A Python Lecture Series

Lecture 3

by Luca Mingarelli

Lecture 3

Content:

- Object Oriented Programming
- Classes and Objects in Python
- Class Attributes
- Operator Overloading
- Public, Protected, and Private Attributes
- Properties, or *getters* and *setters*
- Types of Methods
- Inheritance

A Brief overview of different programming paradigms

- **Procedural**: A sequence of instructions to elaborate the input in order to solve a given problem. *E.g.*: C, Pascal, UNIX(sh).
- **Declarative**: A specification describes the problem, and the language figures out how to carry out the task. *E.g.*: SQL.
- **Functional**: The problem to be solved is decomposed into a set of functions. E.g. Q, Haskell.
- **Object-Oriented**: A problem is broken down into abstract collections of objects with internal states and methods to query and modify them. *E.g.*: Java.

- **Multi-paradigm**: a blend of two or more of the previous paradigms. E.g.: C++, Python.

Object Oriented Programming

- **Object**: A notion of *entity*, characterised by some values or *attributes* and functions or *methods*.
- **Class**: The abstract notion through which we group objects of the same type together.

Four major principles:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Humans have the habit of thinking in terms of abstract concepts. Think for example of a car, which is a well defined uniform object in our mind: when we think of it, we abstract from its inner functioning and components (**abstraction**), as well as from its content (**encapsulation**).

In addition, in our minds, we often group objects in broader categories (or classes), grouping together objects with similar purposes and functioning, thus abstracting further. Think of the generalised class of vehicles, which includes cars, vans, trucks and so on. Many of the attributes and funtions of these are nonetheless shared among different subclasses. This relates to the concepts of **inheritance**.

Finally, different objects, part of the same broader class, might perform a specific action differently, depending on some of their distinctive attributes. As an example, loading a car's boot is a specific action which requires different implementation depending on whether your car has a boot in front or on the rear. The action to be performed is different, nonetheless we refer to both actions as *loading the car*. This is what we mean by function's **polymorphism**.

In [1]:
```
print(type(2))
print(dir(2))
```

```
<class 'int'>
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__dela
ttr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor_
_', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs_
_', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int
__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__
ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__ra
nd__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv_
_', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setatt
r__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_byte
s', 'imag', 'numerator', 'real', 'to_bytes']
```

In [2]:
```
(2).__class__
```

Out[2]:
```
int
```

In [3]:
```
(2).numerator
```

Out[3]:
```
2
```

In [4]:
```
(2).__str__()
```

Out[4]:
```
'2'
```

In [5]:
```
(2).__add__(10)
```

Out[5]:
```
12
```

In [6]:
```
int.__add__(2,10)
```
Out[6]:
```
12
```

The distinction between a class and an object

In [7]:
```
class Person:
    pass
```

In [8]:
```
person1 = Person()
person2 = Person()
print(person1)
print(person2)
```

```
<__main__.Person object at 0x104ddf978>
<__main__.Person object at 0x104ddf908>
```

Notice the two different locations in memory: `person1` and `person2` are two distinct instances of the class `Person`.

In [9]:
```
person1.name="Giuseppe Luigi Lagrangia"
person1.yob = 1736
```

In [10]:

```
print(person1.name,", born",
      person1.yob)
person2.name
```

```
Giuseppe Luigi Lagrangia , born 1736

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-f4169f585d6d> in <module>
      1 print(person1.name,", born",
      2       person1.yob)
----> 3 person2.name

AttributeError: 'Person' object has no attribute 'name'
```

A silly but useful usage of classes: containers

In [11]:
```
class box:
    pass
```

In [12]:
```
B = box()
B.string = "I can store a string"
B.list = [1,2,3,4,5]
```

In [13]:
```
B.mean = lambda x: sum(x)/len(x)
B.mean
```

Out[13]:
```
<function __main__.<lambda>(x)>
```

In [14]:
```
print(B.string)
print(B.list)
B.mean(B.list)
```

```
I can store a string
[1, 2, 3, 4, 5]
```

Out[14]:
```
3.0
```

In [15]:
```
B.box2 = box()
B.box2.value = 100
```

In [16]:
```
print(B.box2.value)
```

```
100
```

Classes are specified by their:

- Attributes (and Properties)
- Methods

Objects are instances of a class and as such are specified by their:

- Class
- Name

In [17]:
```
class a_python_course:
    def __init__(self):
        self.name = "A Python Course"
```

In [18]:
```
APC = a_python_course()
print(APC.name)
type(APC)
```

```
A Python Course
```

Out[18]:
```
__main__.a_python_course
```

Warning: To define a class is different from initialising it.

