

# Core OO Concepts from a Programming Perspective — Python — Part II Object Oriented Analysis and Design

Aron Trauring

T++ Technical Skills Training Program

CUNY Institute for Software Design & Development (CISDD)

New York Software Industry Association (NYSIA)

December 3rd, 2004

# Inheritance Benefit of Objects

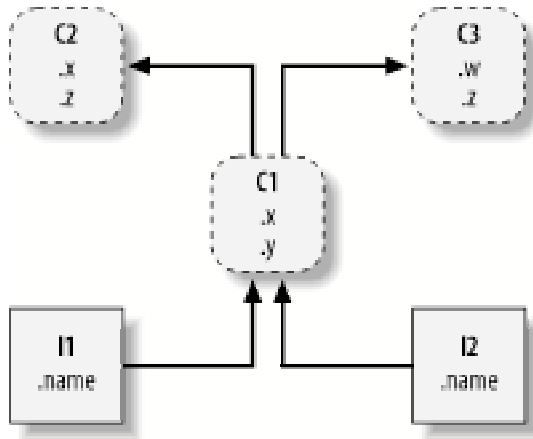
- Set of attributes passed to descendants
- Conceptual — Is-A Relationship — Defines sets of responsibilities down the tree
- Protocol (Interface) — Customization / Specialization — Sub-classes can replace, provide or extend behaviors of parent classes
- Implementation — Code reuse

## object.attribute

- object is derived from a class — instantiation
- attribute is a name found by a search in object and all its ancestors
- Search algorithm: find the first occurrence of attribute by looking in object, and all classes above it
- Inheritance — objects linked into a class tree are the union of all the attributes defined in all their tree parents (ancestors), all the way up the tree
- In Python, this is literal: it builds up trees of linked objects with code, and really does climb this tree at runtime searching for attributes, every time we say object.attribute

- Multiple inheritance — left to right search

# Mini Exercise 1



- `I1.x` and `I2.x` both find `x` in ?
- `I1.y` and `I2.y` both find `y` in ?
- `I1.z` and `I2.z` both find `z` in ?

- l1.w and l2.w both find w in ?
- l2.name finds name in ?

## Conceptual — Is-A Relationship

```
class Employee(object):
    def __init__(self, name, salary=0):
        self.name    = name
        self.salary  = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print self.name, "does stuff"
    def resign(self):
self.myresign()
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.s

class Chef(Employee):
```

```
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print self.name, "makes food"

class Server(Employee):
    def __init__(self, name, counter=1):
        Employee.__init__(self, name, 40000)
    self.counter = counter
    def work(self):
        print self.name, "interfaces with customer at counter", self
    def myresign(self):
        print "I resign you SOB"

class PizzaRobot(Chef):
    def __init__(self, name):
```

```
        Chef.__init__(self, name)
    def work(self):
        print self.name, "makes pizza"

if __name__ == "__main__":
    bob = PizzaRobot('bob')           # Make a robot named bob.
    print bob                          # Runs inherited __repr__
    bob.work( )                        # Run type-specific action.
    bob.giveRaise(0.20)                # Give bob a 20% raise.
    print bob; print

    print "Work:"
    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work( )
```

```
print ; print "Resign:"  
joe = Server('joe')  
joe.resign()
```

```
$ python employees.py  
<Employee: name=bob, salary=50000>  
bob makes pizza  
<Employee: name=bob, salary=60000.0>
```

Work:

Employee does stuff

Chef makes food

Server interfaces with customer at counter 1

PizzaRobot makes pizza

Resign:

I resign you SOB

- Python code allows you to quickly build a working model of domain layer

## Protocol — Classes are Customized by Inheritance

- `giveRaise` is *inherited* from the base `Employee` super-class and can be used in all sub-classes
- `work` is *replaced* (over-ridden) in each of the sub-classed employees
- `work` is *extended* in the `Server` sub-class by adding the counter parameter
- `Employee` *delegates* `resign` which is provided by `Server`

## Implementation — Code Reuse

- PizzaRobot trivial to implement
- New types of employees can be added with minimal amount of coding

## Polymorphism Benefit of Objects

- “Many forms”
- Conceptual — Delegation of responsibility down the tree — each Employee sub-type defines his/her own work
- Protocol (Interface) — “many forms” of methods for same operation (depends on instance)
- Implementation — overloading operators or operations

