SoftServe
Ability. Agility. Advantage.

Object Oriented Programming in Python

Vadym Shcherbakov

Agenda

1. Introduction
2. Objects, Types and Classes
   - Class Definition
   - Class Instantiation
   - Constructor and Destructor
   - Lifetime of an Object
   - Encapsulation and Access to Properties
   - Polymorphism
3. Relations between classes
   - Inheritance and Multiple Inheritance
   - "New" and "Classic" Classes
   - Metaclasses
   - Aggregation. Containers. Iterators
4. Methods
   - Methods
   - Static Methods
   - Class Methods
   - Multimethods (Multiple Dispatch)
5. Object Persistence

Introduction. It's all objects…

- Everything in Python is really an object.
  - We've seen hints of this already…

```
"hello".upper()
list3.append('a')
dict2.keys()
```

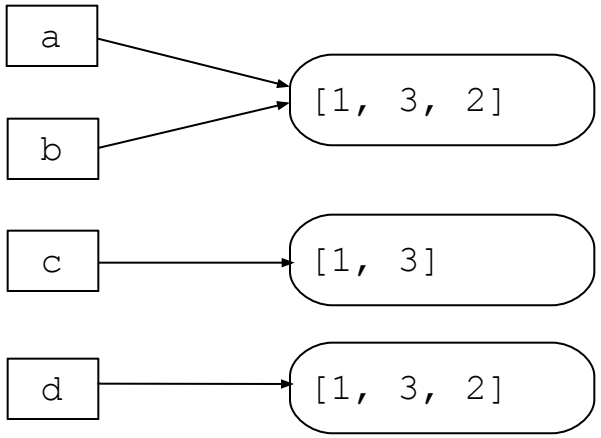
  - These look like Java or C++ method calls.
  - New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Objects, names and references

- All values are objects
- A variable is a name referencing an object
- An object may have several names referencing it
- Important when modifying objects in-place!
- You may have to make proper copies to get the effect you want
- For immutable objects (numbers, strings), this is never a problem

```
>>> a = [1, 3, 2]
>>> b = a
>>> c = b[0:2]
>>> d = b[:]
```

```
>>> b.sort()    # 'a' is affected!
>>> a
[1, 2, 3]
```



Class Definition

- For clarity, in the following discussion we consider the definition of class in terms of syntax. To determine the class, you use **`class`** operator:

```
class ClassName (superclass 1, superclass 2, ...)
    # Define attributes and methods of the class
```

- In a class can be basic (parent) classes (superclasses), which (if any) are listed in parentheses after the defined class.
- The smallest possible class definition looks like this:

```
class A:
    pass
```

Class Definition

- In the terminology of the Python members of the class are called attributes, functions of the class - methods and fields of the class - properties (or simply attributes).
- Definitions of methods are similar to the definitions of functions, but (with some exceptions, of which below) methods always have the first argument, called on the widely accepted agreement `self`:

```
class A:
    def m1 (self, x):
        # method code block
```

- Definitions of attributes - the usual assignment operators that connect some of the values with attribute names:

```
class A:
    attr1 = 2 * 2
```

Class Definition

- In Python a class is not something static after the definition, so you can add attributes and after:

```
class A:
    pass

def myMethod (self, x):
    return x * x

A.m1 = myMethod
A.attr1 = 2 * 2
```

Class Instantiation

- To instantiate a class, that is, create an instance of the class, simply call the class name and specify the constructor parameters:

```
class Point:
    def __init__ (self, x, y, z):
        self.coord = (x, y, z)
    def __repr__ (self):
        return "Point (%s, %s, %s)" % self.coord
>>> P = Point(0.0, 1.0, 0.0)
>>> P
Point (0.0, 1.0, 0.0)
```

- `__init__` is the default constructor.
- `self` refers to the object itself, like `this` in Java.

Class Instantiation

- By overriding the class method **`__new__`**, you can control the process of creating an instance of the class. This method is called before the method **`__init__`** and should return a new instance, or **`None`** (in the latter case will be called **`__new__`** of the parent class).
- Method **`__new__`** is used to control the creation of unchangeable (immutable) objects, managing the creation of objects in cases when **`__init__`** is not invoked.

Class Instantiation

The following code demonstrates one of the options for implementing Singleton pattern:

```
>>> class Singleton(object):
        obj = None # attribute for storing a single copy
        def __new__(cls, * dt, ** mp): # class Singleton.
            if cls.obj is None:
                # If it does not yet exist, then
                # call __new__ of the parent class
                cls.obj = object. __new__ (cls, *dt, **mp)
            return cls.obj # will return Singleton
 ...
 >>> obj = Singleton()
 >>> obj.Attr = 12
 >>> new_obj = Singleton()
 >>> new_obj.Attr
 12
 >>> new_obj is obj # new_obj and obj - is one and the same object
 True
```

Constructor and Destructor

- Special methods are invoked at instantiation of the class (constructor) and disposal of the class (destructor). In Python is implemented automatic memory management, so the destructor is required very often, for resources, that require an explicit release.
- The next class has a constructor and destructor:

```
class Line:
    def __init__(self, p1, p2):
        self.line = (p1, p2)
    def __del__(self):
        print "Removing Line %s - %s" % self.line
>>> L = Line((0.0, 1.0), (0.0, 2.0))
>>> del l
Removing Line (0.0, 1.0) - (0.0, 2.0)
>>>
```

Lifetime of an object

- Without using any special means lifetime of the object defined in the Python program does not go beyond of run-time process of this program.
- To overcome this limitation, there are different possibilities: from object storage in a simple database (shelve), application of ORM to the use of specialized databases with advanced features (eg, ZODB, ZEO). All these tools help make objects persistent. Typically, when write an object it is serialized, and when read - deserializated.

```
>>> import shelve
>>> s = shelve.open("somefile.db")
>>> s['myobject'] = [1, 2, 3, 4, 'candle']
>>> s.close()
>>>
>>> s = shelve.open("somefile.db")
>>> print s['myobject']
[1, 2, 3, 4, 'candle']
```

Encapsulation and access to properties

- Encapsulation is one of the key concepts of OOP. All values in Python are objects that encapsulate code (methods) & data and provide users a public interface. Methods and data of an object are accessed through its attributes.
- Hiding information about the internal structure of the object is performed in Python at the level of agreement among programmers about which attributes belong to the public class interface, and which - to its internal implementation.
- A single underscore in the beginning of the attribute name indicates that the method is not intended for use outside of class methods (or out of functions and classes of the module), but the attribute is still available by this name.
- Two underscores in the beginning of the name give somewhat greater protection: the attribute is no longer available by this name. The latter is used quite rarely.

Encapsulation and access to properties

- There is a significant difference between these attributes and personal (private) members of the class in languages like C++ or Java: attribute is still available, but under the name of the form _ClassName__AttributeName, and each time the Python will modify the name, depending on the instance of which class is handling to attribute.
- Thus, the parent and child classes can have an attribute name, for example, __f, but will not interfere with each other.

```
>>> class parent(object):
...     def __init__(self):
...         self.__f = 2
...     def get(self):
...         return self.__f
...
>>> class child(parent):
...     def __init__(self):
...         self.__f = 1
...         parent.__init__(self)
...     def cget(self):
...         return self.__f
...
>>> c = child()
>>> c.get()
2
>>> c.cget()
1
>>> c.__f
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'child' object has no attribute
'__f'
>>> c.__dict__
{'_child__f': 1, '_parent__f': 2}
>>> c._parent__f
2
```

Encapsulation and access to properties

- Access to the attribute can be either direct:

```
>>> class A(object):
...     def __init__(self, x):
...       # attribute gets the value in the constructor
...       self.x = x
...
>>> a = A(5)
>>> print a.x
5
```

- …Or using the properties with the specified methods for getting, setting and removing an attribute:

Encapsulation and access to properties

- …Or using the properties with the specified methods for getting, setting and removing an attribute:

```
>>> class A(object):
...     def __init__(self, x):
...         self._x = x
...     def getx(self): # method to obtain the value
...         return self._x
...     def setx(self, value): # assign new value
...         self._x = value
...     def delx(self): # delete attribute
...         del self._x
...     # define x as the property
...     x = property(getx, setx, delx, "property x")
...
>>> a = A(5)
>>> print a.x # syntax for accessing an attribute remains former
5
>>> a.x = 6
>>> print a.x
6
```

Encapsulation and access to properties

- There are two ways to centrally control access to attributes. The first is based on method overloading `__getattr__()`, `__setattr__()`, `__delattr__()`, and the second - the method `__getattribute__()`.
- The second method helps to manage reading of the existing attributes.
- These methods allow you to organize a fully dynamic access to the attributes of the object or that is used very often, and imitation of non-existent attributes.
- According to this principle function, for example, all of RPC for Python, imitating the methods and properties that are actually existing on the remote server.

Polymorphism

- In the compiled programming languages, polymorphism is achieved by creating virtual methods, which, unlike non-virtual can be overload in a descendant.
- In Python all methods are virtual, which is a natural consequence of allowing access at run time.

```
>>> class Parent(object):
...     def isParOrPChild(self):
...         return True
...     def who(self):
...         return 'parent'
...
>>> class Child (Parent):
...     def who (self):
...         return 'child'
...
>>> x = Parent()
>>> x.who(), x.isParOrPChild()
('parent', True)
>>> x = Child()
>>> x.who(), x.isParOrPChild()
('child', True)
```

Polymorphism

- Explicitly specifying the name of the class, you can call the method of the parent (as well as any other object):

```
>>> class Child (Parent):
...     def __init__ (self):
...         Parent.__init__(self)
...
```

- In general case to get the parent class the function **`super`** is applied:

```
>>> class Child(Parent):
...     def __init__ (self):
...         super(Child, self).__init__(self)
...
```

Polymorphism: Virtual Methods

- Using a special provided exception **`NotImplementedError`**, you can simulate pure virtual methods:

```
>>> class Abstobj (object):
...     def abstmeth (self):
...         raise NotImplementedError ('Method Abstobj.abstmeth is pure virtual')
...
>>> Abstobj().abstmeth()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in abstmeth
NotImplementedError: Method Abstobj.abstmeth is pure virtual
```

Polymorphism: Virtual Methods

- Or, using a python decorator:

```
>>> def abstract(func):
...     def closure(*dt, **mp):
...         raise NotImplementedError("Method %s is pure virtual" % func.__name__)
...     return closure
...
>>> class abstobj(object):
...     @abstract
...     def abstmeth(self):
...         pass
...
>>> A = abstobj()
>>> A.abstmeth()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in closure
NotImplementedError: Method abstmeth is pure virtual
```

Polymorphism

- Changing attribute **`__class__`**, you can move an object up or down the inheritance hierarchy (as well as to any other type):

```
>>> class Parent(object):
...     def isParOrPChild(self):
...         return True
...     def who(self):
...         return 'parent'
...
>>> class Child (Parent):
...     def who (self):
...         return 'child'
...
>>> c = Child()
>>> c.val = 10
>>> c.who()
'child'
>>> c.__class__ = Parent
>>> c.who()
'parent'
>>> c.val
10
```

- However, in this case, no type conversions are made, so care about data consistency remains entirely on the programmer.

Inheritance and Multiple Inheritance

- Python supports both single inheritance and multiple, allowing the class to be derived from any number of base classes:

```
>>> class Par1 (object): # inherits the base class - object
...     def name1 (self):
...         return 'Par1'
...
>>> class Par2 (object):
...     def name2 (self):
...         return 'Par2'
...
>>> class Child (Par1, Par2): # create a class that inherits Par1, Par2 (and object)
...     pass
...
>>> x = Child()
>>> x.name1(), x.name2() # instance of Child has access to methods of Par1 and Par2
('Par1', 'Par2')
```

- In Python (because of the "duck typing"), the lack of inheritance does not mean that the object can not provide the same interface.

