University of Zurich[UZH]

Department of Physics

Object-Oriented Programming

Scientific Programming with Python

Andreas Weiden

Based on talks by Niko Wilbert and Roman Gredig

Outline

What is OOP?

Fundamental Principles of OOP

Specialities in Python

Science Examples

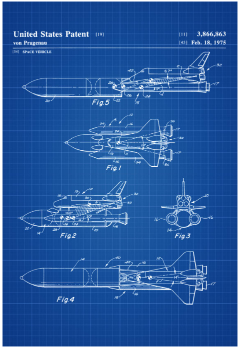
Design Patterns

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 instances of different classes of objects:

- **Instance variables** to describe the state of the object
- **Methods** to model the behaviour of the object

The definition of a **class** can be considered like a **blue print**. The program will create 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 programms.
In addition inheritance in OOP allows us to easily create new classes by extending 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 is not sufficient to effectively maintain the code.

Example of a class

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

    def make_sound(self):
        print("Wuff!")

# create an instance 'snoopy' of the class Dog
snoopy = Dog()

# first argument (self) is bound
# to this Dog instance
snoopy.make_sound()

# change snoopy's color
snoopy.color = "yellow"
```

- Started with `class` keyword.
- Methods defined as functions in class scope with at least one argument (usually called `self`).
- Special method `__init__` called when a new instance is created.
- Always define your data attributes first in `__init__`.

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 to break up a large problem into understandable parts.

In Python:

- No explicit declaration of variables/functions as private or public.
- Usually parts supposed to be private start with an underscore _.
- Python works with documentation and conventions instead of enforcement.

Example of Encapsulation

```python
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    def _change_mood(self, change):
        self._mood += change
        self.make_sound()

    def make_sound(self):
        if self._mood < 0:
            print("Grrrr!")
        else:
            print("Wuff!")

    def pat(self):
        self._change_mood(1)

    def beat(self):
        self._change_mood(-2)
```

- The author of the class `Dog` wants you to *pat* and *beat* the dog to change its mood.
- Do not use the `_mood` variable or the `_change_mood` method directly.

Fundamental Principles of OOP (II)

Inheritance

- 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:

- Inherit from one or multiple classes (latter one not recommended!)
- Invocation of parent methods with `super` function.
- All classes are derived from `object`, even if this is not specified explicitly.

Example of Inheritance

```
class Mammal:
    def __init__(self, color="grey"):
        self.color = color
        self._mood = 5

    def _change_mood(self, change):
        self._mood += change
        self.make_sound()

    def make_sound(self):
        raise NotImplementedError

    def pat(self):
        self._change_mood(1)

    def beat(self):
        self._change_mood(-2)
```

```
from mammal import Mammal

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

    def make_sound(self):
        if self._mood < 0:
            print("Grrrr!")
        else:
            print("Wuff!")
```

- `super().__init__(color)` is the call to the parent constructor.
- `super` allows also to explicitly access methods of the parent class.
- This is usually done when extending a method 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

```python
from animals import Dog, Cat, Bear

def caress(mammal, number_of_pats):
    if isinstance(mammal, Bear):
        raise TypeError("Bad Idea!")
    for _ in range(number_of_pats):
        mammal.pat()

d, c, b = Dog(), Cat(), Bear()
caress(d, 3) # "Wuff!" (3x)
caress(c, 3) # "Purr!" (3x)
caress(b, 3) # raises TypeError
```

- `caress` would work for all objects having a method `pat`, not just mammals.
- `isinstance(mammal, Bear)` checks if `mammal` is a bear.
- Dynamic typing makes proper function overloading impossible!

Python Specialities – Magic Methods

```python
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    def __repr__(self):
        return f"This is a {self.color} dog"

snowy = Dog("white")
print(snowy) # This is a white dog
```

- Magic methods (full list here) start and end with two underscores ("dunder").
- They customise standard Python behavior (*e.g.* string representation or operator definition).

Python Specialities – Property

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    def _get_mood(self):
        if self._mood < 0:
            return "angry"
        return "happy"

    def _set_mood(self, value):
        if not -10 <= value <= 10:
            raise ValueError("Bad range!")
        self._mood = value

    mood = property(_get_mood, _set_mood)

snowy = Dog("white")
print("Snowy is", snowy.mood) # Snowy is happy
snowy.mood = -3
print("Snowy is", snowy.mood) # Snowy is angry
```

- `property` has upto four arguments:
  1. Getter
  2. Setter
  3. Deleter
  4. Documentation string
- Access calculated values as if they were stored data attributes.
- Define read-only “data attributes”.
- Check if value assigned to “data attribute” fullfills conditions.
- Can also be used as a Python decorator.

Python Specialities – Property

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    @property
    def mood(self):
        if self._mood < 0:
            return "angry"
        return "happy"

    @mood.setter
    def mood(self, value):
        if not -10 <= value <= 10:
            raise ValueError("Bad range!")
        self._mood = value

# create an instance 'snowy' of the class Dog
snowy = Dog("white")
print("Snowy is", snowy.mood)
snowy.mood = 100
```

- `property` has upto four arguments:
  1. Getter
  2. Setter
  3. Deleter
  4. Documentation string
- Access calculated values as if they were stored data attributes.
- Define read-only “data attributes”.
- Check if value assigned to “data attribute” fulfils conditions.
- Can also be used as a Python decorator.

Python Specialities – Classmethods

```python
class Dog:
    def __init__(self, name, color="brown"):
        self.name = name
        self.color = color
        self._mood = 5

