Object Oriented Design

Scientific Programming with Python

Roman Gredig
Universität Zürich

Universität Zürich[UZH]

Overview

- General Design Principles
- Object Oriented Programming in Python
- Object Oriented Design Principles and Patterns
- Design Pattern Examples

Based on the lecture slides of
Niko Wilbert
Advanced Scientific Programming in Python
Summer School 2013, Zurich

General Design Principles

The Problem of Scale

"Beyond a certain critical mass, a building becomes a BIG Building. Such a mass can no longer be controlled by a singular architectural gesture, or even by any combination of architectural gestures. The impossibility triggers the autonomy of its parts, which is different from fragmentation: the parts remain committed to the whole."

Rem Koolhaas in “Bigness and the Problem of Large”

Effective Software Design

KIS:

- Keep it simple: No Overengineering, no Spaghetti code.

DRY:

- Don't Repeat Yourself: Code duplication equals bug reuse.

Iterative Development:

- One cannot anticipate every detail of a complex problem.
- Start simple (with something that works), then improve it.
- Identify emerging patterns and continuously adapt the structure of your code.

Object Oriented Programming (in Python)

Object Oriented Programming

Objects

Combine state (data) and behavior (algorithms).

Encapsulation

Only what is necessary is exposed (public interface) to the outside. Implementation details are hidden to provide abstraction. Abstraction should not leak implementation details. Abstraction allows us to break up a large problem into understandable parts.

Classes

Define what is common for a whole class of objects, e.g.: "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.

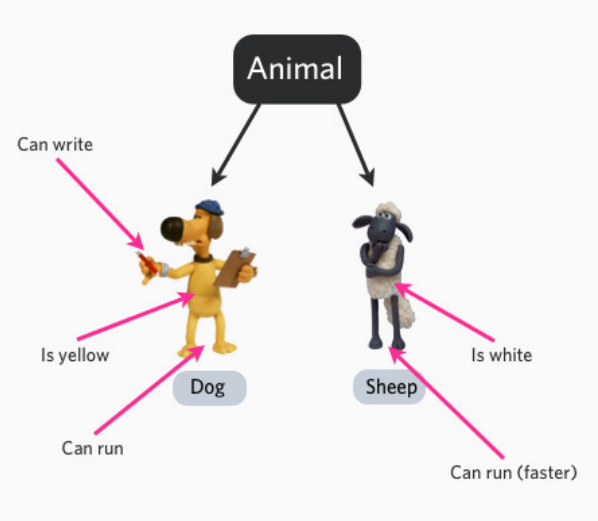
Object Oriented Programming

Inheritance

"a dog (subclass) is a mammal *(parent / superclass)*"
Subclass is *derived from / inherits / extends* a parent class.

Override parts with specialized behavior and extend it with additional functionality.

> Liskov substitution principle: What works for the parent class should also work for any subclass.



Object Oriented Programming

Polymorphism

Different subclasses can be treated like the parent class, but execute their specialized behavior.

Example: When we let a mammal make a sound that is an instance of the dog class, then we get a barking sound.

Object Oriented Programming

- Python is a *dynamically* typed language, which means that the type (class) of a variable is only known when the code runs.
- Duck Typing: No need to know the class of an object if it provides the required methods:
  “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”
- Type checking can be performed via the `isinstance` function, but generally prefer duck typing and polymorphism.
- Python relies on convention and documentation instead of enforcement. No enforced private attributes, use a single underscore to signal that an attribute is not intended for public use (encapsulation).

All classes are derived from object (new-style classes):

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

Python objects have data and function attributes (methods):

```
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 your data attributes first in `__init__`:

```
class Dataset(object):
    def __init__(self):
        self.data = None
    def store_data(self, raw_data):
    ... # process the data
        self.data = processed_data
```

Class attributes are shared across all instances:

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

Use `super` to call a method from a superclass:

```
class Dataset(object):
    def __init__(self, data):
        self.data = data

class MRIDataset(Dataset):
    # __init__ does not have to follow the Liskov principle
    def __init__(self, data, parameters):
        # here has the same effect as calling
        # Dataset.__init__(self, data)
        super().__init__(data)
        self.parameters = parameters

mri_data = MRIDataset([1,2,3], {'amplitude': 11})
```

Special / magic methods start and end with two underscores ("dunder") and customize standard Python behavior (e.g., operator overloading):

```
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
```

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):
        print("returning x, which is", self._x)
        return self._x

    def set_x(self, x):
        print("setting x to", x)
        self._x = x

    x = property(get_x, set_x)

v1 = My2Vector(1, 2)
x = v1.x # uses the getter, which prints the value
v1.x = 4 # uses the setter, printing the value
```

Helps with refactoring (can replace direct attribute access with a property).

