

Scripting Programming Languages and their Applications

Jan Gaura

October 2, 2012



Object Oriented Programming

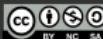


First of all – Quote ;-)

Alan Kay:

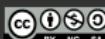
Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.

The Computer Revolution hasn't happened yet – Keynote,
OOPSLA 1997



Motivation

```
1   struct {
2       name = ''
3   } Dog;
4   struct {
5       name = ''
6   } Cat;
7   make_sound(struct) {
8       if type(struct) == Dog {
9           print struct.name + 'Haf!'
10      }
11     else if type(struct) == Cat {
12         print struct.name + 'Minau!'
13     }
14 }
15
16     struct dog = Dog("Lassie");
17     struct cat = Cat("Tom");
18     make_sound(dog); // 'Lassie: Haf!'
19     make_sound(cat); // 'Tom: Minau!'
```



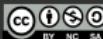
Three Pillars of OOP

- 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 statically 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

```
1 class Battlestar(object):
2
3     def __init__(self, name, commander): # initializer
4         self.name = name                 # instance attr
5         self.commander = commander
6
7     def identify(self):               # method
8         return 'This is Battlestar %s, commanded by %s.' \
9             % (self.name, self.commander)
10
11    galactica = Battlestar('Galactica', 'Bill Adama')
12    pegasus = Battlestar('Pegasus', 'Helena Cain')
13
14    print galactica.identify()
15    This is Battlestar Galactica, commanded by Bill Adama.
16
17    print pegasus.identify()
18    This is Battlestar Pegasus, commanded by Helena Cain
```



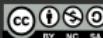
Classes - Short Summary

- `__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

```
1 class eg(object):
2     cla = []                      # class attribute
3
4     def __init__(self):           # initializer
5         self.ins = {}            # instance attribute
6
7     def meth1(self, x):          # a method
8         self.cla.append(x)
9
10    def meth2(self, y, z):       # another method
11        self.ins[y] = z
12
13    es1 = eg()
14    es2 = eg()
```



Classes - Attributes II

```
print es1.cla, es2.cla, es1.ins, es2.ins
2          []        []        {}        {}

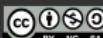
4      es1.meth1(1)
      es1.meth2(2, 3)

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

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

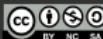
12     print es1.cla is es2.cla
14     True

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



Subclassing

```
1  class sub(eg):
2      def meth2(self, x, y=1):      # override
3          eg.meth2(self, x, y)      # super-call
4          # or: super(sub, self).meth2(x, y)
5
6
7  class repeater(list):
8      def append(self, x):
9          for i in 1, 2:
10             list.append(self, x)
11
12 class data_overrider(sub):
13     cla = repeater()
```

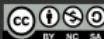


Properties

```
1 class blah(object):
2     def getter(self):
3         return ...
4     def setter(self, value): ...
5
6     name = property(getter, setter)
7
8     inst = blah()
```

Now...:

```
1 print inst.name # read, like inst.getter()
2 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"):

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



Overloading Operators, Example Part I

```
1   class Vector(object):
2       def __init__(self, *vec):
3           #warning: vec will be tuple, immutable
4           self.vec = vec
5
6       def __add__(self, other):
7           ret_vec = []
8           for no, item in enumerate(self.vec):
9               ret_vec.append(item + other.vec[no])
10
11      #return immutable tuple
12      return tuple(ret_vec)
```



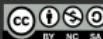
Overloading Operators, Example Part II

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

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

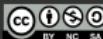
v1 = v1 + v2
print v1

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



Overloading Operators, Example Part III

```
Vector v1:  
2      1 2 3 4  
  
4      Vector v2:  
5      6 7 8  
  
6      Result vector:  
8      6 8 10 12
```



References

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

