Object Oriented Design

Niko Wilbert

G-Node Python Summer School 2009, Berlin

Overview

1. General Design Principles
2. Object Oriented Programming in Python
3. Object Oriented Design Principles
4. Design Patterns

General Design Principles

Disclaimer

Learning good software design is a never ending process,
these slides can only get you started.

What really counts is your motivation to improve and question your code.

Some parts of this talk are just teasers to get you interested ;-)

Good Software Design

The single two most important principles are probably:

KIS

Keep it simple.
Overengineering is a dangerous trap.
Scientists: think Ockham's razor...

DRY

Don't repeat yourself.
(Sure path to a maintenance nightmare.)

import this

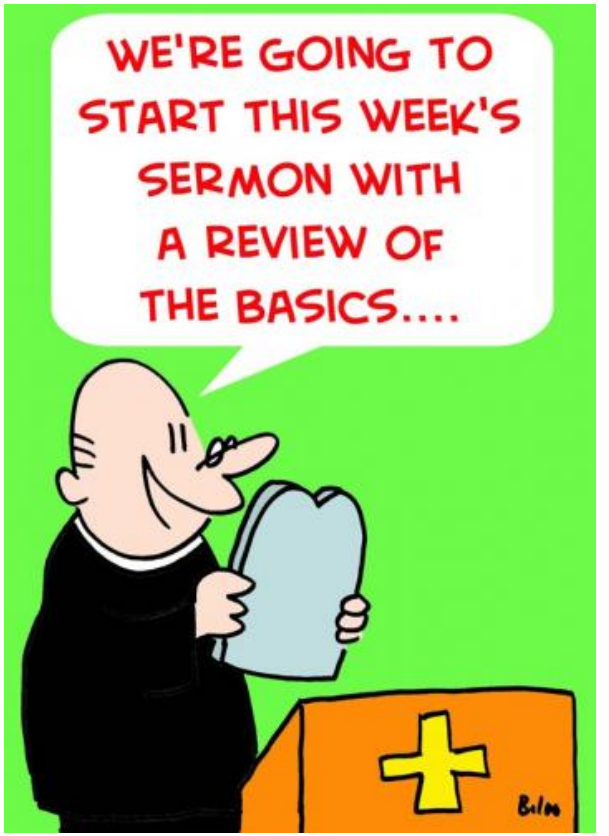
Python has its own set of guidelines, just execute `import this`:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Code that follows this is often called "pythonic".

Object Oriented Programming (in Python)



Object Orientated Programming (classic)

Objects

combine state (data) and behavior (algorithms).

Classes

define what is common for a whole class of objects.
"Snowy **is a** dog" =
"The Snowy object is an *instance* of the dog class."
Define once how a dog works and then reuse it for all dogs.
Classes correspond to variable types (they are *type objects*).

Encapsulation

Objects decide what is exposed to the outside world (by their *public interface*) and hide their implementation details to provide *abstraction*.
The abstraction should not *leak* implementation details.

Object Orientated Programming 2

Inheritance

"**a** dog (subclass) **is a** mammal (parent/superclass)"
A subclass *is derived from / inherits / extends* a parent class.
It reuses and extends it, and it can *override* parts that need specialization.
Liskov substitution principle: "What works for the Mammal class should also work for the dog class".

Polymorphism

Provide common way of usage for different classes, pick the correct underlying behavior for a specific class.
Example: the + operator for real and complex numbers.

Python OO Basics

- All classes are derived from `object` (new-style classes).

```
class Dog(object):
    pass
```

- Python classes / objects can have function attributes (*methods*) and data attributes (called members in other languages).
  "Attributes are whatever can be used with the dot-notation."

```
class Dog(object):
    def bark(self):
        print "Wuff!"

snowy = Dog()
snowy.bark()  # first argument (self) is bound to this Dog instance
snowy.a = 1  # added attribute a to snowy
```

- Always define all instance data attributes in `__init__` first (even if they are only used later):

```
class Dataset(object):
    def __init__(self):
        self.data = None

    def store_data(self, raw_data):
        # process the data
        ...
        self.data = processed_data
```

Python OO Basics 2

- Don't confuse class and instance attributes.

```
class Platypus(Mammal):
    latin_name = "Ornithorhynchus anatinus"
```

This is a class attribute, it is shared across all instances.

