 1 different models
Model None, Brand SkyNet, Type humanoid, Price 0, Since 0
Model None, Brand SkyNet, Type humanoid, Price 0, Since 0
Model None, Brand SkyNet, Type humanoid, Price 0, Since 0
```

```
In [23]: # It's clear that we could add many other methods, for dealing with the complexity of
         # inventory, adding, removing, selling, inspecting, maintaining robots ...
```

```
In [24]: # Let's move to another example, that will allow to discuss the notion of inheritance
         # Below a basic class for dealing with the operations of a bank account
         #
         class BankAccount:
             def __init__(self, id_num, initial_balance):
                 self.balance = initial_balance
                 self.id = id_num
                 self.in_operations = 0
                 self.out_operations = 0

             def withdraw(self, amount):
                 self.balance -= amount
                 self.out_operations += 1
                 return self.balance

             def deposit(self, amount):
                 self.balance += amount
                 self.in_operations += 1
                 return self.balance

             def status(self):
                 return self.balance, self.in_operations, self.out_operations

             def get_id(self):
                 return(self.id)

             def add_stock_asset(self):
                 pass

             def get_stocks(self):
                 pass
         #
         # These last two methods are empty. They are there to make it clear that, by design,
         # a class dealing with  bank accounts should have methods for dealing with stock assets.
         # We will see that in a derived class these methods could be filled with some appropriate code.
```

```
In [25]: # MinimumBalanceAccount is a derived class of a BankAccount class: it inherits all the
         # properties (attributes and methods) of the base (super) class BankAccount, and specializes /
         # override the method withdraw to address the needs of a bank account with a special limitation
         # in terms of minimum balance. The method withdraw() is an example of polymorphism: transparently,
         # the same 'name' can be used by objects of different classes, withdraw() is an overloaded method.
         # Also add_stock_asset() and get_stocks() are polymophic, however, if these methods are called
         # from the base class, nothing happens since they were empty there.
         #
         class MinimumBalanceAccount(BankAccount):
             def __init__(self, id_num, minimum_balance):
                 BankAccount.__init__(self, id_num, minimum_balance)
                 self.minimum_balance = minimum_balance
                 self.assets = 0
```

```
            def withdraw(self, amount):
                if self.balance - amount < self.minimum_balance:
                    print('Sorry, minimum balance must be maintained. Current balance:',
                          self.balance)
                else:
                    BankAccount.withdraw(self, amount)
                return self.balance

            def add_stock_asset(self, asset):
                self.assets += asset

            def get_stocks(self):
                return self.assets
        #
        # Class variables of the base class can be invoked directly using the self. notation.
        # Equivalenty, methods from the base class can be invoked using self.method(). However,
        # if the method is overloaded in the derived class, in order to invoke the base class version,
        # the syntax beccomes: BaseClass.method(). This is the case of calling the __init__ and the
        # withdraw() methods in the code above (lines 10 and 18).
        #
        #
```

```
In [26]: # Example of use of the base class
         #
         john = BankAccount(9754, 10)
         john.deposit(100)
         john.withdraw(99)
         print(john.status())

         print(john.get_stocks())
```

```
(11, 1, 1)
None
```

```
In [27]: # Example of use of the derived class and of the overloaded methods
         #
         ann = MinimumBalanceAccount(76876, 100)
         print(ann.status())
         ann.deposit(200)
         print(ann.status())
         ann.withdraw(100)
         print(ann.status())
         ann.withdraw(200)
         print(ann.status())

         ann.add_stock_asset(500)
         print(ann.get_stocks())
```

```
(100, 0, 0)
(300, 1, 0)
(200, 1, 1)
Sorry, minimum balance must be maintained. Current balance: 200
(200, 1, 1)
500
```

```
In [28]: # Another simple example of class inheritance
         # Let's define a class to represent a generic person
```

```
#
class Person:

    def __init__(self, first, last, age=None):
        self.firstname = first
        self.lastname = last
        self.age = age

    def name(self):
        return self.firstname + " " + self.lastname

    def set_age(self, age):
        self.age = age

    def get_age(self):
        return self.age

    def set_job(self, job):
        self.job = job

    def get_job(self):
        return self.job

    def set_education(self, degrees):
        pass

    def get_education(self):
        pass

x = Person("Marge", "Simpson", 30)
x.set_job('housemaker')
print('{}, {}, {}'.format(x.name(), x.get_age(), x.get_job()))
```

```
Marge Simpson, 30, housemaker
```

In [29]:

```
# An employee is a person + some other properties that are specific to the company / job.
# Therefore, it makes sense to define the class Employee as a child (derived) class of the
# Person class, that becomes the base  (super) class. This will save a lot of code rewriting.
# Employee adds a few attributes and methods. More could be added here of course ...
#
class Employee(Person):

    def __init__(self, first, last, age=None, staff_num=None, position=None, since=None):
        Person.__init__(self, first, last, age)
        self.staff_number = staff_num
        self.position = position
        self.since = since

    def get_employee(self):
        return [self.name(), self.staff_number, self.get_age(), self.position, self.since]

    def set_education(self, degrees):
        self.bachelor = degrees['bachelor']
        self.master = degrees['master']
        self.doctorate = degrees['doctorate']

    def get_education(self):
```

```
                return 'Bachelor: {}, Master: {}, Doctorate: {}'.format(self.bachelor,
                                                                        self.master, self.doctorate)

        y = Employee("Homer", "Simpson", 40, "1007", 'troublemaker')
        y.set_education({'bachelor': 'drinkology', 'master':'sofalogy', 'doctorate':'still enrolled'})
        print(y.get_employee())
        print(y.get_education())
```

```
['Homer Simpson', '1007', 40, 'troublemaker', None]
Bachelor: drinkology, Master: sofalogy, Doctorate: still enrolled
```