# Python is Dynamically and Strongly Typed

- Separation between Object (Type) and Variable (Name)
- Objects have fixed types
- Variables (names) are merely references (pointers) to objects (mathematical model of a variable).
- Variables (names) have no fixed type — they can reference any object

## All Python Objects Have

- Unique identity (an integer, returned by `id(x)`) which can't be changed
- Type (returned by `type(x)`) which can't be changed
- Some content which may or may not be changeable (mutable)

## Some Python Objects Have

- Zero or more methods
- Zero or more names
- Methods that allow you to change the contents of the object (modify it in place, that is)
- Methods that allow you to access the contents, not change it

# Python Names

- Not really properties of the object, and the object itself doesn't know what it's called
- An object can have any number of names, or no name at all
- Names live in namespaces
- Assignments change references and modify namespaces, not objects

# Python Names and Objects

- A name (variable) like  $V$  is created when it is first assigned a value by your code
- The assignment also creates a reference between (binds) the name in the namespace and the object
- Future assignments change the already-created name to have a new reference
- All names (variables) must be explicitly assigned before they can be used; use of unassigned variables results in an exception
- When names are made to reference new objects, Python also reclaims the old object, if it is not reference by any other name (or object). (Garbage Collection)

- For efficiency purposes some objects are cached (zero-named objects)

## Mini Exercise 2

```
>>> a=3
>>> a
?
>>> b=a
>>> b
?
>>> b='spam'
>>> b
?
>>> a
?
>>> b=a
>>> b
?
```

```
>>> a='spam'  
>>> b  
?  
>>> c  
>>> a = b = 3  
>>> a = 4  
>>> print a, b  
?
```

## Built-in Types

Object Type	Examples	Category
Number	3.1415, 1234, 999L, 3+4j	Immutable
Strings	'spam', "guido's"	Immutable Sequence
Tuples	(1,'spam', 4, 'U')	Immutable Sequence
Lists	[1, [2, 'three'], 4]	Mutable Sequence
Dictionaries	{'food': 'spam', 'taste': 'yum'}	Mutable Mapping
Files	text = open('eggs', 'r').read( )	

- Lists are ordered collections of other objects, and indexed by positions that start at 0.
- Dictionaries are collections of other objects that are indexed by key instead of position.

- Both dictionaries and lists may be nested, can grow and shrink on demand, and may contain objects of any type
- Mutability means applying a method

## Built-in Types and Classes

- In modeling at any level, there is no distinction between types and classes
- In Python, built-in types and defined classes operate in an identical fashion
- In Python, built-in types may be sub-classed
- Smaller conceptual gap in implementing the Domain Layer

## Mini Exercise 3

```
>>> name = []
>>> name.append(1)
>>> name
[1]
>>> name[0] = 4
>>> name
[4]
>>> print a, b
4 3
>>> c = a + b
>>> c
7
```

- `name[]` is just a method call

- $a + b$  is just a method call

# Protocol Polymorphism

- Some OOP languages define polymorphism to mean overloading functions based on the type signatures of their arguments
- Python Polymorphism is a more powerful and dynamic concept — it is responsibility based not “data hiding” based
- Based on object interfaces, not types
- Because attributes are always resolved at runtime, objects that implement the same protocols are interchangeable
- Clients don't need to know what sort of object is implementing a method they call

# Python Polymorphism Implements The Three Levels of Flexibility

1. Conceptual — Objects responsible for own behavior
2. Protocol (Interface) — Control program can talk to different objects as if they were the same
3. Implementation — Control program doesn't need to be aware of the internal behaviors that lead to desired outcome

## Mini Exercise 4

```
>>> class One(object):
    def AMethod1(self):
        print "this is a Class One method"

class Two(object):
    def AMethod1(self):
        print "this is a Class Two method"

    c1=One()
    c2=Two()

>>> c1.AMethod1()
?
```

```
>>> c2.AMethod1()
```

```
?
```

```
>>> for C in c1,c2:  
        C.AMethod1()
```

```
?
```

# Implementation Polymorphism — Overloading Operators and Operations

- Operator overloading lets classes intercept normal Python operations
- Classes can overload all Python expression operators
- Classes can also overload operations: printing, calls, qualification, etc.
- Overloading makes class instances act more like built-in types
- Overloading is implemented by providing specially named class methods ( of form “\_\_method\_\_”)
- All operator overloading methods are optional