- Use `super` to call a method from a superclass. `super(B, self)` starts the attribute lookup in the class "above" class B.

```
class Dataset(object):
    def __init__(self, data=None):
        self.data = data

class MRIDataset(Dataset):
    def __init__(self, data=None, parameters=None):
        # here has the same effect as calling
        # Dataset.__init__(self)
        super(MRIDataset, self).__init__(data)
        self.parameters = parameters

mri_data = MRIDataset(data=[1,2,3])
```

Python OO Basics 3

- *Special methods* start and end with two underscores and customize standard Python behavior (e.g. operator overloading).

```
class Dataset(object):
    def __len__(self):
        return 42

a = Dataset()
len(a)  # this is 42

class My2Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return My2Vector(self.x+other.x, self.y+other.y)

v1 = My2Vector(1, 2)
v2 = My2Vector(3, 2)
v3 = v1 + v2
```

Python OO Basics 4

- *Properties* allow you to add behavior to data attributes:

```
class My2Vector(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def set_x(self, x):
        self._x = x

    x = property(get_x, set_x)

    # define getter using decorator syntax
    @property
    def y(self):
        return self._y

v1 = My2Vector(1, 2)
x = v1.x  # use the getter
v1.x = 4  # use the setter
x = v1.y  # use the getter
```

OO Principles in Python

Python is a *dynamically typed* language, which means that the type of a variable is only known when the code runs (as opposed to *statically typed languages*, where it is known when you compile code).

- Python supports "duck typing" instead of strict type checking ("if it talks like a duck, walks like a duck..."). Important: document your code (e.g. which arguments can be passed to a function). However, there are legitimate use cases for explicitly checking the type of an object (using `isinstance`).

- Python relies on convention instead of enforcement. If you want to create a giant mess, Python isn't going to stop you.

- No attributes are really private, use a single underscore to signal that an attribute is for internal use only (encapsulation).

Python Example

```
import random

class Die(object):  # derive from object for new style classes
    """Simulate a generic die."""

    def __init__(self, sides=6):
        """Initialize and roll the die.

        sides -- Number of faces, with values starting at one (default is 6).
        """
        self._sides = sides  # leading underscore signals private
        self._value = None  # value from last roll
        self.roll()

    def roll(self):
        """Roll the die and return the result."""
        self._value = 1 + random.randrange(self._sides)
        return self._value

    def __str__(self):
        """Return string with a nice description of the die state."""
        return "Die with %d sides, current value is %d." % (self._sides, self._value)

class WinnerDie(Die):
   """Special die class that is twice as likely to return a 1."""

   def roll(self):
        """Roll the die and return the result."""
        super(WinnerDie, self).roll()  # use super instead of Die.roll(self)
        if self._value == 1:
            return self._value
        else:
            return super(WinnerDie, self).roll()
```

Python Example 2

```
>>> die = Die()
>>> die._sides  # we should not access this, but nobody will stop us
6
>>> die.roll
<bound method Die.roll of <dice.Die object at 0x03AE3F70>>
>>> for _ in range(10):
        print die.roll()


2 2 6 5 2 1 2 6 3 2
>>> print die  # this calls __str__
Die with 6 sides, current value is 2.
>>> winner_die = dice.WinnerDie()
>>> for _ in range(10):
        print winner_die.roll(),


2 2 1 1 4 2 1 5 5 1
>>>
```

Advanced Kung-Fu

Python OO might seem primitive at first. But the dynamic and open nature means that there is enough power to hang yourself.

