The (lack of) design patterns in Python

Joe Gregorio
Google

Scope

My opinions.

A Story

Mythology

Blog

Python isn't Java without the compile

Language

Not just about Python

Language

Aren't Patterns Good?

The Lack of Patterns in Python

1. Define 'lack of patterns'
2. Show there really is a lack
3. Explain why
4. Draw useful conclusions

Hard numbers

comp.lang.python

comp.lang.python

“factory method pattern” - 0
“abstract-factory pattern” - 0
“flyweight pattern” - 3
“flyweight” - 36
“state pattern” - 10
“strategy pattern” - 25
“visitor pattern” - 60

comp.lang.python

“dark matter” - 2
“the pope” - 16
“sausage” - 66

Why

The patterns are built in.

```
class Bisection (FindMinima):
        def algorithm(self,line):
                return (5.5,6.6)

class ConjugateGradient (FindMinima):
        def algorithm(self,line):
                return (3.3,4.4)

class MinimaSolver: # context class
        strategy=''
        def __init__ (self,strategy):
                self.strategy=strategy

        def minima(self,line):
                return self.strategy.algorithm(line)

        def changeAlgorithm(self,newAlgorithm):
                self.strategy = newAlgorithm

def test():
        solver=MinimaSolver(ConjugateGradient())
        print solver.minima((5.5,5.5))
        solver.changeAlgorithm(Bisection())
        print solver.minima((5.5,5.5))
test()
```

An Example

```
def bisection(line):
    return 5.5, 6.6

def conjugate_gradient(line):
    return 3.3, 4.4

def test():
    solver = conjugate_gradient
    print solver((5.5,5.5))
    solver = bisection
    print solver((5.5,5.5))

test()
```

WikiPedia

This pattern is invisible in languages with first-class functions.

http://en.wikipedia.org/wiki/Strategy_pattern

Catalog of Language Features

- First-class functions
- Meta-programming
- Iterators
- Closures

First Class Functions

```
>>> def f(a, b):
...     return a + b
...
>>> g = f
>>> f(1, 2)
3
>>> g(1, 2)
3
>>> a = [f, g]
>>> a[0](4, 5)
9
```

Meta-Programming

```
class A(object):
    def __init__(self):
        self.a = "Hello"

class B(object):
    def __init__(self):
        self.a = " World"

def make_a_B():
    b = B()
    b.a = "!"
    return b

mycallables = [A, B, make_a_B]

>>> print [x().a for x in mycallables]

['Hello', ' World', '!']
```

Iterators

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Iterators

```
class MyFib(object):
    def __init__(self):
        self.i = 2
    def __iter__(self):
        return self
    def next(self):
        if self.i > 1000:
            raise StopIteration
        self.i = self.i * self.i
        return self.i

>>> print [x for x in MyFib()]
[4, 16, 256, 65536]
```

Iterator Pattern

In object-oriented programming, the Iterator pattern is a design pattern in which iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.

http://en.wikipedia.org/wiki/Iterator_pattern

Factory Method Pattern

The factory method pattern deals with the problem of creating objects without specifying the exact class of the object to be created.

Factory Method Pattern

```
class A(object):
    def __init__(self):
        self.a = "Hello"

class B(object):
    def __init__(self):
        self.a = " World"

myfactory = {
   "greeting" : A,
   "subject"  : B,
}

>>> print myfactory["greeting"]().a
Hello
```

Abstract Factory Pattern

This just a Factory Factory

Strategy Pattern

```
def bisection(line):
    return 5.5, 6.6

def conjugate_gradient(line):
    return 3.3, 4.4

def test():
    solver = conjugate_gradient
    print solver((5.5,5.5))
    solver = bisection
    print solver((5.5,5.5))

test()
```

Closures

Closures = First Class Functions + Env

Closure Example

```
>>> def too_big(limit):
        def compare(x):
            return x > limit
        return compare

>>> f = too_big(100)

>>> f(100)
False
>>> f(101)
True
```

Closure Example

```
def Dx(f, dx):
    def dfdx(x):
        return (f(x + dx) - f(x))/dx
    return dfdx

def f(x):
    return 3*x**2+x


>>> print f(1.0)
4.0
>>> print Dx(f, 0.01)(1.0)
7.03
>>> print Dx(Dx(f, 0.01), 0.01)(1.0)
6.0
```

Observer Pattern

The observer pattern (sometimes known as publish/subscribe) is a design pattern used in computer programming to observe the state of an object in a program.

http://en.wikipedia.org/wiki/Observer_pattern

Observer Pattern

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def scale(self, n):
        self.x = n * self.x
        self.y = n * self.y

def notify(f):
    def g(self, n):
        print n
        return f(self, n)
    return g