- `__init__` constructor (object creation operation) tends to appear in most classes
- `__repr__` (print operation) often appears - see employee example
- It is still all about delegating responsibility

## Encapsulation Benefits

- Hiding
- Conceptual — Composition — Has-a-Relationship
- Protocol (Interface) — hiding implementation details
- Implementation — *Properties* hide accessors attributes

## Conceptual Encapsulation — Composition

```
from employees import PizzaRobot, Server

class Customer(object):
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print self.name, "orders from", server, "at counter", server
    def pay(self, server):
        print self.name, "pays for item to", server, "at counter", s

class Oven(object):
    def bake(self):
        print "oven bakes"
```

```
class PizzaShop(object):
    def __init__(self):
        self.server = Server('Pat', 2)           # Embed other objects.
        self.chef    = PizzaRobot('Bob')        # A robot named bob
        self.oven    = Oven( )
        self.cashier = Server('Matt', 1)

    def order(self, name):
        customer = Customer(name)                # Activate other objects
        customer.order(self.server)              # Customer orders from s
        self.chef.work( )
        self.oven.bake( )
        customer.pay(self.cashier)

if __name__ == "__main__":
    scene = PizzaShop( )                        # Make the composit
```

```
    scene.order('Homer')           # Simulate Homer's order
    print '...'
    scene.order('Shaggy')          # Simulate Shaggy's order
```

```
Homer orders from <Employee: name=Pat, salary=40000> at counter 2
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Matt, salary=40000> at counter 2
...
Shaggy orders from <Employee: name=Pat, salary=40000> at counter 2
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Matt, salary=40000> at counter 2
```

- Composition has to do with components: parts of a whole

- Composition also reflects the “has-a” relationships between parts
- Pizzashop *contains* servers, oven, robochef
- Pizzashop *controls* how contained objects react to Customer through delegation

## Protocol Encapsulation

```
class Processor(object):
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while 1:
            data = self.reader.readline( )
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
        assert 0, 'converter must be defined'
```

- Processor is a *protocol container*

- reader and writer objects passed as parameters — what or how is read or written is hidden from Processor
- Converter is *delegated* through inheritance (assertion raises exception if there is no *provider*)
- How conversion takes place is hidden from Processor

```
from streams import Processor
```

```
class Uppercase(Processor):  
    def converter(self, data):  
        return data.upper( )
```

```
if __name__ == '__main__':  
    import sys
```

```
Uppercase(open('3spam.txt'), sys.stdout).process( )
```

- Uppercase gets processing logic through inheritance — how it is done is hidden
- reader and writer methods also hidden from Uppercase

```
spam  
Spam  
SPAM!
```

```
$ python converters.py  
SPAM  
SPAM  
SPAM!
```

- To process different sorts of streams, pass in different sorts of reader and writer objects to the class construction call

```
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt'),
>>> prog.process( )
```

```
$ cat spamup.txt
SPAM
SPAM
SPAM!
```

- Can also pass in arbitrary objects wrapped up in classes that define the required input and output method protocols

```
>>> from converters import Uppercase
```

```
>>>
>>> class HTMLize:
...     def write(self, line):
...         print '<PRE>%s</PRE>' % line[:-1]
...
>>> Uppercase(open('spam.txt'), HTMLize( )).process( )
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
```

- We get uppercase conversion (by *inheritance*) and HTML formatting (by *composition*), even though Processor knows nothing about either
- Code reuse — all we had to code here was the HTML formatting step; the rest is free

- The power of frameworks

## Implementation Encapsulation

```
>>> class Rectangle(object):
    def __init__(self):
        self.width=0
        self.height=0
        self.area=0
    def setSize(self,size):
        self.width, self.height = size
    def getSize(self):
        self.area = self.width * self.height
        return self.width, self.height, self.area
    size = property(getSize, setSize)
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
```

```
>>> r.size
(10, 5, 50)
>>> r.size = 150, 100
>>> r.width
150
>>> r.size
(150, 100, 15000)
```

- Hides implementation details of Rectangle's *accessor* methods
- Size looks like a normal attribute
- Can change implementation without effecting using code