Some Buzzwords to give you an idea:

- *Multiple inheritance* (deriving from multiple classes) can create a real mess. You have to understand the *MRO* (Method Resolution Order) to understand `super`.
- You can modify classes at runtime, e.g., overwrite or add methods ("monkey patching" or "duck punching").
- *Metaclasses* are derived from `type`, their instances are classes! They can be used to manipulate class creation.

...and there is more (descriptors, slots,...)

Try to avoid all of this unless you really need it! (KIS)

Functional programming in Python

Python also supports functional programming to some extend.

- Functions are first class objects, you can pass them around.

```
def func():
   print "test"

a = func
a() # prints "test"
```

- Python has *closures* (nested scopes).
  Functions can be embedded in functions and remember their context at the time of creation.

```
def get_func(text):
    def func():
        print text
    return func

a = get_func("aaa")
b = get_func("bbb")
a() # prints "aaa"
b() # prints "bbb"
```

Functional programming in Python 2

- *lambda* (anonymous functions), but limited to a single expression.

```
>>> square = lambda x: x**2
>>> square(2)
4
```

- The *decorator* syntax provides widely used syntactic sugar:

```
def say_hello(func):
    def wrapper(*args, **kwargs):
        print "Hello!"
        return func(*args, **kwargs)
    return wrapper

@say_hello
def square(x):
    return x**2

# decorator works like:
# square = say_hello(square)

x2 = square(2)  # this will print "Hello!"
```

Functional programming in Python 3

- Python supports some typical functional patterns like `map`:

```
>>> map(range(3), lambda x: x**2)
[0, 1, 4]
```

  But the preferred way in Python is to use the more explicit *list comprehensions* (or iterator comprehensions):

```
>>> [x**2 for x in range(3)]
[0, 1, 4]
```

- Python does not support tail-call optimization, so recursion is limited to a depth of around 1000.

Object Oriented Design Principles

OO Design

How do you decide what classes should be defined and how they interact?

- First of all realize that this is highly nontrivial!
  So take a step back and start with pen and paper.

- *OO design principles* tell you in an abstract way what a good oo design should look like.

- *Design patterns* are concrete solutions for reoccurring problems.
  They satisfy the design principles and can be used to understand and illustrate them.
  They provide a terminology to communicate effectively with other programmers

- Note that the classes and their inheritance in a good design often have no correspondence to real-world objects.

Design Principles

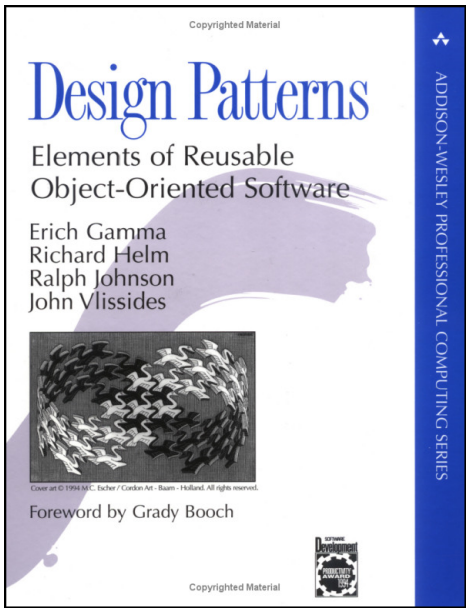
- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interface, not an implementation.
- Favor composition over inheritance.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension, but closed for modification. (Open-Closed Principle)
- A class should have only one reason to change. (Single Responsibility)

from "Head First Design Patterns"