Point.scale = notify(Point.scale)

p = Point(2.0, 3.0)

p.scale(2.5)
```

Decorators

```
def notify(f):
    def g(self, n):
        print n
        return f(self, n)
    return g

class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @notify
    def scale(self, n):
        self.x = n * self.x
        self.y = n * self.y

p = Point(2.0, 3.0)

p.scale(2.5)
```

So What?

So what?

Other Patterns

Thoughts for the future

Patterns

Concurrency Patterns

Active Object
Balking
Guarded
Thread Pool
Reactor

Language Features

- Macros (Hygienic)
- Channels
- Multiple Dispatch

The (lack of) design patterns in Python

Joe Gregorio
Google

Scope

My opinions.

A Story

Mythology

Let me tell you a story
worked for new company (Java)
this company had a mythology
all companies have mythologies
you have to choose a subset of design tools, and then you have to continually justify those choices.
(embedded - C++)
Java was best
Language didn't matter (it was all Turing complete in the end)
(the code in java byte code)
All scripting languages were just Java w/o the compile

Blog

Python isn't Java without the compile

4

So what do you do as a frustrated geek?
you blog!
be clear, I'm not first person to talk about this

Peter Norvig
http://norvig.com/design-patterns/ppframe.htm

Bruce Tate – Beyond Java

Language

Not just about Python

could be any language
and not just about bashing Java
(we don't have the time for that)
What features of Python obviate Patterns

Language

Aren't Patterns Good?

Patterns are good because they give you a language to talk about program structure

OTOH, their use also points to a weakness in a language

The Lack of Patterns in Python

1. Define 'lack of patterns'
2. Show there really is a lack
3. Explain why
4. Draw useful conclusions

Hard numbers

comp.lang.python

8

Now my talk hinges on their being an actual lack of design patterns in Python.

104,128 messages

comp.lang.python

"factory method pattern" - 0
"abstract-factory pattern" - 0
"flyweight pattern" - 3
"flyweight" - 36
"state pattern" - 10
"strategy pattern" - 25
"visitor pattern" - 60

comp.lang.python

“dark matter” - 2
“the pope” - 16
“sausage” - 66

Why

The patterns are built in.

If your language of choice, in this case Python, supports an idiom natively, you don't need a name for it.

Nobody talks about the 'structured programming pattern', or the 'function pattern', or the 'object-oriented pattern'.

If you are old enough, you remember that there were actual arguments about this stuff, honest pushback from some programmers to 'structured programming'.

```
class Bisection (FindMinima):
        def algorithm(self,line):
                return (5.5,6.6)

class ConjugateGradient (FindMinima):
        def algorithm(self,line):
                return (3.3,4.4)

class MinimaSolver: # context class
        strategy=''
        def __init__ (self,strategy):
                self.strategy=strategy

        def minima(self,line):
                return self.strategy.algorithm(line)

        def changeAlgorithm(self,newAlgorithm):
                self.strategy = newAlgorithm

def test():
        solver=MinimaSolver(ConjugateGradient())
        print solver.minima((5.5,5.5))
        solver.changeAlgorithm(Bisection())
        print solver.minima((5.5,5.5))
test()
```


This example comes from comp.land.python and is an example of the “Strategy Pattern” strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime.

An Example

```
def bisection(line):
    return 5.5, 6.6

def conjugate_gradient(line):
    return 3.3, 4.4

def test():
    solver = conjugate_gradient
    print solver((5.5,5.5))
    solver = bisection
    print solver((5.5,5.5))

test()
```


Peter Otten:
“When most of your code does nothing in a pompous way that is a sure sign
that you are heading in the wrong direction. Here's a translation into
python”

WikiPedia

This pattern is invisible in languages with first-class functions.

http://en.wikipedia.org/wiki/Strategy_pattern

First-class functions make this pattern go away!

What other language features are there? And what patterns do they make 'invisible'?

Catalog of Language Features

- First-class functions
- Meta-programming
- Iterators
- Closures

First Class Functions

```
>>> def f(a, b):
...     return a + b
...
>>> g = f
>>> f(1, 2)
3
>>> g(1, 2)
3
>>> a = [f, g]
>>> a[0](4, 5)
9
```

16

A programming language is said to support first-class functions (or function literal) if it treats functions as first-class objects. Specifically, this means that the language supports constructing new functions during the execution of a program, storing them in data structures, passing them as arguments to other functions, and returning them as the values of other functions.

First Class **Object** Definition:

* being expressible as an anonymous literal value
* being storable in variables
* being storable in data structures
* having an intrinsic identity (independent of any given name)
* being comparable for equality with other entities
* being passable as a parameter to a procedure/function
* being returnable as the result of a procedure/function
* being constructible at runtime
* being printable
* being readable
* being transmissible among distributed processes
* being storable outside running processes

The fetish seems to be to define it so that **your** language has them, but **C** does not

Meta-Programming