Notice:

- `__init__` is a special function called at the very beginning of the instantiation of the class, used to initialise it. It is not mandatory and can take several arguments.

- `self` is a special keywork used to address to object itself.

- **Attributes** are defined attached to the keywork `self`, separated by `.`.

- **Methods** are defined similarly to function definitions, however they require the keywork `self` as a first argument.

In [19]:

```python
class a_python_course:
    def __init__(self):
        self.name = "A Python Course"
    def next_lecture(self):
        import datetime as dt
        D = dt.date.today()
        print(D+ dt.timedelta(days=7))
        print("at 16.30,room HS32.50.")
```

In [20]:

```python
APC = a_python_course()
APC.next_lecture() ## or,alternatively:
#a_python_course.next_lecture(APC)
```

```
2019-05-22
at 16.30,room HS32.50.
```

An object's attribute can be easily initialised when the object is created:

In [21]:
```
class attendee:
     '''A class for attendees
of some course.'''
    def __init__(self, name):
        self.name="My name is "+name
```

In [22]:
```
d = attendee("Paul")
print(d.__doc__)
d.name
```

```
A class for attendees
of some course.
```

Out[22]:
```
'My name is Paul'
```

Apart from the `self` keywork, methods are very much the same as functions:

In [23]:
```
class attendee:
    def __init__(self, name, homework_response = None):
        self.name = name
        if not homework_response:
            self.homework = "Homework done by "+name
        else:
            self.homework = self.name+ ": " + homework_response
```

Encapsulation works for any kind of data structure, also for user defined classes:

In [24]:
```
class a_python_course:
    def __init__(self):
        self.name = "A Python Course"
        self.attendees = list()
    def next_lecture(self):
        import datetime as dt
        D = dt.date.today()
        print(D + dt.timedelta(days=7))
        print("at 16.30,room HS32.50.")
    def register_attendee(self,
                          attendee):
        self.attendees.append(attendee)
    def handin_homework(self):
        for attendee in self.attendees:
            print(attendee.homework)
```

In [25]:
```
APC = a_python_course()
A = list() ##list of attendees
A+=[attendee("Julian")]
A+=[attendee("Domenico",
            "The dog ate my homework")]
A+=[attendee("Elisa")]
A+=[attendee("Simone", "Ehh...")]
for att in A:
    APC.register_attendee(att)
```

In [26]:

```
APC.handin_homework()
```

```
Homework done by Julian
Domenico: The dog ate my homework
Homework done by Elisa
Simone: Ehh...
```

Moreover, notice that the encapsulated data is unique to each instance of a class.

Warning: Attributes and methods **cannot** have the same name.

Class Attributes

Class attributes are shared among all instances of the class, and can be accessed either through the class' name or through each instance.

In [31]:
```
class Person:
    _type = "human"
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
    def get_info(self):
        print(self.name,"is a",
              self._type,"born in",
              self.yob)
```

In [32]:
```
GLL=Person("Giuseppe Luigi Lagrangia",
           1736)
GLL.get_info()
```

```
Giuseppe Luigi Lagrangia is a human born in 1736
```

Notice class attributes can be overriden for an individual instance, without affecting the others.

In [33]:
```
MK = Person("Milan Kundera",1929)
MK._type = "writer" ## however...
# Person._type = "writer"
```

In [34]:

```
MK.get_info()
GLL.get_info()
```

```
Milan Kundera is a writer born in 1929
Giuseppe Luigi Lagrangia is a human born in 1736
```

Keep track of the number of instances

In [35]:
```
class Person:
    _num_of_persons = 0
    _type = "human"
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
        Person._num_of_persons += 1
    def get_info(self):
        print(self.name,"is a",
              self._type,"born in",
              self.yob)
```

In [36]:
```
GLL=Person("Giuseppe Luigi Lagrangia",
             1736)
MK = Person("Milan Kundera",1929)
JPR = Person("Jean-Philippe Rameau",
             1683)
Person._num_of_persons
```

Out[36]:
```
3
```

However notice that reinstanciating the same object, still increases the counter `_num_of_persons`.

However, using `hash()` ...

In [159]:

```
class Person:
    _num_of_persons = 0
#     __persons = set()
    _type = "human"
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
#         Person.__persons.add(hash((name,yob)))
        Person._num_of_persons += 1
#         Person._num_of_persons=len(Person.__persons)
    def get_info(self):
        print(self.name,"is a",self._type,"born in",self.yob)
```

Operator overloading

Method's name	Description
`object.__add__(self, other)`	implements the + operator
`object.__sub__(self, other)`	implements the – operator
`object.__mul__(self, other)`	implements the `*` operator
`object.__matmul__(self, other)`	implements the `@` operator
`object.__truediv__(self, other)`	implements the / operator
`object.__floordiv__(self, other)`	implements the // operator
`object.__mod__(self, other)`	implements the `%` operator
`object.__pow__(self, other[, modulo])`	implements the `**` operator
`object.__lshift__(self, other)`	implements the << operator
`object.__rshift__(self, other)`	implements the >> operator
`object.__lt__(self, other)`	implements the < operator
`object.__le__(self, other)`	implements the <= operator
`object.__eq__(self, other)`	implements the == operator
`object.__ne__(self, other)`	implements the `!=` operator
`object.__gt__(self, other)`	implements the > operator
`object.__ge__(self, other)`	implements the >= operator
`object.__and__(self, other)`	implements the `&` operator
`object.__xor__(self, other)`	implements the ^ operator
`object.__or__(self, other)`	implements the | operator
`object.__neg__(self)`	implements the unary – operator
`object.__pos__(self)`	implements the unary + operator
`object.__abs__(self)`	implements the `abs()` operator
`object.__str__(self,)`	defines what the function `print()` should return
`object.__repr__(self,)`	defines the string representation of `object`
...	...

Prefixing `i` to the name, as in `object.__iadd__(self, other)`, allows to implement augmented operations such as `+=`.

Consider the following example class:

In [37]:
```
class Complex:
    def __init__(self, realpart,
                 imagpart):
        self.r = realpart
        self.i = imagpart
x = Complex(3.0, -4.5)
x.r, x.i
```

Out[37]:
```
(3.0, -4.5)
```

In [38]:
```
z1 = Complex(1,1)
z2 = Complex(1,2)
z1+z2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-4f73bb7ace45> in <module>
      1 z1 = Complex(1,1)
      2 z2 = Complex(1,2)
----> 3 z1+z2

TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

In [39]:

```python
class Complex:
    def __init__(self, realpart,
                 imagpart):
        self.r = realpart
        self.i = imagpart
    def __add__(self, other):
        real = self.r + other.r
        imag = self.i + other.i
        return Complex(real,imag)
```

In [40]:

```python
z1 = Complex(1,1)
z2 = Complex(1,2)
print(z1+z2)
z1+z2
```

```
<__main__.Complex object at 0x103b458d0>
```

Out[40]:

```
<__main__.Complex at 0x103b45668>
```

In [41]:

```python
class Complex:
    def __init__(self, realpart,
                 imagpart):
        self.r = realpart
        self.i = imagpart
    def __add__(self, other):
        real = self.r + other.r
        imag = self.i + other.i
        return Complex(real,imag)
    def __str__(self):
        return str(self.r)+"+"+str(self.i)+"i"
    def __repr__(self):
        return self.__str__()
```

In [42]:

```python
z1 = Complex(1,1)
z2 = Complex(1,2)
print(z1+z2)
# z1-z2      #NOTICE: z1-z2 still wouldn't work!
```

```
2+3i
```

Public, Protected, and Private Attributes

Naming of attributes and methods

Name	Type	Meaning
`name`	Public	Can be freely used inside or outside of a class definition.
`_name`	Protected	Should not be used outside of the class definition, unless inside of a subclass definition.
`__name`	Private	Inaccessible and invisible (except inside of the class definition itself).

In [43]:
```python
class a_class:
    def __init__(self):
        self.__priv = "I am a private attribute"
        self._prot = "I am a protected attribute"
        self.pub = "I am a public attribute"
```

In [44]:
```python
X = a_class()
X.pub
```

Out[44]:
```
'I am a public attribute'
```

In [45]:
```python
X.pub += " and my value can be changed"
X.pub
```

Out[45]:
```
'I am a public attribute and my value can be changed'
```

In [46]:
```python
X._prot
```

Out[46]:
```
'I am a protected attribute'
```

In [47]:
```python
X._prot+=" and my value can be changed"
X._prot ## DON'T DO IT!
```

Out[47]:
```
'I am a protected attribute and my value can be changed'
```

In [38]:

```
X.__priv
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-38-249b76ef9d4d> in <module>
----> 1 X.__priv

AttributeError: 'a_class' object has no attribute '__priv'
```

Notice it says `object has no attribute '__priv'`: perfect information hiding.

Properties, or *getters* and *setters*

Assume you want to impose contraints on the possible values of an attribute. As an example, consider a class implementation of a thermometer:

In [48]:
```
class thermometer:
    def __init__(self, temperature = 0):
        self.__temperature = temperature
#     @property       #   <- decorator
    def temperature(self):
        return self.__temperature
#     print(type(temperature))
#     @temperature.setter
#     def temperature(self, value):
#         if value < -273:
#             raise ValueError("Temperature cannot go below -273.")
#         self.__temperature = value
```

In [49]:
```
T = thermometer()
T.temperature = 0
T.temperature
```

Out[49]:
```
0
```

Types of Methods

Type	Description
Instance Method	The regular method: takes one parameter `self` pointing at one instance of the class. This allows the instance method to address other attributes and methods of the class
Class Method	Obtained through the decorator `@classmethod`, takes as input the parameter `cls` (pointing to the class, as opposed to the object) allowing it to modify the class' states across all instances, but not instances' states.
Static Method	Obtained through the decorator `@staticmethod`, they can neither modify object state nor class state. Such methods are restricted in what data they can access.

Class methods

In [50]:
```
class Person:
    __type = "human"
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
    def get_info(self):
        print(self.name,"is a",self.__type,"born in",self.yob)
    @classmethod
    def set_type(cls, string):
        cls.__type = string
```

In [51]:
```
JPR = Person("Jean-Philippe Rameau", 1683)
JPR.get_info()
###
Person.set_type("composer")
JPR.get_info()
```

```
Jean-Philippe Rameau is a human born in 1683
Jean-Philippe Rameau is a composer born in 1683
```

Class Methods: alternative constructors

In [52]:

```
class Person:
    __type = "human"
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
    def get_info(self):
        print(self.name,"is a",self.__type,"born in",self.yob)
    @classmethod
    def set_type(cls, string):
        cls.__type = string
    @classmethod
    def from_single_string(cls,person_string):
        name, yob = person_string.split(';')
        return cls(name, yob)
```

In [53]:

```
JPR = Person.from_single_string("Jean-Philippe Rameau;1683")
JPR.get_info()
```

```
Jean-Philippe Rameau is a human born in 1683
```

Class Methods: how to create factory objects

Class methods also allow to easily create factory functions for the different kind of objects we want to create:

In [54]:
```
class Person:
    __type = "human"
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
    def get_info(self):
        print(self.name,"is a",self.__type,"born in",self.yob)
    @classmethod
    def jb(cls,yob):
        P = cls("J. Bernoulli", yob)
        P.__type = "scientist"
        return P
```

In [55]:
```
JB = (Person.jb(yob) for yob in [1654,1667,1710,1744,1759])
for person in JB:
    person.get_info()
```

