Python
OOP

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

October 9, 2009

Outline

Classes

Description

- A class is a data structure that we can use to define objects that hold together data values and behavorial characteristics.
- Classes are entities that are the pragmatic form of an abstraction for a real-world problem, and instances are realizations of such objects

Definition!

```
def function_name(args):
    'function documentation string'
    function_suite

class classeName(base_class):
    'class documentation string'
    class_suite
```

Classes

Simple example

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'

x = MyClass()
print "x.f():", x.f()
print "x.i:", x.i
```

```
$ python myclass.py
x.f(): hello world
x.i: 12345
$
```

Classes

Simple example

```
class MyClass:
    """A simple example class"""

    def myNoActionMethod(self):
        print 'in MyClass.myNoActyionMethod'

x = MyClass()
x.myNoActionMethod()
myNoActionMethod()
```

```
$ python myclass.py
in MyClass.myNoActionMethod
Traceback (most recent call last):
  File "myclass.py", line 9, in <module>
    myNoActyionMethod()
NameError: name 'myNoActionMethod' is not defined
$
```

Classes

Simple example

```
class MyClass:
    """A simple example class"""

    def myNoActionMethod(self):
        print 'in MyClass.myNoActyionMethod'

MyClass.myNoActionMethod()
```

```
$ python myclass.py
Traceback (most recent call last):
  File "myclass.py", line 7, in <module>
    MyClass.myNoActionMethod()
TypeError: unbound method myNoActionMethod() must be called with
MyClass instance as first argument (got nothing instead)
$
```

Determining class attributes

Example

```
class MyClass(object): # new style class
    'MyClass class definition'

    # static data
    version = '1.1'

    # method
    def show_version(self):
        'docstring for method show_version'
        print Myclass.version
```

Determining class attributes

Example

```
>>> from myclass import MyClass
>>> dir(MyClass)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
 '__getattribute__', '__hash__', '__init__', '__module__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'show_version', 'version']
>>> MyClass.__dict__
<dictproxy object at 0xa3b8164>
>>>print MyClass.__dict__
{'__module__': 'my_class_for_attributes',
 'show_version': <function show_version at 0xa358304>,
 'version': '1.1', '__dict__': <attribute '__dict__' of 'MyClass' objects>,
 '__weakref__': <attribute '__weakref__' of 'MyClass' objects>,
 '__doc__': 'MyClass class definition'}
>>> print MyClass.__dict__['__doc__']
MyClass class definition
>>>
```

Special class attributes

`C.__name__`	String name of class `C`
`C.__doc__`	Documentation string for class `C`
`C.__bases__`	Tuple of class `C`'s parent classes
`C.__dict__`	Attributes of `C`
`C.__module__`	Module where `C` is defined
`C.__class__`	Class of which `C` is an instance

Example

```
>>> MyClass.__name__
'MyClass'
>>> MyClass.__module__
'myclass'
>>> MyClass.__class__
<type 'type'>
>>> MyClass.__bases__
(<type 'object'>,)
>>>
```

Outline

`__init__` constructor method

Description

- When the class is invoked, the first step in the instantiation process is to create the instance object.
- Once the object is available, Python checks if an `__init__` method has been implemented.

Important

- You do not call `new` to create an instance.
- `__init__` is simply the first method that is called after the interpreter creates an instance.
- `__init__` should return `None`.

`__new__` constructor method

Description

- The `__new__` (static) method bears a much closer ressemblance to a real constructor.
- Python users have the ability to subclass built-in types, and so there needed to a way to instantiate immutable objects.

Important

- It is the responsability of `__new__` to create the object (delegating upward).
- `__new__` returns a valid instance so that the interpreter then call `__init__` with that instance as `self`.

`__del__` destructor method

Description

- The `__del__` method is not executed until all references to an instance object have been removed.
- Python users have the ability to subclass built-in types, and so there needed to a way to instantiate immutable objects.

Important

- In Python, destructors are methods that provide special processing before instances are deallocated.
- Destructors are not coomonly implemented.

__init__ and __del__

Example

