



# Object-Oriented Programming

Scientific Programming with Python

Jonas Eschle

Based on slides by Andreas Weiden and talks by Niko Wilbert and Roman Gredig



This work is licensed under the [Creative Commons Attribution-ShareAlike 3.0 License](#).



## Outline

What is OOP?

Fundamental Principles of OOP

Specialities in Python

Science Examples

Design Patterns



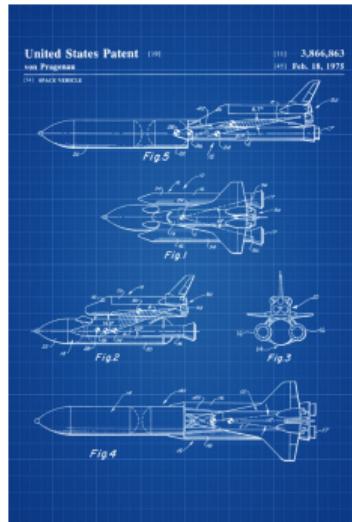
## Setting the scene

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 programs.

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

```
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 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__` is called when a new instance is created.
- ▶ Define your data attributes first in `__init__`.



## Fundamental Principles of OOP (I)

### Encapsulation

- ▶ Only expose **what is necessary** (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/methods as private or public.
- ▶ Conventionally, private parts start with an underscore `_`.
- ▶ Python works with **documentation** and **conventions** instead of enforcement.



## Example of Encapsulation

```
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 class of object if it provides the required methods: “If it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck.” (and we treat it as a duck)
- ▶ Type checking can be performed via the `isinstance` function, but generally prefer duck typing and polymorphism



## Example of Polymorphism

```
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` works for all objects having a method `pat`, not just mammals.
- ▶ `isinstance(mammal, Bear)` checks if `mammal` is a bear.
- ▶ Dynamic typing makes function overloading like in other languages impossible!



## Python Specialities – Magic Methods

```
class Dog:  
    def __init__(self, name, color="brown"):  
        self.name = name  
        self.color = color  
        self._mood = 5  
  
    def __repr__(self):  
        return f"{self.name}: {self.color} dog"  
  
snowy = Dog("snowy", "white")  
print(snowy) # snowy: 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"
        else:
            return "happy"

    mood = property(_get_mood)

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

- ▶ `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”.
- ▶ Preprocess value assigned to “data attribute”. (see later)
- ▶ Can also use special `@-syntax` (function 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"
        else:
            return "happy"

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

- ▶ `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”.
- ▶ Preprocess value assigned to “data attribute”. (see later)
- ▶ Can also use special `@`-syntax (function decorator).



## Python Specialities – Classmethods

```
class Dog:  
    def __init__(self, name, color="brown"):  
        self.name = name  
        self.color = color  
  
    @classmethod  
    def from_string(cls, s):  
        name, *color = s.split(",")  
        if not color or type(color) != str:  
            return cls(name)  
        return cls(name, color)  
  
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`.
- ▶ One usecase is 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

```
class Dog:
    breed = "dog"
    all_ = set()

    def __init__(self, name, color="brown"):
        self.name = name
        self.color = color
        type(self).all_.add(self)

    def __repr__(self):
        return f"{self.name}: {self.color} {self.breed}"

Dog("snowy", "white")
balto = Dog("balto")
balto.breed = "husky"
print(Dog.all_) # {snowy: white dog, balto: brown husky}
```

- ▶ A class can also have attributes that are shared among all its objects.
- ▶ If the attribute is modified, all objects will see this ("class global").
- ▶ **Pitfall assignment:** Assigning to an instance (`balto.breed = "husky"`), creates a new instance attribute, hiding the class one. You need the class to modify the class attribute  
(`type(balto).breed = "canis"`)



## Advanced OOP Techniques

There are 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

```
class Vector3D:  
    def __init__(self, x, y, z):  
        self.x, self.y, self.z = x, y, z  
  
    def __add__(self, other):  
        return type(self)(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  
  
    @length.setter  
    def length(self, length):  
        scale = length/self.length  
        self.x *= scale; self.y *= scale; self.z *= scale
```