Design Patterns

Origins

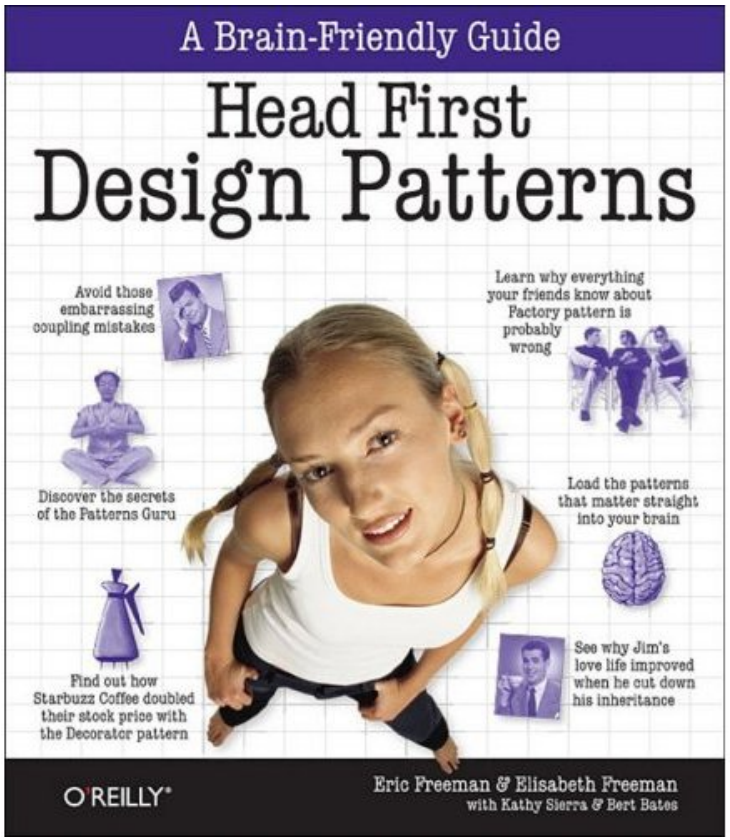
It started with this book (1995):
"Design Patterns. Elements of Reusable Object-Oriented Software."
(GoF, "Gang of Four")



Actually the idea came from a book on architecture.
Obviously identifying patterns is not limited to software...

Learning Design Patterns

Easier to read and more modern (uses Java):



Iterator Pattern

Description

- We want to iterate over different collections / containers of items. We call such a collection an *iterable*.
- We create an *iterator* object that manages the iteration (keeps track of where we are, which items have already been passed)
- The *iterator* has a `next()` method that returns an item from the collection. When all items have been returned it raises a `StopIteration` exception.
- The *iterable* provides an `__iter__()` method, which returns an *iterator* object.

Example

```
class MyIterable(object):
    """Example iterable that wraps a sequence."""

    def __init__(self, items):
        """Store the provided sequence of items."""
        self.items = items

    def __iter__(self):
        return MyIterator(self)

class MyIterator(object):
    """Example iterator that is used by MyIterable."""

    def __init__(self, my_iterable):
        """Initialize the iterator.

        my_iterable -- Instance of MyIterable.
        """
        self._my_iterable = my_iterable
        self._position = 0

    def next(self):
        if self._position < len(self._my_iterable.items):
            value = self._my_iterable.items[self._position]
            self._position += 1
            return value
        else:
            raise StopIteration()

    # in Python iterators also support iter by returning self
    def __iter__(self):
        return self
```

Example 2

```
iterable = MyIterable([1,2,3])

## perform the iteration manually:
iterator = iter(iterable)  # or use iterable.__iter__()
try:
    while True:
        item = iterator.next()
        print item
except StopIteration:
    pass
print "Iteration done."

## or use the Python for-loop:
for item in iterable:
    print item
print "Iteration done."

## iterator also supports iter:
iterator = iter(iterable)  # the old iterator has been used up!
for item in iterator:
    print item

## actually lists in Python are already iterables
for item in [1,2,3]:
    print item
```

Python Specifics

- Whenever you use a for-loop in Python you use the power of the iterator pattern!
  This is why they work with so many data types.
  (in Java this capability was only added later on)

- A typical use case in science is working with a huge data set (represented by an iterable) by loading and processing it in chunks (by iterating over it).

- For convenience iterators also have an `__iter__` method, but it is semantically different (returns `self`).
  This is a case where duck typing can be dangerous.
  Do not confuse iterables (which are collections) and iterators (they manage an iteration)!

