University of Zurich[UZH]

Department of Physics

Object-Oriented Programming

Scientific Programming with Python

Christian Elsasser

Based on talks by Niko Wilbert and Roman Gredig

Outline

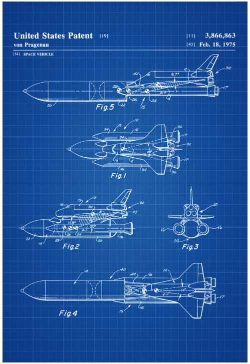
- What is OOP?
- Encapsulation & Inheritance
- Specialities in Python
- Design Patterns

Setting the scene

Object-oriented programming is a **programming paradigm**.

- Imperative programming
  - **Object-oriented**
  - Procedural
- Declarative programming
  - Functional
  - Logic

What is Object-Oriented Programming?



Aim to segment the program into objects/instances of different classes:

- **Instance variables** to describe the characteristic and state of the object
  - Fundamental types
  - Objects
- **Methods** to model the behaviour of the object

The definition of a class can be considered like a **blue print**. The program itself will invoke instances of classes and execute methods of these instances.

Why might OOP be a good idea?

DRY (Don't repeat yourself):

OOP means to **create the functionality of classes once** with the possibility to **use them repeatedly** in different algorithms.
In addition the inheritance in OOP means that we can easily create new classes acting as extensions based on existing classes (see below).

KIS (Keep it simple):

The OOP paradigm allows to split the functionality of programs into the **basic building blocks** and **the algorithm invoking them.** Thus it creates a natural structure within your code.

At one point the problem to solve becomes so complicated that a single sequence of program instructions are not sufficient to effectively maintain the code.

Example of a class

All classes are derived from `object`, even if this is not specified explicitly:

```
class Dog:
    pass
```

```
class Dog:
    def bark(self):
         print("Wuff!")
snowy = Dog()
snowy.bark() # first argument (self) is bound to this Dog instance
snowy.color = "yellow" # added attribute a to snowy
```

Always define your data attributes first in `__init__`:

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
```

Fundamental Principles of OOP (I)

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.

In Python:

- No explicit declaration of variables/ functions as private or public
- Usually parts of class that are supposed to be private with a starting underline _
- Python works with documentation and conventions instead of enforcement

Example of Encapsulation

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._sound = "Wuff!"

    def _open_mouth(self):
        pass

    def bark(self):
        self._open_mouth()
        print(self._sound)
```

The author of the class `Dog` does not want you to access explicitly the sound variable or the method to open the mouth.

Fundamental Principles of OOP (II)

Inheritance

- Possibility to define **new classes** as subclasses that are derived from / inherit / **extend a parent class**.
- Override parts with specialized behavior and extend it with additional functionality.

In Python:

- Possibility of inherit from one or multiple classes (latter one rather depreciated!!!)
- Invocation of parent methods with `super` function

Example of Inheritance

```python
class Mammal:
    def __init__(self):
        self._heart = "Bubum!"
    def heart_beat(self):
        print(self._heart)
    def make_sound(self):
        print("?")
class Dog(Mammal):
    def __init__(self):
        super(Dog,self).__init__()
    def make_sound(self):
        print("Bark!")

d = Dog()
d.make_sound()                # "Bark!"
d.heart_beat()                # "Bubum!"
super(type(d),d).make_sound() # "?"
```

- `super(Dog,self).__init__()` is the call to the parent constructor. Without this command the dog will not have a heart.
- `super` allows also to explicitly access methods of the parent class.

Fundamental Principles of OOP (III)

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.

In Python:

- 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.

Example of Polymorphism

```
def record_sound(mammal):
    _start_recording()
    if isinstance(mammal,Cat)
        print("No recording for you!")
    else:
        mammal.make_sound()
    _stop_recording()

d,c,b = Dog(),Cat(),Bear()
record_sound(d) # "Bark!"
record_sound(c) # "No recording for you!"
record_sound(b) # "Brum!"
```

- `record_sound` would work for all objects having a method `make_sound`, not just mammels.
- Dynamic typing make proper function overloading impossible!
- `isinstance(mammal,Cat)` is equivalent to `type(mammal)==Cat`.

Python Specialities – Magic Methods

```python
class Vector3D:
    def __init__(self,x,y,z):
        self.x,self.y,self.z=x,y,z
    def __add__(self,other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
v1 = Vector3D(1,2,3)
v2 = Vector3D(2,3,4)
v3 = v1 + v2          # (3,5,7)
```

- Magic methods (full list here) start and end with two underscores (“dunder”)
- They customise standard Python behavior (*e.g.* operator overloading)

Python Specialities – Function Decorators

```
class Vector3D:
    def __init__(self,x,y,z):
        self.x,self.y,self.z=x,y,z
    def _get_length(self):
        return (self.x**2+self.y**2
                +self.z**2)**0.5
    length = property(_get_length)

v1 = Vector3D(1,2,2)
v1.length              # 3.0
```

- `property` allow you to add behavior to data attributes.
- `property` has upto four variables:
  1. Getter
  2. Setter
  3. Deleter
  4. Documentation string

Advanced OOP Techniques

There many advanced techniques that we didn't cover:

- Multiple inheritance: Deriving from multiple classes; it can create a real mess. Need to understand the MRO (Method Resolution Order) to understand `super`.
- Monkey patching: Modify classes and objects at runtime, *e.g.* overwrite or add methods
- 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.
- **Testability** (with unittests) is a good design criterium.