```
from vector import Vector3D  
  
v1 = Vector3D(0, 1, 2)  
v2 = Vector3D(1,-3, 0)  
v3 = v1 + v2  
print(v3.length) # 3.0  
v3.length = 6  
print(v3.x, v3.y, v3.z)
```

- ▶ Variable type with optimized behaviour.
- ▶ Add custom functionality
- ▶ Note the property setter



## 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, has to be (4, 10).")

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

    def peak_row(self):
        10 July 2023
        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

```
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())
```

```
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

```
class UncertVal:  
    def __init__(self, value, uncertainty=0):  
        self.val = value  
        self.std = uncertainty  
  
    def __str__(self):  
        return f"{self.val} +/- {self.std}"  
  
    def add(self, other, corr=0):  
        variance = (self.std ** 2 + other.std ** 2  
                    + 2 * self.std * other.std * corr)  
        return type(self)(self.val + other.val,  
                         variance ** 0.5)  
  
    def __add__(self, other):  
        return self.add(other)
```

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

- ▶ Group several values.
- ▶ Add useful representation.
- ▶ Define operators respecting relations between values.



## Object-Oriented Design Principles and Patterns

### How to do Object-Oriented Design right:

- ▶ Rule of three: When you see the same functionality the third time it might be a good time to create a class (or function).
- ▶ 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

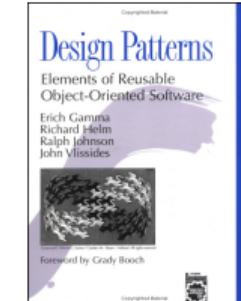
### Purpose & background:

- ▶ 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?

(Do not confuse the decorator pattern with function decorators!)

```
class Beverage:  
    # imagine some attributes like  
    # temperature, amount left,...  
    _name = "beverage"  
    _cost = 0.00  
  
    def __str__(self):  
        return self._name  
  
    @property  
    def cost(self):  
        return self._cost  
  
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`).

```
class Coffee(Beverage):  
    _name = "coffee"  
    _cost = 3.00  
  
class CoffeeWithMilk(Coffee):  
    _name = "coffee with milk"  
    _cost = 3.30  
  
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!).

```
class Beverage:  
    _name = "beverage"  
    _cost = 0.00  
  
    def __init__(self, milk=False, sugar=False):  
        self._milk = milk  
        self._sugar = sugar  
  
    def __str__(self):  
        desc = self._name  
        if self._milk:  
            desc += ", with milk"  
        if self._sugar:  
            desc += ", with sugar"  
        return desc  
  
    @property  
    def cost(self):  
        cost = self._cost  
        if self._milk:  
            cost += 0.30  
        if self._sugar:
```



## Decorator Pattern – Implementation

### Solution:

- ▶ Create a class that wraps a beverage and behaves like a beverage itself. (i.e. implements the beverage interface)
- ▶ Possibility to create a chain of decorators.
- ▶ Composition solves the problem.
- ▶ Downside: Need to implement all functions of beverage even if they do not need to be changed.

```
class Ingredient:  
    def __init__(self, beverage):  
        self.base = beverage  
  
class Milk(Ingredient):  
    def __str__(self):  
        return f"{self.base}, with milk"  
  
    @property  
    def cost(self):  
        return self.base.cost + 0.30
```



## 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 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 red 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 (flying strategy)...
- ▶ ... 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...
```



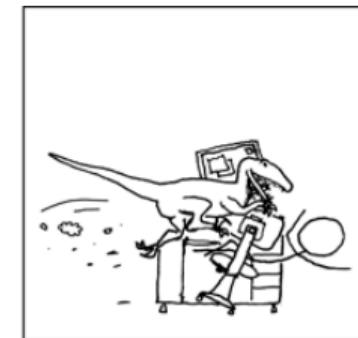
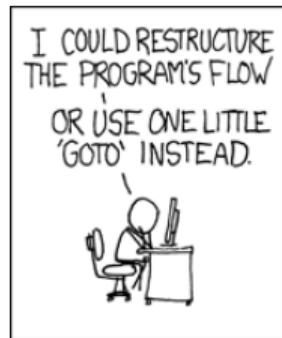
## