Scripting Programming Languages and their Applications

Jan Gaura

October 2, 2012

Object Oriented Programming

First of all – Quote ;-)

Alan Kay:

Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.

> The Computer Revolution hasn’t happend yet – Keynote, OOPSLA 1997

Motivation

```
struct {
    name = ''
} Dog;
struct {
    name = ''
} Cat;
make_sound(struct) {
    if type(struct) == Dog {
        print struct.name + 'Haf!'
    }
    else if type(struct) == Cat {
        print struct.name + 'Minau!'
    }
}

struct dog = Dog("Lassie");
struct cat = Cat("Tom");
make_sound(dog); // 'Lassie: Haf!'
make_sound(cat); // 'Tom: Minau!'
```

Three Pillars of OOP

- inheritance
- encapsulation
- polymorphism

- we'll see that OOP can be very easy in dynamic languages
- we don't have to write so much code like in staticaly typed languages such as Java, C++, C#

Object Oriented Programming

- so far we've seen that everything is an object of some class, but we didn't create our own classes yet
- let's start doing it now

Classes - Syntax

```
1    class <name>(<bases>):
         <body>
```

- if we have no base class(es), we use `object`
- in <body>, we define methods using `def` as normal methods
- assignment becomes class attributes
- attributes of any base class are also attributes of the new class until "overridden"

Classes - Example

```
class Battlestar(object):

    def __init__(self, name, commander): # initializer
        self.name = name                 # instance attr
        self.commander = commander

    def identify(self):                  # method
        return 'This is Battlestar %s, commanded by %s.' \
            % (self.name, self.commander)

galactica = Battlestar('Galactica', 'Bill Adama')
pegasus = Battlestar('Pegasus', 'Helena Cain')

print galactica.identify()
This is Battlestar Galactica, commanded by Bill Adama.

print pegasus.identify()
This is Battlestar Pegasus, commanded by Helena Cain.
```

Classes - Short Summary

- __init__(self, . . .) is always used as initializer, something like Java's constructor
- we don't use class name as initializer
- instance attributes are those with `self` defined in a method
- class attributes are defined outside methods – see next example
- we don't use `new` to create new instance – it's useless

Classes - Attributes I

```
class eg(object):
    cla = []                    # class attribute

    def __init__(self):         # initializer
        self.ins = {}           # instance attribute

    def meth1(self, x):         # a method
        self.cla.append(x)

    def meth2(self, y, z): # another method
        self.ins[y] = z

es1 = eg()
es2 = eg()
```

Classes - Attributes II

```
print es1.cla, es2.cla, es1.ins, es2.ins
      []       []       {}       {}

es1.meth1(1)
es1.meth2(2, 3)

es2.meth1(4)
es2.meth2(5, 6)

print es1.cla, es2.cla, es1.ins, es2.ins
      [1, 4]   [1, 4]   {2: 3}   {5: 6}

print es1.cla is es2.cla
True

print es1.ins is es2.ins
False
```

Subclassing

```
class sub(eg):
    def meth2(self, x, y=1):      # override
        eg.meth2(self, x, y)      # super-call
        # or: super(sub, self).meth2(x, y)

class repeater(list):
    def append(self, x):
        for i in 1, 2:
            list.append(self, x)

class data_overrider(sub):
    cla = repeater()
```

Properties

```
class blah(object):
    def getter(self):
        return ...
    def setter(self, value): ...

    name = property(getter, setter)

inst = blah()
```

Now...:

```
print inst.name  # read, like inst.getter()
inst.name = 23   # write, like inst.setter(23)
```

Properties

- "hide" attributes behind getters/setters "for flexibility"
- expose interesting attributes directly
- if/when in a future release you need a getter and/or a setter...:
  - write the new needed methods,
  - wrap them up into a property
  - and all client-code of the class need NOT change!

Overloading Operators

- "special methods" have names starting and ending with __ (double-underscore AKA "dunder"):

```
__new__      __init__     __del__      # ctor, init, finalize
__repr__     __str__      __int__      # convert
__lt__       __gt__       __eq__  ...  # compare
__add__      __sub__      __mul__  ... # arithmetic
__call__     __hash__     __nonzero_  ...
__getattr__  __setattr__  __delattr__
__getitem__  __setitem__  __delitem__
__len__      __iter__     __contains__
__get__      __set__      __enter__
__exit__  ...
```

Overloading Operators, Example Part I

```
class Vector(object):
    def __init__(self, *vec):
        #warning: vec will be tuple, immutable
        self.vec = vec

    def __add__(self, other):
        ret_vec = []
        for no, item in enumerate(self.vec):
            ret_vec.append(item + other.vec[no])

        #return immutable tuple
        return tuple(ret_vec)
```

Overloading Operators, Example Part II

```
def main():
    v1 = Vector(1, 2, 3, 4)
    print v1

    v2 = Vector(5, 6, 7, 8)
    print v2

    v1 = v1 + v2
    print v1

if __name__ == '__main__':
    main()
```

Overloading Operators, Example Part III

```
Vector v1:
1 2 3 4

Vector v2:
5 6 7 8

Result vector:
6 8 10 12
```

References

- Alex Martelli, Painless Python for Proficient Programmers
- Django documentation
- Adam Fast, intro to geodjango