Object-oriented programming in Python

0. Motivation for object-oriented approach

0.1 Combining data and functions

```
In [1]: student1_name = 'Bob'
        student2_name = 'Sarah'

        student1_age = 25
        student2_age = 26
```

```
In [2]: def printStudent(name,age):
            print '%s is %u years old' % (name, age)
```

```
In [3]: printStudent(student1_name, student1_age)
        printStudent(student2_name, student2_age)
```

```
Bob is 25 years old
Sarah is 26 years old
```

Adding more students becomes more and more cumbersome. One way out: using arrays:

```
In [4]: name = ['Bob', 'Sarah', 'Joe']
        age  = [25, 26, 27]
```

```
In [5]: def printAllStudents():
            for i in range(len(name)):
                printStudent(name[i], age[i])
```

```
In [7]: printAllStudents()
```

```
Bob is 25 years old
Sarah is 26 years old
Joe is 27 years old
```

Still a big annoying: adding more attributes to a student requires a change of printStudent and printAllStudents

```
In [*]: program = ['Physics', 'Politics', 'Sociology']
```

In [8]:

```
def printStudent2(name, age, program):
    print '%s is %u years old' % (name, age, program)

def printAllStudents2():
    for i in range(len(name)):
        printStudent(name[i], age[i], program[i])
```

What we need is a way to combine data and functions -> Classes

Another example

Python already provides certain types, like int,float,list,dict,... But some others, which also seem fundamental, are missing. For example, there is no vector type. (Of course, one can abuse a list, but adding two lists is not the same as a vector addition.) Is there a possibility to create your own type which behaves in a definable way?

0.2 Privacy

In [9]:

```
grade = [1, 2, 6]
```

In [10]:

```
def iDontLikeMyGrade(studentID):
    grade[studentID] = 1
```

In [11]:

```
iDontLikeMyGrade(2)
print grade
```

```
[1, 2, 1]
```

Is there a way to prevent this?

0.3 Inheritance

Example: We want to create a zoo-simulator So we would like some code ('objects') to represent animals. All animals have some things in common (all have a name, all have a weight, and an age, all can move. But: moving means different things for different animals. A fish swims, a sparrow flies, a penguin walks or swims ... Is there a way to model situations like this where you want to reuse code for related objects but need different implementations for different objects while keeping the generality in the syntax

1. Classes

Blueprint for a student

In [12]:

```
class Student:
    """This is the blueprint for a student""" # docstring

    # NOTE: 'self' has to be stated explicitely
    # 'self' is like 'this' in C++/Java
    def hi(self):
        print "hi"

    # a method
    # NOTE again: the first argument of a method must be 'self'
    # In principle you could call it differently but 'self' is convention
    def get_age(self):
        return self.age

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

    # constructor
    def __init__(self,n,a):

        # attributes are defined simply by using them
        self.name = n
        self.age = a

        print 'Hi, I am student %s. Thanks for creating me.' % self.name
```

Now that we have a blueprint for a student, let's instantiate one:

In [13]:

```
bob = Student('Bob',25)
```

```
Hi, I am student Bob. Thanks for creating me.
```

Accessing an attribute:

In [14]:

```
print 'Age of Bob:', bob.age
```

```
Age of Bob: 25
```

Calling a method:

In [15]:

```
print "It's Bob's birthday today"
bob.set_age(26)
```

```
It's Bob's birthday today
```

Alternative way to call a method:

In [16]:

```
print 'Age of Bob:', Student.get_age(bob)
```

```
Age of Bob: 26
```

2. Private attributes and methods - encapsulation

```
In [17]: from datetime import datetime
```

```
In [18]: class Student:
             """This is the blueprint for a student"""

             def __init__(self,n,a):
                 # The name of the student should not be changed after instanciation
                 # therefore make it 'private' by adding '__' to name
                 self.__name = n
                 # Likewise for age
                 self.__birthyear = datetime.now().year - a

                 print 'Hi, I am student %s. Thanks for creating me.' % self.__name

                 # NOTE: Real privacy doesn't exist in python
                 # if 'bob' is an instance of Student '__name' can be accessed from
                 # outside writing
                 # >>> bob._Student__name

             # a method
             # NOTE again: the first argument of a method must be 'self'
             # In principle you could call it diffenetly but 'self' is convention
             def get_age(self):
                 return datetime.now().year - self.__birthyear

             # a special method, intended for 'pretty print' of the object
             # cf below for more special methods
             def __str__(self):
                 return 'I am student %s and am %u years old.' % (self.__name, self.get_age
```

Now we execute the same code as above:

```
In [19]: bob = Student('Bob', 25)
```

```
Hi, I am student Bob. Thanks for creating me.
```

N.B.: the following won't work any more because we changed the implementation

```
In [20]: print 'Age of Bob:', bob.age
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-20-9ae676b20538> in <module>()
----> 1 print 'Age of Bob:', bob.age

AttributeError: Student instance has no attribute 'age'

Age of Bob:
```

Hence: it's generally a good idea to hide (make private) the internal details of the implementation and provide access to the Class's functionality only through a defined interface

This still works:

In [21]:
```
print 'Age of Bob:', bob.get_age()
```

```
 Age of Bob: 25
```

By the way, accessing special method __str__

In [36]:
```
print bob
print str(bob)
print bob.__str__()
```

```
I am student Bob and am 25 years old.
I am student Bob and am 25 years old.
I am student Bob and am 25 years old.
```

Note also: No need to free memory, Python has automatic garbage collection and will free an instance once it's not referenced any more.