"New" and "Classic" Classes

- In versions prior to 2.2, some object-oriented features of Python were noticeably limited. Starting with version 2.2, Python object system has been significantly revised and expanded. However, for compatibility with older versions of Python, it was decided to make two object models: the "classical" type (fully compatible with old code) and "new". In version Python 3.0 support "old" classes will be removed.
- To build a "new" class is enough to inherit it from other "new". If you want to create a "pure" class, you can inherit from the **`object`** - the parent type for all "new" classes.

```
class OldStyleClass: # class of "old" type
    pass

class NewStyleClass(object): # class of "new" type
    pass
```

- All the standard classes - classes of "new" type.

Settlement of access to methods and fields

Behind a quite easy to use mechanism to access attributes in Python lies a fairly complex algorithm. Below is the sequence of actions performed by the interpreter when resolving `object.field` call (search stops after the first successfully completed step, otherwise there is a transition to the next step):

1. If the object has method `__getattribute__`, then it will be called with parameter '`field`' (or `__setattr__` or `__delattr__` depending on the action over the attribute)
2. If the object has field `__dict__`, then `object.__dict__['field']` is sought
3. If `object.__class__` has field `__slots__`, a '`field`' is sought in `object.__class__.__slots__`
4. Checking `object.__class__.__dict__['fields']`
5. Recursive search is performed on `__dict__` `of` all parent classes
6. If the object has method `__getattr__`, then it is called with a parameter '`field`'
7. An exception `AttributeError` is roused.

Aggregation. Containers. Iterators

Aggregation, when one object is part of another, or «HAS-A» relation, is implemented in Python using references. Python has some built-in types of containers: list, dictionary, set. You can define your own container classes with its own logic to access stored objects.

The following class is an example of container-dictionary, supplemented by the possibility of access to the values using the syntax of access to attributes:

```
class Storage(dict):
    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError, k:
            raise AttributeError, k

    def __setattr__(self, key, value):
       self[key] = value

    def __delattr__(self, key):
        try:
            del self[key]
        except KeyError, k:
            raise AttributeError, k

    def __repr__(self):
      return '<Storage ' +
             dict.__repr__(self) + '>'
```

Aggregation. Containers. Iterators

- Here's how it works:

```
>>> v = Storage(a=5)
>>> v.a
5
>>> v['a']
5
>>> v.a = 12
>>> v['a']
12
>>> del v.a
```

- To access containers it’s very convenient to use iterators:

```
>>> cont = dict(a=1, b=2, c=3)
>>> for k in cont:
...     print k, cont[k]
...
a 1
c 3
b 2
```

Metaclasses

- I’s not always enough to have ordinary capabilities of object-oriented programming. In some cases you want to change the character of the class system: extend the language with new types of classes, change the style of interaction between the classes and the environment, add some additional aspects that affect all classes used in applications, etc.
- When declare a metaclass, we can take class **`type`** as a basis. For example:

```
>>> # Description metaclass
... class myobject (type):
...     def __new__ (cls, name, bases, dict):
...         print "NEW", cls.__name__, name, bases, dict
...         return type. __new__ (cls, name, bases, dict)
...     def __init__ (cls, name, bases, dict):
...         print "INIT", cls.__name__, name, bases, dict
...         return super (myobject, cls). __init__ (cls, name, bases, dict)
...
```

Metaclasses

```
>>> # Description metaclass
... class myobject (type):
...     def __new__ (cls, name, bases, dict):
...         print "NEW", cls.__name__, name, bases, dict
...         return type. __new__ (cls, name, bases, dict)
...     def __init__ (cls, name, bases, dict):
...         print "INIT", cls.__name__, name, bases, dict
...         return super (myobject, cls). __init__ (cls, name, bases, dict)
...
>>> # Derived class based on the metaclass (replaces 'class' operator)
>>> MyObject = myobject ("MyObject", (), {})
NEW myobject MyObject () {}
INIT MyObject MyObject () {}
>>> # Pure inheritance of another class from just generated
>>> class MySubObject (MyObject):
...     def __init__ (self, param):
...         print param
...
NEW myobject MySubObject (<class '__main__.MyObject'>,) {'__module__': '__main__',
'__init__': <function __init__ at 0x6c15f0>}
INIT MySubObject MySubObject (<class '__main__.MyObject'>,) {'__module__':
'__main__', '__init__': <function __init__ at 0x6c15f0>}
>>> # Get an instance of the class
>>> myobj = MySubObject ("parameter")
parameter
```