```
In [1]: class A(object):          # class declaration
   ...:     def __init__(self):   # constructor
   ...:         print 'initialized'
   ...:     def __del__(self):    # destructor
   ...:         print 'deleted'
   ...:
In [2]: a1 = A()                  # instantiation
initialized
In [3]: a2, a3 = a1, a1           # create additional aliases
In [4]: id(a1), id(a2), id(a3)    # all refer to the same object
Out[4]: (160657548, 160657548, 160657548)
In [5]: del a1                    # delete one reference
In [6]: del a2                    # delete a second reference
In [7]: del a3                    # delete te thirs reference
deleted
In [8]:
```

Instance attributes

Example

```
from math import sqrt

class Point(object):
    'A 2D point'

    def __init__(self, x = 0, y = 0):
        'Initialize a point'
        self.x = x
        self.y = y

    def distance(self, another_point):
        'Return the euclidean distance between two points'
        diff_x = self.x - another_point.x
        diff_y = self.y - another_point.y
        return sqrt(diff_x * diff_x + diff_y * diff_y)

    def __str__(self):
        'String representation of a point'
        return 'Point(' + str(self.x) + ',' + str(self.y) + ')'
```

Instance attributes

Example

```
In [1]: from point import Point

In [2]: p1, p2 = Point(1, 2), Point(2, 5)

In [3]: type(p1), type(p2)
Out[3]: (<class 'point.Point'>, <class 'point.Point'>)

In [4]: p1, p2
Out[4]: (<point1.Point object at 0x93affec>,
         <point1.Point object at 0x932178c>)

In [5]: print p1
Point(1,2)

In [6]: print p2
Point(2,5)

In [7]: p1.distance(p2), p2.distance(p1), p1.distance(p1)
Out[7]: (3.1622776601683795, 3.1622776601683795, 0.0)
```

Outline

Creating subclasses

Example

```
In [1]: class Parent(object):
   ...:     def parent_method(self):
   ...:         print 'calling parent method'
   ...:
In [2]: class Child(Parent):
   ...:     def child_method(self):
   ...:         print 'calling child method'
   ...:
In [3]: p, c = Parent(), Child()

In [4]: p.parent_method()
calling parent method

In [5]: c.child_method()
calling child method

In [6]: c.parent_method()
calling parent method
```

Inheritance

Example

```
In [1]: class P(object):
   ...:     def __init__(self):
   ...:         print 'created an instance of', self.__class__.__name__
   ...:
In [2]: class C(P):
   ...:     pass
   ...:
In [3]: p = P()
created an instance of P

In [4]: p.__class__
Out[4]: <class '__main__.P'>

In [5]: c = C()
created an instance of C

In [6]: c.__class__
Out[6]: <class '__main__.C'>
```

Overriding methods

Example

```
In [1]: class P(object):
   ...:     def f(self):
   ...:         print 'calling P.f()'
   ...:
In [2]: class C(P):
   ...:     def f(self):
   ...:         print 'calling C.f()'
   ...:
In [3]: p, c = P(), C()

In [4]: p.f()
calling P.f()

In [5]: c.f()
calling C.f()
```

Overriding methods

Example

```
In [1]: class P(object):
   ...:     def f(self):
   ...:         print 'calling P.f()'
   ...:
In [2]: class C(P):
   ...:     def f(self):
   ...:         super(C, self).f()
   ...:         print 'calling C.f()'
   ...:
In [3]: c = C()

In [4]: c.f()
calling P.f()
calling C.f()

In [5]: P.f(c)
calling P.f()
```

Overriding methods

Example

```
In [1]: class P(object):
   ...:     def __init__(self):
   ...:         print 'calling P constructor'
   ...:
In [2]: class C(P):
   ...:     def __init__(self):
   ...:         print 'calling C constructor'
   ...:
In [3]: p = P()
calling P constructor

In [4]: c = C()
calling C constructor
```

Overriding methods

Example

```
In [1]: class P(object):
   ...:     def __init__(self):
   ...:         print 'calling P constructor'
   ...:
In [2]: class C(P):
   ...:     def __init__(self):
   ...:         super(C, self).__init__()
   ...:         print 'calling C constructor'
   ...:
In [3]: p = P()
calling P constructor

In [4]: c = C()
calling P constructor
calling C constructor
```

Deriving immutable standard type 1/2

Example

```
In [1]: class RoundFloat(float):
   ...:     'float with two decimal places'
   ...:     def __new__(cls, val):
   ...:         return super(RoundFloat, cls).__new__(cls, round(val, 2))
   ...:
In [2]: RoundFloat(1.234)
Out[2]: 1.23