- Python also has *generator functions* (using the `yield` keyword), *generator expressions*, and *generator objects*, so don't get confused.

Decorator Pattern

Starbuzz Coffee

```
class Beverage(object):

    # imagine some attributes like temperature, amount left,...

    def get_description(self):
        return "beverage"

    def get_cost(self):
        return 0.00

class Coffee(Beverage):

    def get_description(self):
        return "normal coffee"

    def get_cost(self):
        return 3.00

class Tee(Beverage):
    def get_description(self):
        return "tee"
    def get_cost(self):
        return 2.50

class CoffeeWithMilk(Coffee):

    def get_description(self):
        return super(CoffeeWithMilk, self).get_description() + ", with  milk"

    def get_cost(self):
        return super(CoffeeWithMilk, self).get_cost() + 0.30

class CoffeeWithMilkAndSugar(CoffeeWithMilk):

    # And so on, what a mess!
```

Second Attempt

```
class Beverage(object):

    def __init__(self, with_milk, with_sugar):
        self.with_milk = with_milk
        self.with_sugar = with_sugar

    def get_description(self):
        description = str(self._get_default_description())
        if self.with_milk:
            description += ", with milk"
        if self.with_sugar:
            description += ", with_sugar"
        return description

    def _get_default_description(self):
        return "beverage"

    def get_cost(self):
        cost = self._get_default_cost()
        if self.with_milk:
            cost += 0.30
        if self.with_sugar:
            cost += 0.20
        return cost

    def _get_default_cost(self):
        return 0.00

class Coffee(Beverage):

    def _get_default_description(self):
        return "normal coffee"

    def _get_default_cost(self):
        return 3.00
```

Analysis

Second solution already looks much cleaner than the first.

`_get_default_description` is a *factory method*,
it creates something and is overridden by subclasses
(this is another famous pattern).

But one monolithic class that depends on every little detail.
Hard to maintain (e.g. when adding soy milk as a new option).
Violates the open-closed principle.

Decorator pattern to the rescue!

Decorator

```
class Beverage(object):

    def get_description(self):
        return "beverage"

    def get_cost(self):
        return 0.00

class BeverageDecorator(Beverage):

    def __init__(self, beverage):
        super(BeverageDecorator, self).__init__()  # not really needed here
        self.beverage = beverage


class Coffee(Beverage):

    def get_description(self):
        return "normal coffee"

    def get_cost(self):
        return 3.00

class Milk(BeverageDecorator):

    def get_description(self):
        return self.beverage.get_description() + ", with  milk"

    def get_cost(self):
        return self.beverage.get_cost() + 0.30


coffee_with_milk = Milk(Coffee())
```

Notes

- Adding new ingredients like soy milk is now very easy and automatically works with all beverages.
- Anybody can define new custom ingredients without touching the original code (open-closed principle).
- There is no limit to the number of ingredients, this design scales very well.

The decorator pattern is very popular, for example the Java IO library is completely built around it.

Caution: Do not confuse the decorator pattern with the Python *decorator syntax* for the wrapping or modification of functions or classes (2.6) when these are defined (not at runtime, i.e. when the instances are defined).

Strategy Pattern

Duck Simulator

```
class Duck(object):

    def __init__(self):
        # for simplicity this example class is stateless

    def quack(self):
        print "Quack!"

    def display(self):
        print "Boring looking duck."

    def take_off(self):
        print "I'm running fast, flapping with my wings."

    def fly_to(self, destination):
        print "Now flying to %s." % destination

    def land(self):
        print "Slowing down, extending legs, touch down."


class RedheadDuck(Duck):

    def display(self):
        print "Duck with a read head."


class RubberDuck(Duck):

    def quack(self):
        print "Squeak!"

    def display(self):
        print "Small yellow rubber duck."
```

Problem

Oh snap! The `RubberDuck` is able to fly!

Looks like we have to override all the flying related methods.