Hence: usually no need for destructor

3. Inheritance

In [42]:
```
class PhDStudent(Student):
    """A class representing a PhD student"""

    def __init__(self, n, a, gs):
        # Call parent class constructor
        Student.__init__(self, n, a)
        # Note: 'self.__init__(n,a)' not possible

        # adding a new attribute specific for a PhD student
        self.graduateSchool = gs

    # A new method
    def teach(self):
        print 'blablabla...'

    # Override a method
    def __str__(self):
        return "I am Phd student %s and am %u years old. I am enrolled in the grac
            % (self._Student__name, self.get_age(), self.graduateSchool)
```

In [43]:
```
sarah = PhDStudent('Sarah', 26, 'GGNB')
print sarah
print bob
```

```
Hi, I am student Sarah. Thanks for creating me.
I am Phd student Sarah and am 26 years old. I am enrolled in the graduate
school GGNB
I am student Bob and am 25 years old.
```

In [44]:
```
sarah.teach()
```

```
blablabla...
```

```
In [45]: bob.teach()

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-45-c29feb627667> in <module>()
----> 1 bob.teach()

AttributeError: Student instance has no attribute 'teach'
```

4. More about attributes

4.1 Available attributes

automatically generated attributes

```
In [46]: sarah.__class__

Out[46]: __main__.PhDStudent
```

```
In [51]: if isinstance(sarah,int):
             print "What? Sarah is a number? You must be kidding."
         else:
             print "Sarah is no number. I knew it!"

         Sarah is no number. I knew it!
```

Get class name as string

```
In [52]: sarah.__class__.__name__

Out[52]: 'PhDStudent'
```

Get attribute name as string

```
In [53]: sarah.get_age.__name__

Out[53]: 'get_age'
```

Get doc-string

```
In [54]: Student.__doc__

Out[54]: 'This is the blueprint for a student'
```

Dictionary of attributes

```
In [56]: sarah.__dict__

Out[56]: {'_Student__birthyear': 1986,
          '_Student__name': 'Sarah',
          'graduateSchool': 'GGNB'}
```

Get list of all methods and attributes

```
In [58]: dir(sarah)

Out[58]: ['_Student__birthyear',
          '_Student__name',
          '__doc__',
          '__init__',
          '__module__',
          '__str__',
          'get_age',
          'graduateSchool',
          'teach']
```

Checking if attribute exists:

```
In [62]: print hasattr(sarah,'teach')
         print hasattr(sarah,'getBirthday')

         True
         False
```

4.2 Accessing attributes using strings of their names

Sometimes, the name of an attribute is known only at runtime. In that case you can access it by:

```
In [65]: getattr(sarah,'graduateSchool')

Out[65]: <bound method PhDStudent.teach of <__main__.PhDStudent instance at 0x15cb878>>
```

```
In [67]: getattr(sarah,'teach')

Out[67]: <bound method PhDStudent.teach of <__main__.PhDStudent instance at 0x15cb878>>
```

```
In [68]: getattr(sarah,'teach')()

         blablabla...
```

```
In [69]: getattr(sarah,'getBirthday')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-69-302d15d97cfd> in <module>()
----> 1 getattr(sarah,'getBirthday')

AttributeError: PhDStudent instance has no attribute 'getBirthday'
```

4.3 Dynamically adding attributes

```
In [70]: dir(sarah)
```

```
Out[70]: ['_Student__birthyear',
          '_Student__name',
          '__doc__',
          '__init__',
          '__module__',
          '__str__',
          'get_age',
          'graduateSchool',
          'teach']
```

```
In [71]: sarah.onVacation = True
```

```
In [72]: dir(sarah)
```

```
Out[72]: ['_Student__birthyear',
          '_Student__name',
          '__doc__',
          '__init__',
          '__module__',
          '__str__',
          'get_age',
          'graduateSchool',
          'onVacation',
          'teach']
```

4.4 Class attributes (static attributes)

In [73]:

```
class counter:
    # class attribute / static attribute
    # NOTE: no 'self.'
    overall_total = 0

    def __init__(self):
        # data attribute
        self.my_total = 0

    def increment(self):
        counter.overall_total += 1
        self.my_total += 1

    def __str__(self):
        return 'total=%u\toverall_total=%u' % (self.my_total, counter.overall_tota
```

In [76]:

```
a = counter()
b = counter()
```

In [79]:

```
a.increment()
print a
print b
```

```
total=3 overall_total=3
total=0 overall_total=3
```

In [80]:

```
b.increment()
print a
print b
```

```
total=3 overall_total=4
total=1 overall_total=4
```

5. Special methods

We already encountered __init__ and __str__. But there are more like:

- __cmp__: defines how comparison operators (<,<=,==,...) work
- __len__: defines result of 'len(object)'
- __add__: defines how to add objects
- and many more ...

