Object-Oriented Programming in Python

Classes, Inheritance & Polymorphism

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Object-Oriented Programming

Introduction

Python is a multi-paradigm programming language.

Many claims that:

Python is object-oriented

Python is just object-based but we can use it as if it is object-oriented.

Look at

Reference

Peter Wagner.
Dimensions of Object-Based Language Design.
In Proceedings of OOPSLA'87, pp. 168–182, October 1987.

for the differences.

Object-Oriented Programming

Wagner's OO Taxonomy: Objects, Classes and Inheritance.

Objects

An object has a set of operations and a state that remembers the effect of the operations.

Class

A class is a template from which objects may be created.

- objects of the same class have common operations and (therefore) uniform behavior.
- Classes expose a set of operations (public interface) to its clients.

Inheritance

A class may inherit operations from superclasses and its operations inherited by subclasses.

- inheritance can be single or multiple.

Object-Oriented Programming

Wagner's OO Taxonomy (Cont.'d).

Wagner suggests 3 classes for programming languages:

- object-based = objects
- class-based = objects + classes
- object-oriented = objects + classes + inheritance

Data Abstraction.

A data abstraction is an object whose state is accessible only through its operations.

- this concept brings forth to the data hiding property.

Delegation.

Delegation is a mechanism to delegate responsibility for performing an operation to one or more designed ancestors.

- note that ancestors are not always designed by inheritance in this case it is called clientship.

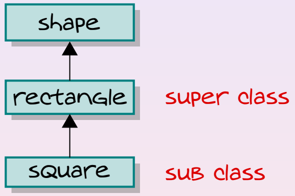
Object-Oriented Programming
Class Definition: Rectangle

```
class rectangle:
   def __init__(self, width, height):
      self._width=width
      self._height=height
   def calculate_area(self):
      return self._width*self._height
   def calculate_perimeter(self):
      return 2*(self._height+self._width)
   def __str__(self):
      return "I'm a Rectangle! My sides are: {0}, {1}\nMy area is {2}".\
              format(self._width,self._height, self.calculate_area())
```

```
[13:08]cazzola@hymir:~/esercizi-pa>python3
>>> from rectangle import rectangle
>>> r = rectangle(7,42)
>>> print(r)
I'm a Rectangle! My sides are: 7, 42
My area is 294
```

Object-Oriented Programming
Inheritance

Inheritance permits to reuse and specialize a class.



```
class shape:
   def calculate_area(self): pass
   def calculate_perimeter(self): pass
   def __str__(self): pass
```

```
from rectangle import rectangle

class square(rectangle):
   def __init__(self, width):
      self._width=width
      self._height=width
   def __str__(self):
      return \
        "I'm a Square! My side is: {0}\n \
           My area is {1}".format( \
             self._width, self.calculate_area())
```

A square is a rectangle that is a shape

Object-Oriented Programming
Inheritance & Polymorphism

```
[22:24]cazzola@hymir:~/esercizi-pa>python3
>>> from rectangle import rectangle
>>> from square import square
>>> from circle import circle
>>> shapes = [square(7), circle(3.14), rectangle(6,7), square(5),
    circle(.7), rectangle(7,2), square(2)]
>>> shapes
[<square.square object at 0x80c698c>, <circle.circle object at 0x80c69ac>,
<rectangle.rectangle object at 0x80c69cc>, <square.square object at 0x80c69ec>,
<circle.circle object at 0x80c6a0c>, <rectangle.rectangle object at 0x80c6a2c>,
<square.square object at 0x80c6a4c>]
>>> for i in shapes: print(i)
...
I'm a Square! My side is: 7
My area is 49
I'm a Circle! My ray is: 3.14
My area is 30.9748469273
I'm a Rectangle! My sides are: 6, 7
My area is 42
I'm a Square! My side is: 5
My area is 25
I'm a Circle! My ray is: 0.7
My area is 1.53938040026
I'm a Rectangle! My sides are: 7, 2
My area is 14
I'm a Square! My side is: 2
My area is 4
```

Object-Oriented Programming
Inheritance & Polymorphism Duck Typing

. . . But is shape really necessary? No

```
class rectangle:
  def __init__(self, w, h):
    self._width=w
    self._height=h
  def calculate_area(self):
    return \
      self._width*self._height
  def calculate_perimeter(self):
    return \
      2*(self._height+self._width)
  def __str__(self):
    return ...
```

```
class circle:
  def __init__(self, ray):
    self._ray=ray
  def calculate_area(self):
    return self._ray**2*math.pi
  def calculate_perimeter(self):
    return 2*self._ray*math.pi
  def __str__(self):
    return ...
```

```
class square(rectangle):
  def __init__(self, width):
    self._width=width
    self._height=width
  def __str__(self):
    return ...
```

```
[22:28]cazzola@hymir:~/esercizi-pa>python3
>>> from rectangle import rectangle
>>> from square import square
>>> from circle import circle
>>> shapes = [square(7), circle(3.14), rectangle(6,7), square(5),
...        circle(.7), rectangle(7,2), square(2)]
>>> for i in shapes: print(i)
I'm a Square! My side is: 7
My area is 49
      ...
```

Duck Typing

Object-Oriented Programming

Summarizing

The meaning of class is changed

- super classes do not impose a behavior (no abstract classes or interfaces)
- super classes are used to group and reuse functionality

Late binding quite useless

- no static/dynamic type
- duck typing

Class vs instance members

- no real distinction between fields and methods
- class is just the starting point
- a member does not exist until you use it (dynamic typing)

References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.
- Mark Summerfield.
  Programming in Python 3: A Complete Introduction to the Python Language.
  Addison-Wesley, October 2009.