A not so short introduction to Python: part IV

Luca Schenato

Research Institute for Hydrogeological Protection
Italian National Research Council
(CNR-IRPI)

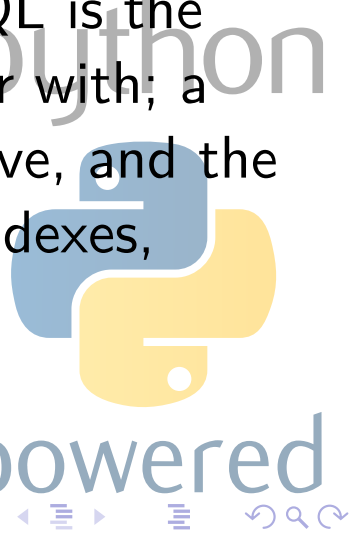
04/21/2011

Programming approaches

Approaches to problem decomposition (i.e. algorithm/programming language):

Procedural: programs are lists of instructions that tell the computer what to do with the program's input (C, Pascal, and even Unix shells are procedural languages).

Declarative: you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. (SQL is the declarative language you're most likely to be familiar with; a SQL query describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc).



Programming approaches

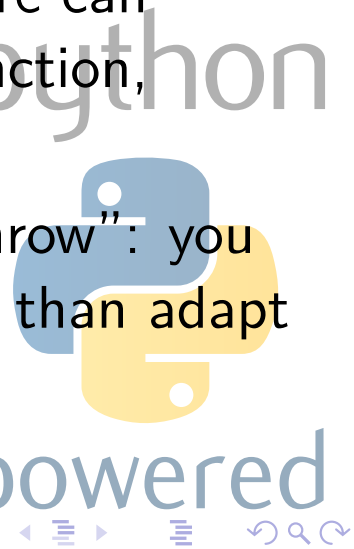
Object-Oriented: programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. (Smalltalk and Java are object-oriented languages. C++ and Python are languages that support object-oriented programming, but don't force the use of object-oriented features).

Functional: problems are decomposed into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input. (Well-known functional languages include the ML family – Standard ML, OCaml, and other variants – and Haskell).

Are you a “Use and throw” programmer?

Pure Functional programming, that means without any “side effects”, could be cumbersome. On the contrary a smoothed functional programming can be the right choice for you if you are an occasional programmer. If you are more than an occasional programmer, even functional programming lacks about the following aspects:

- due to the strict relationship between functions and data, at some point, any modification to your software can originate a domino effect on other modules/function, requiring a strong debug effort;
- your software has to be considered “Use and throw”: you better start writing a new piece of code, rather than adapt an existing one.



Why being OO and not just Functional programmer?

OO programming is an alternative way of decomposing your algorithm: accordingly, elementary unit of decomposition is any longer the function (i.e. operation), but the object (i.e. model of a real entity). This approach introduces a revolutionary programming concept: the algorithm is a set of interacting objects, each with its own data structure and allowed functions on that structure. Each object **encapsulate** the data preserving its integrity from external framework modifications. At the same time, the object can be "queried" by anyone that know its data structure and functions, without any knowledge about the internal implemetation.

OOP: a real world example

A building

Its own characteristics are: dimensions, number of room, destination, occupancy ...
Its peculiar methods (i.e. operation to be applied to): cleaning, painting, going inside, going outside ...

OOP is shared programming

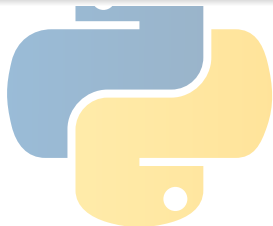
OOP is intrinsically group progamming: each programmer can implement an object that interact with other objects by others programmers, without knowing how they are implemented internally.

OOP: basics

Object and class

An object is defined by a class to which it belongs. A class specifies the general characteristics of the object, by declaring:

- the variables encapsulted within the object, called fields (also called data members or member variables);
- the encapsulated functions, called methods, that allow to operate only onto the fields of the same object.



OOP: how to approach a problem

Let's try and build a database of single persons.

Class "person"

Fields of the class:		**Methods of the class:**
name		
surname		
address	$\leftrightsquigarrow$	change address
phone	$\leftrightsquigarrow$	change phone
marriage status	$\leftrightsquigarrow$	change status

Once the class is defined, we are allowed to defined as many objects as we want belonging to that class. This objects are called *instances* of the class. Examples are:

- Luca Schenato
- Elvis Presley

...

