Python – MetaMeta + Patterns

Isak Karlsson
isak-kar@dsv.su.se

Implementing a descriptor

```
def trace(f):
    def log_wrapper(*args, **kv):
        call = "%s(%s)" % (f.func_name,
                ", ".join(map(lambda a: a.__class__.__name__, args)))

        print "TRACE:\t Start call: '%s'" % (call)

        start = time.time()
        ret = f(*args, **kv)
        elapsed = time.time() - start
        print "TRACE:\t End call  : '%s' \n\t Took %s sek" % (call, elapsed)

    log_wrapper.__name__= f.__name__
    log_wrapper.__doc__ = f.__doc__
    return log_wrapper
```

Using the descriptor

```
@trace
def test(s):
   """ Documentation ...."""
   print s

class Test(object):
   def class_test(self, x):
      print x
   class_test = trace(class_test)

>> test("Test")
TRACE:     Start call: 'test(str)'
isak
TRACE:     End call  : 'test(str)' Took 6.91413879395e-06 sek
>> Test().class_test("test")
TRACE:     Start call: 'class_test(Isak, str)'
isak
TRACE:     End call  : 'class_test(Isak, str)' Took 5.00679016113e-06 sek

>> help(test)
Help on function test in module __main__:

test(*args, **kv)
    Documentation ....
```

Code objects and Compilation

```
def test():
    pass

def code_test():
    print "This is a test"


>> test.__code__ = code_test.__code__
>> c = test.__code__
>> c()
This is a test
>> exec c
This is a test
>> c = compile("print 1+1", "<test>", "exec")
>> test.__code__ = c
>> test()
2
```

Table of contents

What is a design pattern?

- “A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.” (Wikipedia, 2011)

Why should I care?

- “Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it.” (Gamma et.al, 2011)

Case study:

- You want to design a game, capable of constructing Mazes
- Mazes consist of rooms, rooms of walls and doors
- Doors connect rooms
- It should be extendable
  - It should be easy to extend with new and fancy walls or doors
    - Doors, rooms and walls must fit
  - The maze creation algorithm must not be rewritten for any change of, door, room or wall-type

First try

```
class Maze(object):
    def __init__(self):
        self.rooms = []
    def add_room(self, room):
        self.rooms.append(room)
   def get_room(self, no):
        return self.rooms[no]
```

```
class Part(object): def enter(self): pass
```

```
class Wall(Part): Pass
```

```
class Door(Part):
    def __init__(self, r1=None, r2=None):
        self.room1 = r1
        self.room2 = r2
        self.is_open = False
```

```
class Room(Part):
    def __init__(self):
        self.walls = {}
    def set_side(self, side, part):
        self[side] = part
```

```
def create_maze():
    m = Maze()
    r1 = Room()
    r2 = Room()

    d = Door(r1, r2)

    m.add_room(r1)
    m.add_room(r2)

    r1.set_side("north", Wall())
    r1.set_side("south", Wall())
    r1.set_side("east", d)
    r1.set_side("west", Wall())

    r2.set_side("north", Wall())
    r2.set_side("south", Wall())
    r2.set_side("east", Wall())
    r2.set_side("west", d)

    return m
```

First try

```
mace = create_mace()
```

Is it sufficient?

- We can construct mazes
- They consist of rooms, walls, and doors
- Doors connect rooms
- But?
  - It should be extendable
    - It should be easy to extend with new and fancy walls or doors
    - Doors, rooms and walls must fit
    - The maze creation algorithm must not be rewritten for any change of, door, room or wall-type

Second try; Abstract factory to the rescue

```
class MazeFactory(object):
    def make_maze(self):
        return Maze()
    def make_room(self):
        return Room()
    def make_door(self, r1, r2):
        return Door(r1, r2)
    def make_wall(self):
        return Wall()
```

The intent is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes." (Gamma et.al)

What have we accomplished?

- Nothing.... yet!