Advanced Kung-Fu

There many advanced techniques that we didn't cover:

- *Multiple inheritance* (deriving from multiple classes) can create a real mess. Need to understand the MRO (Method Resolution Order) to understand super.
- Modify classes and objects at runtime, e.g., overwrite or add methods *(monkey patching)*.
- *Abstract Base Classes* enforce that derived classes implement particular methods from the base class.
- *Metaclasses* (derived from type), their instances are classes.

Great way to dig yourself a hole when you think you are clever.

> Try to avoid these, in most cases you would regret it. (KIS)

Object Oriented Design Principles and Patterns

How to do Object Oriented Design right?

- KIS & iterate:
  When you see the same pattern for the third time then it might be a good time to create an abstraction (refactor).
- Sometimes it helps to sketch with pen and paper.
- Classes and their inheritance often have no correspondence to the real-world, be pragmatic instead of perfectionist.
- Design principles tell you in an abstract way what a good design should look like (most come down to loose coupling).
- Testability (with unittests) is a good design criterium.
- Design Patterns are concrete solutions for reoccurring problems.

Some Design Principles

- One class, one single clearly defined responsibility.
- Principle of least knowledge:
  Each unit should have only limited knowledge about other units.
  Only talk to your immediate friends.
- Favor *composition* over inheritance.
  Inheritance is not primarily intended for code reuse, its main selling point is polymorphism.
  Ask yourself: “Do I want to use these subclasses interchangeably?”
- Identify the aspects of your application that vary and separate them from what stays the same.
  Classes should be “open for extension, closed for modification” *(Open-Closed Principle)*.
  You should not have to modify the base class.
- Minimize the *surface area* of the interface.
- Program to an interface, not an implementation. Do not depend upon concrete classes.

Examples of Design Patterns

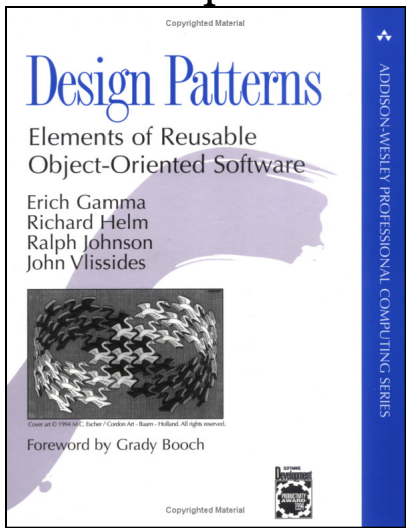
We'll now discuss three popular design patterns:

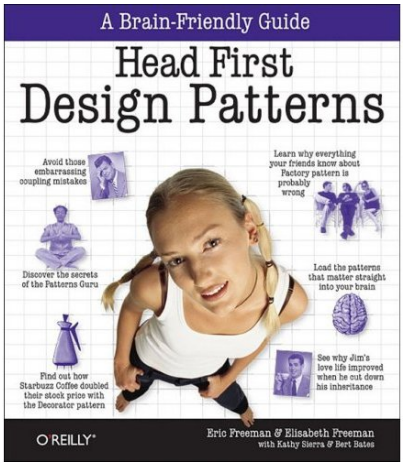
- Iterator Pattern
- Decorator Pattern
- Strategy Pattern

These are just three of many patterns, see for example:

- "Design Patterns. Elements of Reusable Object-Oriented Software." (1995), written by the "Gang of Four" (GoF).
- "Head First Design Pattens" (but uses Java)

Standard pattern names simplify communication between programmers!





Iterator Pattern

Problem

How would you iterate over elements from a collection?

A first (inept) attempt (imitating C code):

```
>>> my_collection = ['a', 'b', 'c']
>>> for i in range(len(my_collection)):
...         print(my_collection[i], end=" ")
a b c
```

But what if `my_collection` does not support indexing?

```
>>> my_collection = {'#x1': 'a', '#x2': 'b', '#y1': 'c'}
>>> for i in range(len(my_collection)):
...     print(my_collection[i], end=" ")
# What will happen here?
```

Should we have to care about this when all we want is iterate?

> Idea: Provide an abstraction for iteration handling that separates us from the collection implementation.

Description

What we want:

- Standard interface for collections that we can iterate over (we call these *iterables*)
- Iteration state (e.g., the position counter) should be decoupled form the container and should be encapsulated. Use an *iterator* object, which keeps track of an iteration and knows what the next element is.

What to implement:

The **iterator** has a `__next()__` method that returns the next 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):
    def __init__(self, items):
        """items -- List of items."""
        self.items = items

    def __iter__(self):
        return _MyIterator(self)

class _MyIterator(object):
    def __init__(self, my_iterable):
        self._my_iterable = my_iterable
        self._position = 0

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

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

Example II

Lets perform the iteration manually using this interface:

```
iterable = MyIterable([1,2,3])
iterator = iter(iterable) # or use iterable.__iter__()
try:
    while True:
        item = iterator.__next__()
        print(item)
except StopIteration:
    pass
print("Iteration done.")
```

...or just use the Python for-loop:

```
for item in iterable:
    print(item)
print("Iteration done.")
```

In fact, Python lists are already iterables:

```
for item in [1, 2, 3]:
    print(item)
```

Summary

- Whenever you use a for-loop in Python you use the power of the Iterator Pattern!
- Implement the iterable interface in your containers.
- The iterator has a single responsibility, while the iterable does not have to keep track of the iteration (which isn't its business).
- Note that `__iter__` is semantically different for iterables and iterators (duck typing fail!).

Use case: Processing huge data sets in manageable chunks that can come from different sources (e.g., from local disk or from the network).

Decorator Pattern

Starbuzz Coffe

```
class Beverage(object):
    # imagine some attributes like temperature,
    # amount left,...
    def get_desc(self):
        return "beverage"

    def get_cost(self):
        return 0.00

class Coffee(Beverage):
    def get_desc(self):
        return "coffee"

    def get_cost(self):
        return 3.00

class Tee(Beverage):
    def get_desc(self):
        return "tee"

    def get_cost(self):
        return 2.50
```

Adding Ingredients: First Try

```
class Beverage(object):
    def __init__(self, with_milk, with_sugar):
        self.with_milk = with_milk
        self.with_sugar = with_sugar

    def get_desc(self):
        desc = str(self._get_default_desc())
        if self.with_milk:
            desc += ", with milk"
        if self.with_sugar:
            desc += ", with sugar"
        return desc

    def _get_default_desc(self):
        return "beverage"

    # same for get_cost...

class Coffee(Beverage):
    def _get_default_desc(self):
        return "normal coffee" # and so on...
```

But what if we want more ingredients? Open-Closed principle?

Adding Ingredients: Second Try

```
class CoffeeWithMilk(Coffee):
    def get_desc(self):
        return (super().get_desc() +
            ", with milk")

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

class CoffeeWithMilkAndSugar(CoffeeWithMilk):
    # And so on, what a mess!
```

What we want:

- Adding a new ingredient like soy milk should not modify the original beverage classes.
- Adding new ingredients should be simple and work automatically across all beverages.

Solution: Decorator Pattern

```
class BeverageDecorator(Beverage):
    def __init__(self, beverage):
        self.beverage = beverage

class Milk(BeverageDecorator):
    def get_desc(self):
        return self.beverage.get_desc() +
            ", with milk"

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

coffee_with_milk = Milk(Coffee())
```

Composition solves the problem.

> **Note**: Do not confuse this with Python function decorators.

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", destination)

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

Duck Simulator II

```
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.")
```

Oh, snap! The `RubberDuck` has same flying behavior like a normal duck, must override all the flying related methods.