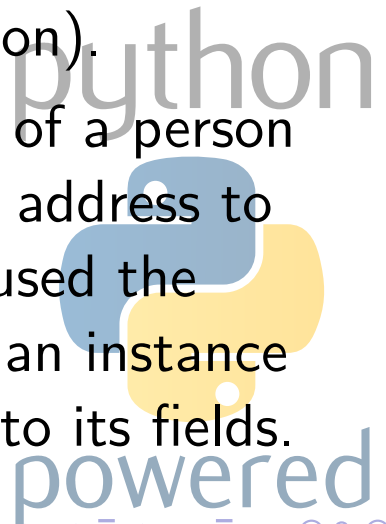
OOP: how to approach a problem

By defining a class you can also implement the so-called *encapsulation*: this term is used to refer to one of two related but distinct notions, and sometimes to the combination[1] thereof:

- A language mechanism for restricting access to some of the object's components (i.e. information restriction).
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data (i.e. not explosing internal implementation).

As an example, if you want to change the address of a person of the class "person", you should't assign the new address to that person instance but you should be forced to used the associated method. As well, if you want to define an instance of person you shouldn't be able to access directly to its fields.

[1] Wikipedia



OOP: Information hiding and class hierarchy and inheritance

Information hiding

Methods and fields of a class object can be public or private: the first ones define the public interface of the class, the others the private interface. By masquerading an encapsulated field and/or method, you make them not visible outside the scope of the object itself.

Hierarchy and Inheritance

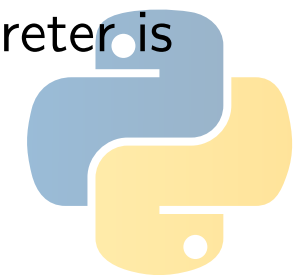
A class can be defined starting from the definition of an existing class (called *parent*) and implementing a more specific object definition (single hierarchy) or a combination of object definitions (multiple hierarchy). Please note that parent methods can be overloaded in children object (polymorfism).

Class Definition

The syntax follows:

```
class class_name [(parent_class,...)]:
field1
field2
method1
method2
```

Fields are assigned as variables, possibly at the moment of object definition. Methods of a class are defined as functions with the only difference that the first parameter of each method is the object itself `self`, so that the interpreter is aware about the calling object.



Class person

Back to our example:

```
class person:
    name = ''
    surname = ''
    address = ''
    phone = ''
    status= ''
    def change_address(self,s):
        self.address = s
    def change_phone(self,s):
        self.phone = s
    def change_status(self,s):
        self.status = s
    def display(self):
        print self.name, self.surname, self.address,
              self.status
```

Class student

And this is how inheritance work:

```
class student (person): # inheritance
    institute = ''
    year = 0
    def change_institute(self,s):
        self.institute = s
    def got_graduated(self):
        if self.year == 5:
            print 'You get graduated'
        else:
            self.year = self.year + 1
```

Object definition

The object definition syntax is similar to the function calling syntax, whereas fields and method are called by "point"-like syntax:

```
>>> p1 = person() #parenthesis are mandatory
>>> p1.name = 'Luca'
>>> p1.surname = 'Schenato'
>>> p1.change_address('Corso Stati Uniti, 4\nPadova')
>>> p1.display()
Luca Schenato Corso Stati Uniti, 4
Padova
```

Inheritance works like this:

```
>>> s1 = student()
>>> s1.name = 'Orazio'
>>> s1.surname = 'The Cat'
>>> s1.change_address('Catlandia')
>>> s1.change_institute('Mousekeepers Inst.')
>>> s1.display()
Orazio The Cat Catlandia
```

Polimorfism

let's try to overload the `display` method:

```
class student (person): # inheritance
    institute = ''
    year = 0
    def change_institute(self,s):
        self.institute = s
    def got_graduated(self):
        if self.year == 5:
            print 'You get graduated'
        else:
            self.year = self.year + 1
    def display(self):
        print self.name, self.surname, self.address,
              self.status
     print 'Institute: ' + self.institute + ' year ' +
           str(self.year)
```