Second try, factory

```python
def make_maze(factory):
    m = factory.make_maze()
    r1 = factory.make_room()
    r2 = factory.make_room()

    d = factory.make_door(r1, r2)

    m.add_room(r1)
    m.add_room(r2)

    r1.set_side("north", factory.make_wall())
    r1.set_side("south", factory.make_wall())
    r1.set_side("east", d)
    r1.set_side("west", factory.make_wall())

    r2.set_side("north", factory.make_wall())
    r2.set_side("south", factory.make_wall())
    r2.set_side("east", factory.make_wall())
    r2.set_side("west", d)
```

Second try, try

```python
mace = create_mace(MaceFactory())
```

Is it sufficient?

- We can construct mazes
- They consist of rooms, walls, and doors
- Doors connect rooms
- But?
  - It should be extendable
    - It should be easy to extend with new and fancy walls or doors
    - Doors, rooms and walls must fit
    - The maze creation algorithm must not be rewritten for any change of, door, room or wall-type

Example; extend

```
class TransparentWall(Part):
    pass

class TransparentFactory(MazeFactory):
    def make_wall(self):
        return TransparentWall()
```

Second try, try, try

```
mace = create_mace(TransparentFactory())
```

Introducing the Visitor pattern

- “Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.” (Gamma et.al, 2011)

Stupid case study:

- We have an Animal, that can eat
  - Eating affects the animal in different ways
    - Apple increase hp
    - Mushroom makes it fly
    - Spike decrease hp
- It should be possible to add new editable items without changing the animal

Static try:

```python
class Apple(object):
    pass

class Spike(object):
    pass

class Mushroom(object):
    pass
```

Static try:

```python
class Animal(object):
    def __init__(self):
        self.hp = 5
        self.fly = False

    def eat(self, food):
        if type(food) == Apple:
            self.hp += 1
        elif type(food) == Spike:
            self.hp -= 1
        elif type(food) == Mushroom:
            self.fly = True
        else:
            raise TypeError("Can't handle %s" % (type(food)))
```

Dynamic try:

```python
class Animal(object):
    def __init__(self):
        self.hp = 5
        self.fly = False

    def eat_apple(self):
        self.hp += 1

    def eat_pear(self):
        self.hp -= 1

    def eat_mushroom(self):
        self.fly = True

    def eat(self, food):
        function = food.__class__.__name__.lower()
        eval("s.eat_%s()" % (function), {"s": self})
```

Stupid case study:

- We have an Animal, that can eat
  - Eating affects the animal in different ways
    - Apple increase hp
    - Mushroom makes it fly
    - Spike decrease hp
- It should be possible to add new editable items without changing the animal

Visitor to the rescue!

```python
class Apple(object):
    def eaten(self, eater):
        eater.hp += 1

class Spike(object):
    def eaten(self, eater):
        eater.hp += 1

class Mushroom(object):
    def eaten(self, eater):
        eater.fly = True
```

Visitor to the rescue!

```python
class Animal(object):
    def __init__(self):
        self.hp = 5
        self.fly = False

    def eat(self, food):
        return food.eaten(self)
```

Stupid case study:

- We have an Animal, that can eat
  - Eating affects the animal in different ways
    - Apple increase hp
    - Mushroom makes it fly
    - Spike decrease hp
- It should be possible to add new editable items without changing the animal

Visitor to the rescue!

```python
class Milk(object):
     def eaten(self, eater):
          eater.hp += 10
          eater.white = True

# We can eat milk, and there are no
# requirements to edit the animal!
a = Animal()
a.eat(Milk())
print a.hp, a.white
```

Case study, more interesting:

- You want to implement a programming language...
- … and from an abstract syntax tree interpret the semantics
- But, you want each of the node in the tree to know nothing of how they are used, and:
  - It shall be possible to use them in different ways without sub classing them
  - Add new nodes to the tree without changing existing Nodes

Case study, more interesting:

```
<num> ::= 0...9+
<expr> ::= <term> [+|-] <expr> | <term>
<term> ::= <factor> [*|/] <term> | <factor>
<factor> ::= <num>
```

Implementing an interpreter

Live programming ahead – code available separately

Using Python to parse!

Live programming ahead – code available separately