What if we want to introduce a `DecoyDuck` as well? (DRY)

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

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

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

class Duck(object):
    def __init__(self):
        self.flying_behavior = FlyingBehavior()
    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()

    # display, quack as before...
```

Solution II

```
class NonFlyingBehavior(FlyingBehavior):
    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()
    # different display, quack implementation...
```

Analysis

The *strategy* in this case is the flying behavior.

- 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`). Could make sense, but is less flexible.
- Relying less on inheritance and more on composition.

Strategy Pattern means:

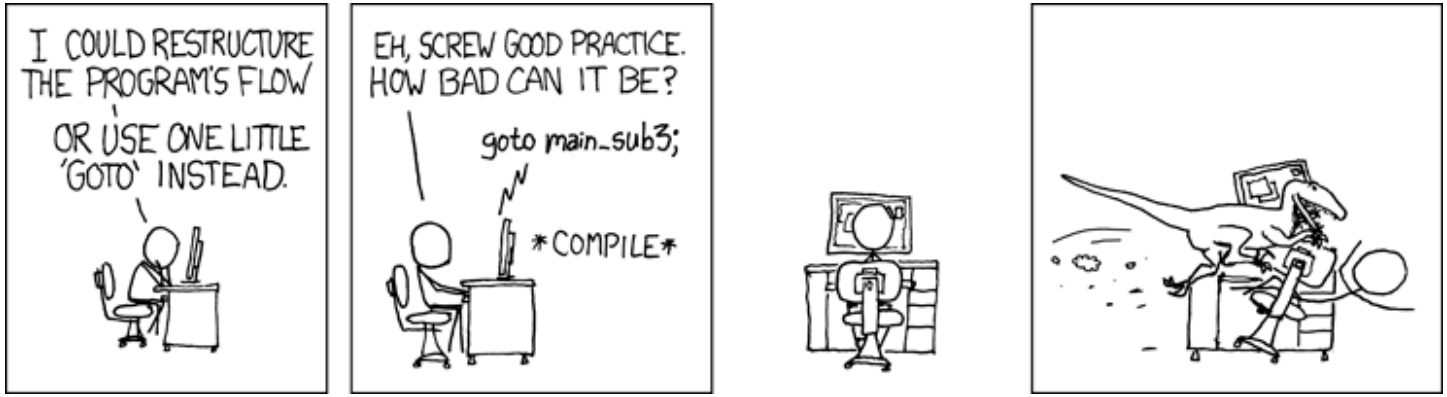
- **Encapsulate** the different strategies in different classes.
- Store a strategy object in your main object as a data attribute.
- **Delegate** all the strategy calls to the strategy object.

Acknowledgment

The examples were partly adapted from

- “Head First Design Patterns” (O’Reilly)
- “Design Patterns in Python” on youtube

99% of the slides are copied from Niko Wilbert



xkcd.com

Extra

Stop Writing Classes ?

Good reasons for not writing classes:

- A class is a tightly coupled piece of code, can be an obstacle for change.
  Complicated inheritance hierarchies hurt.
- Tuples can be used as simple data structures, together with stand-alone functions.
  Introduce classes later, when the code has settled.
- Functional programming can be very elegant for some problems, coexists with object oriented programming.

(see “Stop Writing Classes” by Jack Diederich)

Functional Programming

- Pure functions have no side effects.
  (mapping of arguments to return value, nothing else)
- Great for parallelism and distributed systems.
  Also great for unittests and TDD (Test Driven Development).
- It’s interesting to take a look at functional programming languages (e.g., Haskell) to get a fresh perspective.

Functional Programming in Python

Python supports functional programming to some extend:

- Functions are just objects, pass them around!

```
def get_hello(name):
    return "hello " + name


a = get_hello
print(a("world")) # prints "hello world"

def apply_twice(f, x):
    return f(f(x))

print(apply_twice(a, "world")) # prints "hello hello world"
```

Functional Programming in Python

- Functions can be nested and remember their context at the time of creation (closures, nested scopes).

```
def get_add_n(n):
    def _add_n(x):
        return x + n
    return _add_n

add_2 = get_add_n(2)
add_3 = get_add_n(3)
add_2(1) # returns 3
add_3(1) # returns 4
```