In [3]: RoundFloat(1.236)
Out[3]: 1.24

In [4]: RoundFloat(-1.999)
Out[4]: -2.0
```

Deriving immutable standard type 2/2

Example

```
In [1]: class UpperString(str):
   ...:     def __new__(cls, s):
   ...:         return super(UpperString, cls).__new__(cls, s.upper())
   ...:
In [2]: UpperString('abc')
Out[2]: 'ABC'

In [3]: UpperString('aBc')
Out[3]: 'ABC'

In [4]: UpperString('ABC')
Out[4]: 'ABC'

In [5]: UpperString('aBc').lower()
Out[5]: 'abc'

In [6]: UpperString('aBc').lower().capitalize()
Out[6]: 'Abc'
```

Deriving mutable standard type

Example

```
In [1]: class SortedKeyDict(dict):
   ...:     'dic type where keys are returned sorted'
   ...:     def keys(self):
   ...:         return sorted(super(SortedKeyDict, self).keys())
   ...:

In [2]: d = SortedKeyDict((('a', 1), ('d', 4), ('b', 2), ('c', 3)))

In [3]: print 'by iterator:'.ljust(12), [key for key in d]
('by iterator:', ['a', 'c', 'b', 'd'])

In [4]: print 'by keys:'.ljust(12), [key for key in d.keys()]
('by keys:    ', ['a', 'b', 'c', 'd'])
```

Outline

Boolean expressions

Description

- An expression that results in a value of either `TRUE` or `FALSE`.
- Boolean operators:
  - `'or`,
  - `'and`,
  - `'not`,
  - `'xor`,
  - ...

Example

```
(TRUE and (not (FALSE or (TRUE nand TRUE)) xor FALSE))
```

Boolean expressions

Abstract classes

```
class BoolExpr(object):
    'Boolean expression (abstract) base class'
    pass

class BinBoolExpr(BoolExpr):
    'Binary boolean expression (abstract) base class'
    pass

class UnBoolExpr(BoolExpr):
    'Unary boolean expression (abstract) base class'
    pass
```

Boolean expressions

Concrete classes

```
class Not(UnBoolExpr):
    'Negation'
    pass

class BoolValue(BoolExpr):
    'True or False'
    pass

class And(BinBoolExpr):
    'AND operator'
    pass

class Or(BinBoolExpr):
    'OR operator'
    pass
```

Boolean expressions

Abstract classes: implementation

```
class BoolExpr(object):
    'Boolean expression (abstract) base class'
    pass

class BinBoolExpr(BoolExpr):
    'Binary boolean expression (abstract) base class'
    def __init__(self, l_bool_expr, r_bool_expr):
        'Initialize a binary boolean expression'
        self.l_bool_expr = l_bool_expr
        self.r_bool_expr = r_bool_expr

class UnBoolExpr(BoolExpr):
    'Unary boolean expression (abstract) base class'
    def __init__(self, bool_expr):
        'Initialize an unary boolean expression'
        self.bool_expr = bool_expr
```

Boolean expressions

Concrete classes: implementation

```
class Not(UnBoolExpr):
    def eval(self):
        return not self.bool_expr.eval()

class BoolValue(BoolExpr):
    def __init__(self, bool_value):
        self.val = bool_value
    def eval(self):
        return self.val

class And(BinBoolExpr):
    def eval(self):
        return self.l_bool_expr.eval() and self.r_bool_expr.eval()

class Or(BinBoolExpr):
    def eval(self):
        return self.l_bool_expr.eval() or self.r_bool_expr.eval()
```

Boolean expressions

Output

```
In [1]: from boolexpr import *

In [2]: T, F = BoolValue(True)
, BoolValue(False)

In [3]: T.eval(), F.eval()
Out[3]: (True, False)

In [4]: e = And(Not(T), Or(Or(T, F), And(F, Not(T))))
In [5]: e
Out[5]: <boolexpr_1.And object at 0x8f6250c>

In [6]: e.eval()
Out[6]: False

In [7]: print e
<boolexpr.And object at 0x8f6250c>
```

Boolean expressions

Output

```
n [1]: from boolexpr import *

In [2]: T, F = BoolValue(True), BoolValue(False)