```
J. Bernoulli is a scientist born in 1654
J. Bernoulli is a scientist born in 1667
J. Bernoulli is a scientist born in 1710
J. Bernoulli is a scientist born in 1744
J. Bernoulli is a scientist born in 1759
```

Static Methods

In [56]:
```
class Person:
    def __init__(self, name, yob):
        self.name, self.yob = name, yob
    def greet():
        return "Hi there!"
```

In [57]:
```
JB = Person("J. Bernoulli",1654)
JB.greet()

## This returns an error! Why?
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-ac05edd6c06f> in <module>
      1 JB = Person("J. Bernoulli",1654)
----> 2 JB.greet()
      3
      4 ## This returns an error!

TypeError: greet() takes 0 positional arguments but 1 was given
```

In [58]:
```
class Person:
    def __init__(self, name, yob):
        self.name, self.yob =name,yob
    @staticmethod
    def greet():
        return "Hi there!"
```

In [59]:

```python
JB = Person("J. Bernoulli",1654)
JB.greet()
```

Out[59]:

```
'Hi there!'
```

Inheritance

When writing a new class, it might be needed to *inherit* methods and functions from previously defined classes. This is often the case when the *child* class represents an abstract subset of the abstract group defined by the *parent* class from which it inherits.

```
class parent:
    <body>
    def parent_method(self):
        <method`s body>
    parent_attribute = 10
    <more body>


class child(parent):
    <body>
```

An instance of child has now automatically access to `parent_attribute` and `parent_method()`.

```
In [60]: class Person:
             def __init__(self, name, yob):
                 self.name, self.yob = name, yob
             def get_info(self):
                 print(self.name,"was born in",self.yob)
```

```
In [61]: class Employee(Person):
             pass
```

```
In [62]: JC = Employee("John Coltrane",1967)
         JC.get_info()
```

```
John Coltrane was born in 1967
```

```
In [84]: class Employee(Person):
             def __init__(self, name, yob, div, pay):
                 super().__init__(name, yob)
                 self.div, self.pay = div, pay
                 self.email = name.replace(' ','.').lower() + '@ecb.europa.eu'
```

```
In [100]: JC = Employee("John Coltrane",1967,'SRF', 60000)
          print(JC.email)
```

```
john.coltrane@ecb.europa.eu
```

Polymorphism: overriding inherited methods

In [63]:

```
class Employee(Person):
    def __init__(self, name, yob, div, pay):
        super().__init__(name, yob)
        self.div, self.pay = div, pay
        self.email = name.replace(' ','.').lower() + '@ecb.europa.eu'
    def get_info(self):
        print(self.name,"born in",self.yob)
        print("Emal: ",self.email)
        print("Pay: ",self.pay,'€')
        print("Role: "+ type(self).__name__)
```

In [64]:

```
JC = Employee("John Coltrane",1967,'SRF', 60000)
JC.get_info()
```

```
John Coltrane born in 1967
Emal:  john.coltrane@ecb.europa.eu
Pay:  60000 €
Role: Employee
```

Of course, an object can create other instances:

In [65]:
```
class Manager(Employee):
    @staticmethod
    def hire(N):
        New_employees = []
        for i in range(N):
            New_employees.append(Employee("new_empl"+str(i),1990,'SRF', 40000))
        return New_employees
    @classmethod
    def promote(cls,Employee):
        new_manager = cls(Employee.name, Employee.yob, Employee.div, Employee.pay)
        new_manager.pay *= 1.4
        del Employee
        return new_manager
```

In [66]:
```
a_manager = Manager("Mister Manager", 1976, 'SRF', 90000)
JC_promoted = a_manager.promote(JC)
JC_promoted.get_info()
```

```
John Coltrane born in 1967
Emal:  john.coltrane@ecb.europa.eu
Pay:  84000.0 €
Role: Manager
```

In [68]:
```
print(isinstance(JC_promoted, Manager))
print(isinstance(JC, Manager))
```

```
True
False
```

In [69]:

```
print(issubclass(Manager, Employee))
print(issubclass(Person, Employee))
```

```
True
False
```

End of Lecture 3