Now, if you ask for displaying information about Orazio The Cat, you will have:

```
>>> s1.display()
Orazio The Cat Catlandia
Institute: Mousekeepers Inst. year 0
```

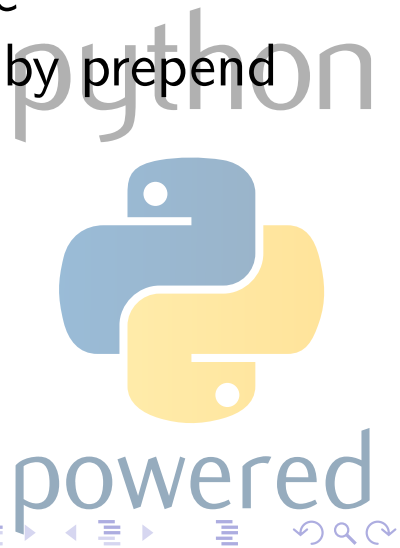
Object initialization

It is recommended to get fields initialized at the definition of the object itself. In order to do so, you can use the build function `__init__()`: this method is called whenever the object is declared, and it does assign the fields to the passed values. Example:

```
class person:
#     name = ''  #not necessary anymore
#     surname = '' #not necessary anymore
    address = ''
    phone = ''
    status= ''
    def __init__(self,n,s):
        self.name = n
        self.surname = s
    def change_address(self,s):
        self.address = s
    def change_phone(self,s):
        self.phone = s
    def change_status(self,s):
        self.status = s
    def display(self):
        print self.name, self.surname, self.address,
              self.status
```

What about information hiding/restriction?

You have been said that "*[. . .] if you want to define an instance of person you shouldn't be able to access directly to its fields.[. . .]*", but we have done that, few slides above...
The Python interpreter, by default, assumes that any fields are public, hence they can be modify directly. To make them private, so to protect them, have to implement the encapsulating paradigm: this can be simply done, by prepend to their name the characters `__`.



Encapsulation

In our example, it is advisable to make all the fields private:

```
class person:
    __address = ''
    __phone = ''
    __status= ''
    def __init__(self,n,s):
        self.__name = n
        self.__surname = s
    def change_address(self,s):
        self.__address = s
    def change_phone(self,s):
        self.__phone = s
    def change_status(self,s):
        self.__status = s
    def display(self):
        print self.__name, self.__surname,
              self.__address, self.__status
```

Encapsulation

If you try:

```
>>> p1 = person('Luca','Schenato')
>>> p1.display()
Luca Schenato
>>> p1.__nome = 'pippo'
>>> p1.display()
Luca Schenato
>>> p1.nome = 'pippo'
>>> p1.display()
Luca Schenato
```

Operators overloading

In this example we have define a complex number class (how to kill your time. . .):

```
class comp_num:
    def __init__(self,r,i):
        self.__real_part = r
        self.__imag_part = i
    def real_part(self):
        return self.__real_part
    def imag_part(self):
        return self.__imag_part
    def sum(self,num):
        self.__real_part = self.__real_part +
             num.real_part()
        self.__imag_part = self.__imag_part +
             num.imag_part()
    def display(self):
        print str(self.__real_part) + '+' +
             str(self.__imag_part) + 'i'
```

Operators overloading

And we can use it in the following way:

```
>>> n1 = comp_num(1,2)
>>> n2 = comp_num(3,4)
>>> n1.display()
1+2i
>>> n1.real_part()
1
>>> n1.sum(n2)
>>> n1.display()
4+6i
```

Operators overloading

It would be better to use the "+" operator and Python allows for it:

```
class comp_num:
    def __init__(self,r,i):
        self.__real_part = r
        self.__imag_part = i
    def real_part(self):
        return self.__real_part
    def imag_part(self):
        return self.__imag_part
    def __add__(self,num):
        ris = comp_num(self.__real_part +
              num.real_part(), self.__imag_part +
              num.imag_part())
        return ris
    def display(self):
        print str(self.__real_part) + '+' +
              str(self.__imag_part) + 'i'
```

Operators overloading

And here is how to use it:

```
>>> n1 = comp_num(1,2)
>>> n2 = comp_num(3,4)
>>> r = n1+n2
>>> r.display()
4+6i
```

Credit

Credit goes to www.python.org and programmazione.html.it and herein contents.

Acknoledgements