Example:

In [91]:
```
class Student:
    """This is the blueprint for a student"""

    def __init__(self,n,a):
        self.__name = n
        self.__birthyear = datetime.now().year - a

        self.__cmpAttr = '_Student__name'

        print 'Hi, I am student %s. Thanks for creating me.' % self.__name

    def get_age(self):
        return datetime.now().year - self.__birthyear

    def __str__(self):
        return 'I am student %s and am %u years old.' % (self.__name, self.get_age

    def compareBirthyear(self):
        self.__cmpAttr = '_Student__birthyear'

    def __cmp__(self,rhs):
        lval = getattr(self,self.__cmpAttr)
        rval = getattr(rhs,self.__cmpAttr)

        if lval<rval:
            return -1
        elif lval==rval:
            return 0
        else:
            return 1
```

In [92]:
```
bob = Student('bob', 25)
joe = Student('joe', 27)
print bob<joe
```

```
Hi, I am student bob. Thanks for creating me.
Hi, I am student joe. Thanks for creating me.
True
```

In [93]:
```
bob.compareBirthyear()
bob<joe
```

Out[93]:
```
False
```

Summary of most important points

Distinguish between *class* and *instance*

A *class* is a template or blueprint, it defines all the specifications an object shall have, but it does not create the object. To define a class, start a block with the "class" keyword, and replace the function "doSomething" with all the functions it should provide

In [13]:

```
class MyClass:
    def doSomething(self):
        self.myVariable = 3.1
```

An *instance* of a class, on the other hand, "lives", it has been assembled according to the specifications in the class definition. To create an instance of a class, write:

In [10]:

```
instance = MyClass()
```

Here, "instance" will be a variable of type MyClass. (Replace it by your favourite name)

Attributes

The syntax to define a "method" (=function in a class) is the same as for a "normal" (=outside a class) function, except that the first argument always is "self". Hence the prototype is:

def functionName(self,remainingArgument1,remainingArgument2,...):

For example:

In [15]:

```
class MyClass:
    def doSomething(self):
        self.myVariable=3.1
    def doSomethingElse(self,anArgument):
        self.myVariable=anArgument
```

Likewise, when defining or using *data attributes* **within the class block** you have to prefix it by "self." (self-dot). Global variables get no prefix, class/static variables need the prefix "MyClass." (Classname-dot)

A *data attribute* is a property/variable of a particular instance of the class. This means that all instances of the class have this property, but, in general, the value of the property is different for each instance. For example, a class "Car" could have a property colour; all Cars have a color, but which exactly differs from one to the other.

A *class/static variable* is a property that all instances of the class share *with the same value*. For example: all Cars have a certain number of wheels, let's call it "numberOfWheels". And this number is the same for all (functioning and sufficiently "normal") cars: numberOfWheels==4. A class/static variable is defined inside the class block but outside any function. It is accessed using the notation: Classname.staticVariable

In [18]:

```
class Car:
    numberOfWheels=4
    def repair(self):
        if Car.numberOfWheels!=4:
            print "Something's terribly wrong"
```

A class can provide a (virtually) arbitrary number of *attributes* (functions and data). In the above class definition, "doSomething" is an example of a function attribute, "myVariable" is an example of a variable to store data. A (living) instance of a class can access its attributes using dot-notation:

In [17]:
```
instance.doSomething()
instance.myVariable=2.7
instance.doSomethingElse(0)
```

Note:

- the similarity to access attributes:
    - *within the class block*: self.attribute
    - *outside*: instance.attribute
- when calling a method, you have to specify all arguments except the "self"

Special attributes

The constructor

Often a class possesses data attributes that can, in principle, assume several values; an instance would work with all of these - as long as they are indeed specified and within a predefined range. For example:

- The colour of a car
- The number of dimensions of a vector
- The membrane capacity of a neuron

To make sure that there cannot exist at any time an instance of a class where these attributes have not (or even not yet) been specified, use a *constructor*. It is a normal method but has the special name "__init__"

In [20]:
```
class Vector2D:
    def __init__(self):
        self.dim = 2
```

As before for the MyClass example, to create a Vector2D instance, write

In [23]:
```
vec = Vector2D()
```

It is also possible to give initial values as arguments to the constructor:

In [ ]:
```
class Neuron:
    def __init__(self,cm):
        self.cm = cm
    # remaining functions
```

In such cases, when the constructor has more arguments than the obligatory "self", to create an instance of that class write:

In [24]:
```
n = Neuron(1.1)
```

That is:

```
instanceName = Classname(all_arguments_of_constructor_except_self)
```

References

- Python course given in spring 2008 at the University of Pennsylvania.
- H. P. Langtangen, *Python Scripting for Computational Science* (Texts in Computational Science and Engineering), 1st ed. Springer, 2004.
- B. Stroustrup, *What is object-oriented programming?*, Software, IEEE, vol. 5, no. 3, pp. 10–20, 1988.