But if we want to introduce a `DecoyDuck` as well we will have to override all three methods again in the same way (DRY).
And what if a normal duck suffers a broken wing?

Idea: Create a `FlyingBehavior` class which can be plugged into the `Duck` class.

Solution

```
class FlyingBehavior(object):
    """Default flying behavior."""

    def take_off(self):
        print "I'm running fast, flapping with my wings."

    def fly_to(self, destination):
        print "Now flying to %s." % destination

    def land(self):
        print "Slowing down, extending legs, touch down."


class Duck(object):

    def __init__(self):
        self.flying_behavior = FlyingBehavior()

    def quack(self):
        print "Quack!"

    def display(self):
        print "Boring looking duck."

    def take_off(self):
        self.flying_behavior.take_off()

    def fly_to(self, destination):
        self.flying_behavior.fly_to(destination)

    def land(self):
        self.flying_behavior.land()
```

Solution 2

```
class NonFlyingBehavior(FlyingBehavior):
    """FlyingBehavior for ducks that are unable to fly."""

    def take_off(self):
        print "It's not working :-("

    def fly_to(self, destination):
        raise Exception("I'm not flying anywhere.")

    def land(self):
        print "That won't be necessary."

class RubberDuck(Duck):

    def __init__(self):
        self.flying_behavior = NonFlyingBehavior()

    def quack(self):
        print "Squeak!"

    def display(self):
        print "Small yellow rubber duck."

class DecoyDuck(Duck):

    def __init__(self):
        self.flying_behavior = NonFlyingBehavior()

    def quack(self):
        print ""

    def display(self):
        print "Looks almost like a real duck."
```

Analysis

- If a poor duck breaks its wing we do:
  `duck.flying_behavior = NonFlyingBehavior()`
  Flexibility to change the behaviour at runtime!
- Could have avoided code duplication with inheritance (by defining a `NonFlyingDuck`), but with additional behaviors gets complicated (requiring multiple inheritance).
- Relying less on inheritance and more on composition
  (good according to the design principles).

Strategy Pattern

The *strategy* in this case is the flying behavior.
Strategy pattern in general means:

Encapsulate the different strategies in different classes.
Classes that use the strategy get a strategy object to which they delegate all the strategy calls.

Note that if the behavior only has a single method we can simply use a Python function! Therefore it is often said that the strategy pattern is "invisible" in Python.

In a scientific applications the strategy could for example be a certain classification method that is used by another program part (e.g., switch between Gaussian classifier and SVM).

Strategy Reloaded: State Pattern

Problem

We have an agent that interacts with an environment or other agents. This is handled through a public interface with several method.

The behaviour depends on some internal state:

```
class PirateAgent(object):

    ...

    def react_to_attack(self, attacker):
        if self.wounded and attacker.is_big:
            return "<run away>"
        else:
            return "Arrrrrhhhh!"
```

But with multiple states and multiple different methods this becomes messy.

Solution

Enter the *state pattern*:

Make use of the strategy pattern. For each possible internal state we have one strategy object. We can switch to another state and strategy at any time.

The state switch is generally under the control of the strategy / state, so the state might need a reference to the enclosing context.

Example: Use states to control agent behaviour.
See example for iterated prisoner's dilemma, file `statepattern.py`.

Closing Notes on Patterns

More on Patterns

Caution: Use patterns only where they fit naturally.
Adapt them to your needs (not the other way round).

Some other famous and important patterns:

- Factory Patterns
- Observer
- Singleton (can use some Python-Fu here)
- Adapter
- Composite

Combine patterns to solve complex problems.
The *Model-View-Controller* (MVC) pattern is the most famous example for such *compound patterns*.

Acknowledgements

The examples were partly adapted from
"Head First Design Patterns" (O'Reilly)
and
"Building Skills in Python"
`http://homepage.mac.com/s_lott/books/python`

The illustration images were downloaded from Google image search,
please inform if they infringe copyright to have them removed.