```
class A(object):
    def __init__(self):
        self.a = "Hello"

class B(object):
    def __init__(self):
        self.a = " World"

def make_a_B():
    b = B()
    b.a = "!"
    return b

mycallables = [A, B, make_a_B]

>>> print [x().a for x in mycallables]

['Hello', ' World', '!']
```


Classes are Fist Class Objects

They are 'callable', like methods

Iterators

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Iterators

```
class MyFib(object):
    def __init__(self):
        self.i = 2
    def __iter__(self):
        return self
    def next(self):
        if self.i > 1000:
            raise StopIteration
        self.i = self.i * self.i
        return self.i

>>> print [x for x in MyFib()]
[4, 16, 256, 65536]
```


Now let's look at some patterns

Iterator Pattern

In object-oriented programming, the Iterator pattern is a design pattern in which iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.

http://en.wikipedia.org/wiki/Iterator_pattern

The definition of low-hanging fruit

Factory Method Pattern

The factory method pattern deals with the problem of creating objects without specifying the exact class of the object to be created.

The essence of the Factory Pattern is to "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses"

Factory Method Pattern

```
class A(object):
    def __init__(self):
        self.a = "Hello"

class B(object):
    def __init__(self):
        self.a = " World"

myfactory = {
   "greeting" : A,
   "subject"  : B,
}

>>> print myfactory["greeting"]().a
Hello
```


This is only a minor variation of using Classes as First Class Objects

Put the class objects in a map

Abstract Factory Pattern

This just a Factory Factory

Strategy Pattern

```
def bisection(line):
    return 5.5, 6.6

def conjugate_gradient(line):
    return 3.3, 4.4

def test():
    solver = conjugate_gradient
    print solver((5.5,5.5))
    solver = bisection
    print solver((5.5,5.5))

test()
```


First Class Functions

Closures

Closures = First Class Functions + Env

Jumping back up to Language Features

Closures are First Class Functions that can keep variables that were in the environment when they were created

Closure Example

```
>>> def too_big(limit):
        def compare(x):
            return x > limit
        return compare

>>> f = too_big(100)

>>> f(100)
False
>>> f(101)
True
```


The variable 'limit' lives on beyond the scope of too_big().

Closure Example

```
def Dx(f, dx):
    def dfdx(x):
        return (f(x + dx) - f(x))/dx
    return dfdx

def f(x):
    return 3*x**2+x


>>> print f(1.0)
4.0
>>> print Dx(f, 0.01)(1.0)
7.03
>>> print Dx(Dx(f, 0.01), 0.01)(1.0)
6.0
```


My favorite closure example of all time

Observer Pattern

The observer pattern (sometimes known as publish/subscribe) is a design pattern used in computer programming to observe the state of an object in a program.

http://en.wikipedia.org/wiki/Observer_pattern

Observer Pattern

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def scale(self, n):
        self.x = n * self.x
        self.y = n * self.y

def notify(f):
    def g(self, n):
        print n
        return f(self, n)
    return g

Point.scale = notify(Point.scale)

p = Point(2.0, 3.0)

p.scale(2.5)
```


First Class Functions/Closure and First Class Classes

Decorators

```
def notify(f):
    def g(self, n):
        print n
        return f(self, n)
    return g

class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @notify
    def scale(self, n):
        self.x = n * self.x
        self.y = n * self.y

p = Point(2.0, 3.0)

p.scale(2.5)
```


First Class Functions/Closure and First Class Classes

So What?

So what?

Now to draw useful conclusions

1. Python isn't Java w/o the compile
   Is a rich language with lots of features that obviate the need for many patterns

   Need to ask yourself, does Python let me do this better with First Class Functions/First Class Classes/Closures/etc.
2. Features reduce/remove patterns, and thus shorten code
3. There are still patterns, and where those patterns exist, that's a ripe place for a new language feature
4. This is a people problem

Other Patterns

Thoughts for the future

The thing to note is that there are patterns that aren't covered by Python today (true for all languages).

What are those patterns?
What are so higher level language features?

Patterns

Concurrency Patterns
Active Object
Balking
Guarded
Thread Pool
Reactor

The Active Object design pattern decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.

The Balking pattern is a software design pattern that only executes an action on an object when the object is in a particular state.

In concurrent programming, guarded suspension is a software design pattern for managing operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.

Language Features

- Macros (Hygienic)
- Channels
- Multiple Dispatch

Macros (Lisp, obviously), D has both hygienic and non-hygienic macros

Channels, see Rob Pike video on channels in Newsqueak. Comes from C.A.R. Hoare's Concurrent Sequential Processes.

Guido gives an example of doing multimethods with decorators, other libraries

Multiple dispatch or multimethods is the feature of some object-oriented programming languages in which a function or method can be specialized on the type of more than one of its arguments.