How design principles can help:

- Design principles tell you in an abstract way what a good design should look like (most come down to loose coupling).
- Design Patterns are concrete solutions for reoccurring problems.

Some Design Principles

Scope of classes:

- **One class = one single clearly defined responsibility.**
- **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).

How to design interfaces:

- **Principle of least knowledge.** Each unit should have only limited knowledge about other units. Only talk to your immediate friends.
- Minimize the *surface area* of the interface.
- **Program to an interface**, not an implementation. Do not depend upon concrete classes.

University of Zurich[UZH]
Department of Physics

Design Patterns

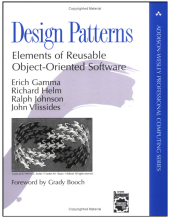
Purpose & background:

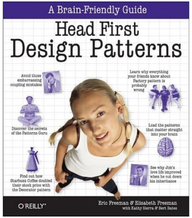
- Idea of concrete design approach for recurring problems.
- Closely related to the rise of the traditional OOP languages C++ and Java
- More important for compiled languages (Open-Closed principle stricter!) and those with stronger enforcement of encapsulation

Examples:

- **Decorator pattern**
- **Strategy pattern**
- Factory pattern
- ...

A comprehensive list can be found here.





Decorator Pattern

Decorator Pattern – Motivation

Challenge:

- How to modify the behaviour of an individual object . . .
- . . . and allowing for multiple modifications

Example: Implement a range of products of a coffee house chain

But what about the beloved add-ons?

```python
class Beverage:
    # 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 "tea"
        ...
```

Decorator Pattern – First try

Solution:

- Implementation via subclasses

Issue: Number of subclasses explodes to allow for multiple modifications (*e.g.* `CoffeeWithMilkAndSugar`)

```python
class Coffee(Beverage):
    def get_desc(self):
        return "coffee"
    def get_cost(self):
        return 3.00

class CoffeeWithMilk(Coffee):
    def get_desc(self):
        return "coffee with milk"
    def get_cost(self):
        return 3.20

class CoffeeWithSugar(Coffee):
    def get_desc(self):
        return "coffee with sugar"
        ...
```

Decorator Pattern – Second try

Solution:

- Implementation with switches

Issue: No additional add-ons implementable without changing the class (violation of the open-close principle!)

```python
class Coffee(Beverage):
    def __init__(self,withMilk,withSugar):
        self._withMilk = withMilk
        self._withSugar = withSugar
    def get_desc(self):
        desc = "coffee"
        if self._withMilk:
            desc += ", with milk"
        if self._withSugar:
            desc += ", with sugar"
        return desc
    def get_cost(self):
        price = 3.00
        if self._withMilk:
            price += 0.2
        if self._withSugar:
            price += 0.3
        return price
```

Decorator Pattern – Implementation

Solution:

- Create a class that is a beverage and wraps a beverage itself
- Possibility to create a chain of decorators
- Composition solves the problem
- Downside of implementation of all functions (some are potentially just fed through the decorator)

```
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())
```

Do not confuse the decorator pattern with Python's function decorators!

Strategy Pattern

Strategy Pattern – Motivation (I)

Let's implement a duck . . .

```
class Duck:
    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("Run fast, flap wings.")

    def fly_to(self, destination):
        print("Fly to", destination)

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

Strategy Pattern – Motivation (II)

. . . and different types of ducks!

Oh, no! The rubber duck does not fly! We need to overwrite all the methods about flying.

- What if we want to introduce a `DecoyDuck` as well?
- What if a normal duck suffers a broken wing?

⇒ It makes more sense to abstract the flying behaviour.

```
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.")
```

Strategy Pattern – Implementation (I)

- Create a class to describe the flying behaviour . . .
- . . . give `Duck` an instance of it . . .
- . . . and handle all the flying stuff via this instance

```python
class FlyingBehavior:
    def take_off(self):
        print("Run fast, flap wings.")
    def fly_to(self, destination):
        print("Fly to", destination)
    def land(self):
        print("Extend legs, touch down.")

class Duck:
    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...
```

Strategy Pattern – Implementation (II)

- Other example of composition over inheritance
- Encapsulation of function implementation in the strategy object
- Useful pattern to *e.g.* define optimisation algorithm at runtime.

```
class NonFlyingBehavior(FlyingBehavior):
    def take_off(self):
        print("It's not working :-(")
    def fly_to(self, destination):
        raise Exception("I'm not flying.")
    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...
```

Take-aways

- Object-oriented programming offers a powerful pradigm to structure your code.
- Inheritance and design principles and patterns allow to avoid repetitions (DRY).
- But do not overcomplicate things and ask always yourself if applying a particular functionality makes sense in the given context!

Extra

Stop Writing Classes?

There are 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

There are good reasons for not writing classes:

- 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, J) to get a fresh perspective.

Functional Programming in Python

Python supports functional programming to some extend:

- Functions are just objects, pass them around!
- Functions can be nested and remember their context at the time of creation (closures, nested scopes).

```python
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"

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
```