Object-Oriented Programming in Python
Part 2: Advance on OOP

Walter Cazzola

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

Object-Oriented Programming
Instance vs Class Attributes

```
class C:
   def __init__(self):
      self.class_attribute="a value"
   def __str__(self):
      return self.class_attribute
```

```
[15:18]cazzola@ulik:~/oop>python3
>>> from C import C
>>> c = C()
>>> print(c)
a value
>>> c.class_attribute
'a value'
>>> c1 = C()
>>> c1.instance_attribute = "another value"
>>> c1.instance_attribute
'another value'
>>> c.instance_attribute
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'instance_attribute'
>>> C.another_class_attribute = 42
>>> c1.another_class_attribute, c.another_class_attribute
(42, 42)
```

C does not describe c!

Object-Oriented Programming
Alternative Way to Access Attributes: __dict__

```
>>> c.__dict__
{'class_attribute': 'a value'}
>>> c1.__dict__
{'class_attribute': 'a value', 'instance_attribute': 'another value'}
>>> c.__dict__['class_attribute'] = 'the answer'
>>> print(c)
the answer
```

__dict__ is an attribute
- it is a dictionary thata contains the user-provided attributes
- it permits introspection

Let's dynamically change how things are printed.

```
>>> def introspect(self):
...     result=""
...     for k,v in self.__dict__.items():
...         result += k+": "+v+"\n"
...     return result
...
>>> C.__str__ = introspect
>>> print(c)
class_attribute: the answer

>>> print(c1)
class_attribute: a value
instance_attribute: another value
```


Object-Oriented Programming
What about the Methods? Bound Methods

```
>>> class D:
...     class_attribute = "a value"
...     def f(self):
...         return "a function"
...
>>> print(D.__dict__)
{'__module__': '__main__', 'f': <function f at 0x80bbb6c>,
 '__dict__': <attribute '__dict__' of 'D' objects>, 'class_attribute': 'a value',
 '__weakref__': <attribute '__weakref__' of 'D' objects>, '__doc__': None}
>>> d = D()
>>> d.class_attribute is D.__dict__['class_attribute']
True
>>> d.f is D.__dict__['f']
False
>>> d.f
<bound method D.f of <__main__.D object at 0x80c752c>>
>>> D.__dict__['f'].__get__(d,D)
<bound method D.f of <__main__.D object at 0x80c752c>>
```

Functions are not accessed through the dictionary of the class.
- they must be bound to a an instance

A bound method is a callable object that calls a function passing an instance as the first argument.

Object-Oriented Programming
Descriptors


```
class Desc(object):
   "A descriptor example that just demonstrates the protocol"
   def __get__(self, obj, cls=None):
       print("{0}.__get({1}, {2})".format(self,obj,cls))
   def __set__(self, obj, val):
       print("{0}.__set__({1}, {2})".format(self,obj,val))
   def __delete__(self, obj):
       print("{0}.__delete__({1})".format(self,obj))

class C(object):
   "A class with a single descriptor"
   d = Desc()
```

```
[15:17]cazzola@ulik:~/esercizi-pa>python3
>>> from descriptor import Desc, C
>>> cobj = C()
>>> x = cobj.d # d.__get__(cobj, C)
<descriptor.Desc object at 0x80c610c>.__get(<descriptor.C object at 0x80c3b0c>, <class 'descriptor.C'>)
>>> cobj.d = "setting a value" # d.__set__(cobj, "setting a value")
<descriptor.Desc object at 0x80c610c>.__set__(<descriptor.C object at 0x80c3b0c>, setting a value)
>>> cobj.__dict__['d'] = "try to force a value" # set it via __dict__ avoiding the descriptor
>>> x = cobj.d # this calls d.__get__(cobj, C)
<descriptor.Desc object at 0x80c610c>.__get(<descriptor.C object at 0x80c3b0c>, <class 'descriptor.C'>)
>>> del cobj.d # d.__delete__(cobj)
<descriptor.Desc object at 0x80c610c>.__delete__(<descriptor.C object at 0x80c3b0c>)
>>> x = C.d # d.__get__(None, C)
<descriptor.Desc object at 0x80c610c>.__get(None, <class 'descriptor.C'>)
>>> C.d = "setting a value on class" # replace the descriptor
```

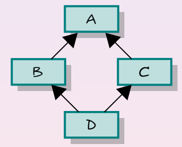

Object-Oriented Programming
Method Resolution Disorder: the Diamond Problem


```
class A(object):
  def do_your_stuff(self):
    # do stuff for A
    return

class B(A):
  def do_your_stuff(self):
    A.do_your_stuff(self)
    # do stuff for B
    return

class C(A):
  def do_your_stuff(self):
    A.do_your_stuff(self)
    # do stuff for C
    return
```

```
class D(B,C):
  def do_your_stuff(self):
    B.do_your_stuff(self)
    C.do_your_stuff(self)
    # do stuff for D
    return
```



Two copies of A

- if do_your_stuff() is called once B or C is incomplete;
- if called twice it could have undesired side-effects.