In [3]: e = And(Not(T), Or(Or(T, F), And(F, Not(T))))

In [4]: e2 = BinBoolExpr(T, e) # BinBoolExpr should be abstract

In [5]: e2
Out[5]: <boolexpr.BinBoolExpr object at 0x992652c>

In [6]: e2.eval() # oups ...
---------------------------
AttributeError Traceback (most recent call last)

.../<ipython console> in <module>()
AttributeError: 'BinBoolExpr' object has no attribute 'eval'
```

Boolean expressions

Abstract classes: implementation

```
class BoolExpr(object):
    def __init__(self, *args):
        raise TypeError, 'BoolExpr is an abstract base class'
    def __str__(self):
        raise NotImplementedError, 'To be implemented in derived classes'

class BinBoolExpr(BoolExpr):
    def __init__(self, l_bool_expr, r_bool_expr):
        if self.__class__ is BinBoolExpr:
            raise TypeError, 'BinBoolExpr is an abstract base class'
        self.l_bool_expr = l_bool_expr
        self.r_bool_expr = r_bool_expr

class UnBoolExpr(BoolExpr):
    def __init__(self, bool_expr):
                if self.__class__ is UnBoolExpr:
            raise TypeError, 'UnBoolExpr is an abstract base class'
        self.bool_expr = bool_expr
```

Boolean expressions

Concrete classes: implementation

```
class Not(UnBoolExpr):
    def eval(self):
        return not self.bool_expr.eval()
    def __str__(self):
        return '(not %s)' % str(self.bool_expr)

class BoolValue(BoolExpr):
    def __init__(self, bool_value):
        self.val = bool_value
    def eval(self):
        return self.val
    def __str__(self):
        return str(self.val)

class And(BinBoolExpr):
    def eval(self):
        return self.l_bool_expr.eval() and self.r_bool_expr.eval()
    def __str__(self):
        return '(%s and %s)' % (str(self.l_bool_expr), str(self.r_bool_expr))
```

Boolean expressions

Concrete classes: refactoring

```
class BinBoolExpr(BoolExpr):
    def __str__(self):
        return '(%s %s %s)' % \
               (str(self.l_bool_expr), self.OP, str(self.r_bool_expr))

class And(BinBoolExpr):
    OP = 'and'
    def __str__(self):
        return '(%s and %s)' % (str(self.l_bool_expr), str(self.r_bool_expr))

class Or(BinBoolExpr):
    OP = 'or'
    def __str__(self):
        return '(%s and %s)' % (str(self.l_bool_expr), str(self.r_bool_expr))
```

Boolean expressions

Concrete classes: implementation

```
class XOr(BinBoolExpr):
    OP = 'xor'
    def eval(self):
        return (self.l_bool_expr.eval() and not self.r_bool_expr.eval() or \
                (not self.l_bool_expr.eval() and self.r_bool_expr.eval()))
class NAnd(BinBoolExpr):
    OP = 'nand'
    def eval(self):
        return not (self.l_bool_expr.eval() and self.r_bool_expr.eval())
class NOr(BinBoolExpr):
    OP = 'nor'
    def eval(self):
        return not (self.l_bool_expr.eval() or self.r_bool_expr.eval())
class XNOr(BinBoolExpr):
    OP = 'xnor'
        def eval(self):
    return not((self.l_bool_expr.eval() and not self.r_bool_expr.eval() or \
                (not self.l_bool_expr.eval() and self.r_bool_expr.eval())))
```

Boolean expressions

Concrete classes: using `Not`, `And` and `Or` gates only

```
def XOr(l_bool_expr, r_bool_expr):
    'a xor b <=> (a and not b) or (not and b)'
    return Or(And(l_bool_expr, Neg(r_bool_expr)),
              And(Neg(l_bool_expr), r_bool_expr))

def XNOr(l_bool_expr, r_bool_expr):
    'a xnor b <=> not (a xor b)'
    return Neg(XOr(l_bool_expr, r_bool_expr))

def NAnd(l_bool_expr, r_bool_expr):
    'a nand b <=> not (a and b)'
    return Neg(And(l_bool_expr, r_bool_expr))

def NOr(l_bool_expr, r_bool_expr):
    'a nor b <=> not (a or b)'
    return Neg(Or(l_bool_expr, r_bool_expr))
```

Outline

Privacy

Description

- Attribute in Python are by default public.
- Many OOP languages provide some level of privacy for the data and provide only accesso functions to provide access to the values. This is known as implementation hiding.
- Python provides an elemenary form of privacy fo class elements (attributes or methods).

Privacy: Double underscore

Example