Methods

- Syntax of a method has no difference from the description of a function, except for its position within a class and specific first formal parameter **`self`**, using which the inside of the method can be invoked the class instance itself (the name of **`self`** is a convention that Python developers follow to):

```
class MyClass(object):
    def mymethod(self, x):
        return x == self._x
```

Static Methods

- Static methods in Python are the syntactic analogues of static functions in the major programming languages. They do not receive neither an instance (**`self`**) nor class (**`cls`**) as the first parameter. To create a static method **`staticmethod`** decorator is used (only "new" classes can have static methods):

```
>>> class D(object):
...     @staticmethod
...     def test (x):
...         return x == 0
...
>>> D.test(1) # access to static method can be obtained via class
False
>>>
>>> F = D()
>>> F.test(0) # and via an instance of the class
True
```

- Static methods are implemented using properties

Class Methods

- Class methods in Python are intermediate between static and regular. While regular methods get as the first parameter an instance of class and static get nothing, in the class methods a class is passed. Ability to create class methods is a consequence of the fact that in Python classes are also objects. To create a class you can use decorator **`classmethod`** method (only "new" classes can have class methods):

```
>>> class A(object):
      def __init__(self, int_val):
          self.val = int_val + 1
      @classmethod
      def fromString(cls, val):    # 'cls' is commonly used instead of 'self'
          return cls(int(val))
...
>>> class B(A):
        pass
...
>>> x = A.fromString("1")
>>> print x.__class__.__name__
A
>>> x = B.fromString("1")
>>> print x.__class__.__name__
B
```

Multimethods (Multiple Dispatch)

- Multimethod is a function that has multiple versions, distinguished by the type of the arguments.

```
from boo import multimethod

@multimethod(int, int)
def foo(a, b):
    ...code for two ints...

@multimethod(float, float):
def foo(a, b):
    ...code for two floats...

@multimethod(str, str):
def foo(a, b):
    ...code for two strings...
```

- An example to illustrate the essence of multiple dispatch can serve **`add()`** function from the module **`operator`**:

```
>>> import operator as op
>>> print op.add(2, 2), op.add(2.0, 2), op.add(2, 2.0), op.add(2j, 2)
4 4.0 4.0 (2+2j)
```

Object Persistence

- Objects always have their representation in the computer memory and their lifetime is not longer than the program's. However, it is often necessary to save data between starting an application and / or transfer them to other computers. One solution to this problem is object persistence which is achieved by storing representations of objects (serialization) in the form of byte sequences and their subsequent recovery (deserialization).
- Module **`pickle`** is the easiest way to "conservation" of objects in Python.
- The following example shows how the serialization-deserialization:

```
>>> # Serialization
>>> import pickle
>>> p = set([1, 2, 3, 5, 8])
>>> pickle.dumps(p)
'c__builtin__\nset\np0\n((lp1\nI8\naI1\naI2\naI3\naI5\natp2\nRp3\n.'

>>> # Deserialization
>>> import pickle
>>> p = pickle.loads('c__builtin__\nset\np0\n((lp1\nI8\naI1\naI2\naI3\naI5\natp2\nRp3\n.')
>>> print p
set([8, 1, 2, 3, 5])
```

References

- http://docs.python.org/tutorial/classes.html
- Объектно-ориентированное программирование на Питоне
- OOP in Python after 2.2
- Python 101 - Introduction to Python
- Python Basic Object-Oriented Programming

SoftServe
Ability. Agility. Advantage.

Questions?