    @classmethod
    def from_string(cls, s):
        name, *color = s.split(",")
        if color:
            return cls(name, color)
        return cls(name)

snowy = Dog.from_string("snowy,white")
```

- A `classmethod` takes as its first argument a class instead of an instance of the class. It is therefore called `cls` instead of `self`.
- The method should return an object of the class.
- This allows you to write multiple constructors for a class, e.g.:
  - The default `__init__` constructor.
  - One constructor from a serialized string.
  - One that reads it from a database or file.
  - ...

Python Specialities – Class attributes

```python
class Dog:
    legs = 4
    all_dogs = set()

    def __init__(self, name, color="brown"):
        self.name = name
        self.color = color
        self._mood = 5
        Dog.all_dogs.add(self)

    def __repr__(self):
        return self.name

snowy = Dog("snowy", "white")
snowy.legs = 3
print(Dog.legs, snowy.legs) # 4 3
print(Dog.all_dogs) # {snowy}
```

- A class can also have attributes, not only an instance.
- All instances have (at initialization) the same attribute as the class.
- If you change the attribute, only the attribute of the instance changes.
- **Beware if the class attribute is mutable**! In this case inplace operations change the class attribute, which is visible in all instances. This can be a good or bad thing.

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 Method Resolution Order (MRO) 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)

Science Examples – Vector

```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)
    @property
    def length(self):
        return (self.x**2+self.y**2
                +self.z**2)**0.5
```

```python
from vector import Vector3D

v1 = Vector3D(0, 1, 2)
v2 = Vector3D(1,-3, 0)
v3 = v1 + v2
print(v3.length) # 3.0
```

- Variable type with optimized behaviour.
- Add custom functionality

Science Examples – Dataset

```
import numpy as np

class Dataset:
    mandatory_metadata = ["label", "color", "marker"]
    def __init__(self, datafile, **metadata):
        for key in self.mandatory_metadata:
            if key not in metadata:
                raise KeyError("Missing metadata", key)
        self.metadata = metadata
        self.data = np.loadtxt(datafile, delimiter=",")
        self.validate()

    def validate(self):
        if self.data.shape != (4, 10):
            raise ValueError("Bad shape of data.")

    @property
    def label(self):
        return self.metadata["label"]

    def peak_row(self):
        return self.data.max(axis=1).argmax()
```

```
from dataset import Dataset

ds = Dataset("data_0.csv",
        label="calibration",
        color="r",
        marker="+")
print(ds.label)
```

- Store additional info with data.
- Validate data on load.
- Calculated specific quantities.

Science Examples – Sensors

```python
from urllib.request import urlopen

class Sensor:
    def __init__(self, offset=0, scale_factor=1):
        self.offset = offset
        self.scale = scale_factor

    def get_value(self):
        return (self._get_raw()+self.offset)*self.scale

    def _get_raw(self):
        raise NotImplementedError

class WebSensor(Sensor):
    def __init__(self, url, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._url = url

    def _get_raw(self):
        res = urlopen(self._url)
        return float(res.read())
```

```python
from sensors import WebSensor

sensor = WebSensor(
        "https://crbn.ch/sensor", 273
        )
print(sensor.get_value())
```

- Store configuration with functionality.
- Allow sensors with different access methods.

Science Examples – Value with Uncertainty

```python
from numpy import sqrt

class UncertVal:
    def __init__(self, value, uncertainty=0):
        self.val = value
        self.sd = uncertainty

    def __str__(self):
        return f"{self.val} +/- {self.uc}"

    def add(self, other, corr=0):
        return UncertVal(self.val+other.val,
            sqrt(self.sd**2 + other.sd**2 +
                 2* self.sd * other.sd * corr))

    def __add__(self, other):
        return self.add(other)
```

```python
from uncertval import UncertVal

a = UncertVal(2, 0.3)
b = UncertVal(3, 0.4)
print(a+b) # 5 +/- 0.5
```

- Group several values.
- Manage access to values.
- Define operators respecting relations between values.

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 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. “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 (programming) 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.

Design Patterns

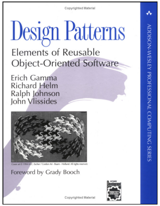
Purpose & background:

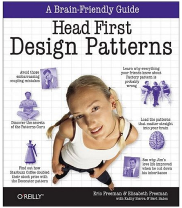
- Idea of concrete design approach for recurring problems.
- Closely related to the rise of the traditional OOP languages C++ and Java.
- More important for compiled languages (Open-Closed principle stricter!) and those with stronger enforcement of encapsulation.

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,...
    name = "beverage"
    cost = 0.00

    def __str__(self):
        return self.name

class Coffee(Beverage):
    name = "coffee"
    cost = 3.00

class Tea(Beverage):
    name = "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):
    name = "coffee"
    cost = 3.00

class CoffeeWithMilk(Coffee):
    name = "coffee with milk"
    cost = 3.20

class CoffeeWithSugar(Coffee):
    name = "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, milk=False, sugar=False):
        self._with_milk = milk
        self._with_sugar = sugar

    def __str__(self):
        desc = "coffee"
        if self._with_milk:
            desc += ", with milk"
        if self._with_sugar:
            desc += ", with sugar"
        return desc

    @property
    def cost(self):
        price = 3.00
        if self._with_milk:
            price += 0.20
        if self._with_sugar:
            price += 0.30
        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: Need to implement all functions (some are potentially just fed through the decorator).

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

class Milk(DecoratedBeverage):
    def __str__(self):
        return str(self.beverage) + ", with milk"

    @property
    def cost(self):
        return self.beverage.cost + 0.30

coffee_with_milk = Milk(Coffee())
```

Strategy Pattern

Strategy Pattern – Motivation (I)

Let's implement a duck . . .

```python
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 should 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

```
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, design principles and patterns allow to avoid repetitions (DRY).
- But do not overcomplicate things and always ask 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
```