Pylos_tutorial

January 15, 2016

1 Pylos : an extension Object System for Python

`Pylos` is a simple (one file!) yet useful extension of the Python object system to integrate **generic methods** as in CLOS (the Common Lisp Object System).

This document is a tutorial motivating and showing the basic usage of `Pylos`.

1.1 1. The (untyped) expression problem illustrated

If like me you have an experience in object-oriented programming (classical, imperative obviously since we are talking in and about Python) as well as in functional programming (in a language like Lisp, Ocaml, Haskell), you probably noticed a tension between :

- adding new data-structures
- addning new operations working on these data-structures

This tension can be illustrated directly in Python, a language supporting both programming styles.

Our case study is a slightly boring example that everyone understands: the geometrical shapes.

1.1.1 The object-oriented programming style

A geometrical shapee in the OO style is an abstract class, e.g. as follows:

```
In [1]: class Shape:
            def __init_(self):
                pass

            def perimeter(self):
                raise NotImplementedError()
```

An example of a shape is a rectangle.

```
In [2]: class Rectangle(Shape):
            def __init__(self, x, y, w, h):
                self.x = x; self.y = y; self.w = w; self.h = h

            def perimeter(self):
                return 2 * (self.w + self.h)

            def __repr__(self):
                return "Rectangle(x={}, y={}, w={}, h={})".format(self.x, self.y, self.w, self.h)
```

Everything's straightforward and we can already "play" with rectangles.

```
In [3]: rect = Rectangle(0, 0, 3, 2)
        rect
```

```
Out[3]: Rectangle(x=0, y=0, w=3, h=2)
```

```
In [4]: rect.perimeter()
```

```
Out[4]: 10
```

Adding a new kind of shape is very natural in the OO style.

```
In [5]: import math

        class Circle(Shape):
            def __init__(self, x, y, r):
                self.x = x; self.y = y; self.r = r

            def perimeter(self):
                return math.pi * 2 * self.r;

            def __repr__(self):
                return "Circle(x={}, y={}, r={})".format(self.x, self.y, self.r)
```

```
In [6]: circ = Circle(1, 1, 1)
        circ
```

```
Out[6]: Circle(x=1, y=1, r=1)
```

```
In [7]: circ.perimeter()
```

```
Out[7]: 6.283185307179586
```

The problem arises when one wants to add a new operation to the shapes. For example suppose we want to add a translation for shapes. There's basically only one way to "solve" this problem: add the operation to the base class and all it descendants!

Thanksfully, Python is a dynamic language, allowing the so-called duck-typing, i.e. adding class (or even instance) methods at runtime.

```
In [8]: def Shape_translate(self, tx, ty):
            raise NotImplementedError()
        Shape.translate = Shape_translate
```

```
In [9]: def Point_translate(self, tx, ty):
            self.x += tx
            self.y += ty
        Rectangle.translate = Point_translate
```

```
In [10]: Circle.translate = Point_translate
```

```
In [11]: rect
```

```
Out[11]: Rectangle(x=0, y=0, w=3, h=2)
```

```
In [12]: rect.translate(2, 2)
```

```
In [13]: rect
```

```
Out[13]: Rectangle(x=2, y=2, w=3, h=2)
```

```
In [14]: circ

Out[14]: Circle(x=1, y=1, r=1)

In [15]: circ.translate(-1, -1)

In [16]: circ

Out[16]: Circle(x=0, y=0, r=1)
```

However, Python code that "opens-up" classes feels a little bit smelly (more so than in e.g. Ruby). It is intrusive to say the least.

1.1.2 The functional way

In the functional way, e.g. using a functional programming language such as Lisp, ML or Haskell, it is very easy to add new operations. The "function-first" is not the default style in Python but it is a possibility, which is great ! Let's add a surface computation operation for example, but this time not as a method but as a function.

```
In [17]: def surface_of_shape(shape):
             if isinstance(shape, Rectangle):
                 return shape.w * shape.h
             elif isinstance(shape, Circle):
                 return math.pi * shape.r * shape.r
             else:
                 raise ValueError("Cannot compute surface of {}".format(shape))

In [18]: surface_of_shape(rect)

Out[18]: 6

In [19]: surface_of_shape(circ)

Out[19]: 3.141592653589793
```

This time, the real pain is when a new kind of shape must be added, which requires to change all the existing operations (those defined as functions).