Object-Oriented Programming
A Pythonic Solution: The "Who's Next" List

The solution is to dynamically determine which do_your_stuff() to call in each do_your_stuff()

```
B.next_class_list = [B,A]
C.next_class_list = [C,A]
D.next_class_list = [D,B,C,A]

class B(A):
  def do_your_stuff(self):
    next_class = self.find_out_whos_next()
    next_class.do_your_stuff(self)
    # do stuff with self for B
  def find_out_whos_next(self):
    l = self.next_class_list # l depends on the actual instance
    mypos = l.index(B)       # Find this class in the list
    return l[mypos+1]        # Return the next one
```

find_out_whos_next() depends on which instance we are working with.

Object-Oriented Programming
__mro__ & super

There are a class attribute __mro__ for each type and a super

- __mro__ keeps the list of the superclasses without duplicates in a predictable order
- super is used in place of the find_out_whos_next()

```
class A:
  def do_stuff(self):
    print('A')

class B(A):
  def do_stuff(self):
    super(B, self).do_stuff()
    print('B')

class C(A):
  def do_stuff(self):
    super(C, self).do_stuff()
    print('C')

class D(C,B):
  def do_stuff(self):
    super(D, self).do_stuff()
    print('D')
```

Computing the method resolution order (MRO)

- if A is a superclass of B then B>A
- if C precedes D in the list of bases in a class statement then C>D
- if E>F in one scenario then E>F in all scenarios

```
[23:04]cazzola@ulik:~/esercizi-pa>python3
>>> from mro import A,B,C, D
>>> D.__mro__
(<class 'mro.D'>, <class 'mro.C'>, <class 'mro.B'>, <class 'mro.A'>, <class 'object'>)
>>> d = D()
>>> d.do_stuff()
A
B
C
D
```

Object-Oriented Programming
Special Methods

Special methods, as `__len__()`, `__str__()`, `__lt__()` and `__add__()` govern the behavior of some standard operations.

```
class C(object):
  def __len__(self):
    return 0

def mylen(): return 1
```

```
[10:03]cazzola@ulik:~/esercizi-pa>python3
>>> cobj = C()
>>> cobj.__len__ = mylen
>>> len(cobj)
0
```

Special methods are "class methods"
- they cannot be changed through the instance
- this goes straight to the type by calling `C.__len__()`

```
class C(object):
  def __len__(self): return self._mylen()
  def _mylen(self): return 0

def mylen(): return 1
```

```
[10:22]cazzola@ulik:~/esercizi-pa>python3
>>> cobj = C()
>>> cobj._mylen = mylen
>>> len(cobj)
1
```

To be more flexible
- the special method must be forwarded to a method that can be overridden in the instance

Object-Oriented Programming
__slots__

Also built-in types, as `list` and `tuple`, can be subclassed

```
class MyList(list):
  """A list that converts added items to ints"""
  def append(self, item):
    list.append(self, int(item))
  def __setitem__(self, key, item):
    list.__setitem__(self,key,int(item))
```

```
[10:45]cazzola@ulik:~/esercizi-pa>python3
>>> l = MyList()
>>> l.append(1.3)
>>> l.append(444)
>>> l
[1, 444]
>>> len(l)
2
>>> l[1] = 3.14
>>> l
[1, 3]
>>> l.color = 'red'
```

Unfortunately the subtype of `list` allow the adding of attributes
- this is due to the presence of `__dict__`

The presence of `__slots__` in a class definition inhibits the introduction of `__dicts__`
- this disallows any user-define attributes

```
class MyList2(list):
  __slots__ = []

class MyList3(list):
  __slots__ = ['color']

class MyList4(list):
  """A list that contains only ints"""
  def __init__(self, itr):
    list.__init__(self, [int(x) for x in itr])
  def append(self, item):
    list.append(self, int(item))
  def __setitem__(self, key, item):
    list.__setitem__(self,key,int(item))
```

```
[11:13]cazzola@ulik:~/esercizi-pa>python3
>>> m2 = MyList2()
>>> m2.color = 'red'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError:
   'MyList2' object has no attribute 'color'
>>> m3 = MyList3()
>>> m3.color = 'red'
>>> m3.weight = 50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError:
   'MyList3' object has no attribute 'weight'
```


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.
- Shalabh Chaturvedi.
  Python Attributes and Objects.
  2009.
  Available at http://www.cafepy.com/article/python_attributes_and_methods/.
- 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.