```
class P(object):
    def __init__(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
class C(P):
    def get_name(self):
        return self.__name

p = P('aa')         # a parent instance
print p.__name      # raise AttributeError: 'P' object has no attribute '__name'
print p.get_name() # ok
c = C('bb')         # a child instance
print c.__name      # raise AttributeError: 'C' object has no attribute '__name'
print c.get_name() # raise AttributeError: 'C' object has no attribute '__name'
```

Privacy: Double underscore

Example

```
class P(object):
    def f(self):
        self.__g()
    def __g(self):
        pass
class C(P):
    pass

p = P()         # a parent instance
print p.f()     # ok
print p.__g()   # raise AttributeError: 'P' object has no attribute '__g'
c = C()         # a child instance
print c.f()     # ok
print c.__g()   # raise AttributeError: 'C' object has no attribute '__name'
```

Privacy: Single underscore

Example

```
class P(object):
    def __init__(self, name):
        self._name = name
    def get_name(self):
        return self._name
class C(P):
    def get_name(self):
        return self._name

p = P('aa')         # a parent instance
print p._name       # ok
print p.get_name() # ok
c = C('bb')         # a child instance
print c._name       # ok
print c.get_name() # ok
```

Privacy: Single underscore

Example

```
# module mymodule
class A(object):
    def f(self):
        self._g()
    def _g(self):
        pass
    def h(self):
        return _B()
class _B(object):
    pass

# in another module
from single_underscore2 import *
a = A()     # instance of class A
a.f()       # ok
a._g()      # ok
a.h()       # ok
b = _B()    # raise NameError: name '_B' is not defined
```

Privacy: Single underscore

Description

Prevent a module attribute from being imported with `from mymodule import *`.

Example

```
# module mymodule
class A(object):
    def f(self):
        self._g()
    def _g(self):
        pass
    def h(self):
        return _B()
class _B(object):
    pass

# in another module
from single_underscore2 import *
a = A()     # instance of class A
a.f()       # ok
```

__slots__ class attribute

Example

```
n [1]: class SlottedClass(object):
   ...:     __slots__ = ('att1', 'att2')
   ...:
In [2]: c = SlottedClass()

In [3]: c
Out[3]: <__main__.SlottedClass object at 0x991be8c>

In [4]: c.att1, c.att2 = 1, 2

In [5]: c.att1, c.att2
Out[5]: (1, 2)

In [6]: c.att3 = 3
---------------------------------------------------------------------------
AttributeError  Traceback (most recent call last)
.../<ipython console> in <module>()
AttributeError: 'SlottedClass' object has no attribute 'att3'
```

`__getattribute__` special method

Description

- Python classes have a special method named `__getattr__` which is called only when an attribute canot be found in an instance's `__dict__`, or its class's `__dict__`, or ancestor class (its `__dict__`).
- `__getattribute__` works just like `__getattr__` except that it is always called when an attribute is accessed.
- if a class has both `__getattribute__` and `__getattr__` defined, the latter will not be called unless explicitly called from `__getattribute__` or if `__getattribute__` raises `AttributeError`.

__getattribute__ special method

Example

```
class A(object):
    def __init__(self, x):
        self.x = x

    def f(self):
        print 'calling A.f()'

    def __getattr__(self, name):
        print 'calling A._getattr(%s)' % name
        return None

if __name__ == '__main__':
    a = A('aa')
    a.f()
    print 'a.x = %s' % a.x
    print 'a.y = %s' % a.y
    a.g()
```

__getattribute__ special method

Example

```
# a = A('aa')
# a.f()
calling A.f()
# print 'a.x = %s' % a.x
a.x = aa
# print 'a.y = %s' % a.y
calling A._getattr(y)
a.y = None
# a.g()
calling A._getattr(g)
Traceback (most recent call last):
  File "getattr.py", line 17, in <module>
    a.g()
TypeError: 'NoneType' object is not callable
```