1.1.3 A solution in Pylos

There are many solutions to the (untyped) expression problem, which we will not enumerate. But there is at least on language in which the tension between adding structures vs. adding operations never occur: the Common Lisp Object System and of course Pylos that gets most of its inspiration from it.

So let's first import the (single-file) Pylos module.

```
In [20]: import sys
         sys.path.append("../src")  # point to where the pylos.py file is

In [21]: from pylos import generic, method
```

The objectif is to unify the concepts of :

- instance method in the OO style
- and function in the functional style

For this is introduced the notion of a **generic method** (a.k.a. generic function). A generic method is at the same time :

- a method because it involves a dispatch mechanism
- a function because it is defined and is called at the top-level, without a notion of a receiver.

Let's first declare the generic method. In Pylos this is a Python `def` decorated with the `generic` decorator, and with (by default) an empty body (using e.g. `pass`). The docstring is highly recommended here.

```
In [22]: @generic
         def surface():
             """Compute the surface of a shape."""
             pass
```

From a Python perspective, `surface` looks like a normal "function" (however with an empty body !).

```
In [23]: help(surface)
```

```
Help on GenericMethod in module pylos object:

class GenericMethod(Generic)
 |  Generic method:
 |  Compute the surface of a shape.
 |
 |  Method resolution order:
 |      GenericMethod
 |      Generic
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *args, **kwargs)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Generic:
 |
 |  __call__(self, *args, **kwargs)
 |
 |  wrap(self, func)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Generic:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

We will now define two specializations of the generic method for computing the surface of a `Rectangle` and a `Circle`.

```
In [24]: @method(surface)
         def _(rect : Rectangle):
             return rect.w * rect.h
```

```
In [25]: @method(surface)
         def _(circ : Circle):
             return math.pi * circ.r * circ.r
```

Note that we use the new annotation feature introduced in Python 3.4, which IMHO makes for an elegant integration of pylos in the language.

Now we can use the two specializations.

```
In [26]: surface(rect)
```

```
Out[26]: 6
```

```
In [27]: surface(circ)
```

```
Out[27]: 3.141592653589793
```

Of course, if we call the generic method with arguments for which no specialization exist, an error is raised.

```
In [28]: surface(12)
```

```
        ---------------------------------------------------------------------------

        GenericError                              Traceback (most recent call last)

        <ipython-input-28-106689c86ede> in <module>()
    ----> 1 surface(12)


        /home/pesch/Projets/pylos/src/pylos.py in __call__(self, *args, **kwargs)
        154                 adispatch = ndispatch
        155             else:
    --> 156                 raise GenericError("Cannot dispatch on argument: {}".format(arg))
        157
        158         if adispatch.dispatch_func:


        GenericError: Cannot dispatch on argument: 12
```

This gives us the opportunity to make an instance-based specialization, i.e. a specific call for a particular value.

```
In [29]: @method(surface)
         def _(twelve : 12):
             return 12
```

```
In [30]: surface(12)
```

```
Out[30]: 12
```

An important restriction here is that the value used for the specialization must be **immutable** (i.e. hashable), and the equality operator == is used for the comparison.

It is also possible to provide a default specialization.

```
In [31]: @method(surface)
         def _(anything):
             print("I don't now the surface of {}".format(anything))
             print("... but let's say it's zero.")
             return 0
```

```
In [32]: surface(42)
```

```
I don’t now the surface of 42
... but let’s say it’s zero.
```

```
Out[32]: 0
```

Of course, default specialization may be dangerous and used with care. In most cases relying on the `GenericError` exception is the way to go.

TODO : removing specializations

1.2 2. Binary (ternary . . .) methods

The second case study for `Pylos` is for what is sometimes called the binary methods problem. However, the problem does not only pop-up for methods of arity 2.

[[**TODO**]]

```
In [ ]:
```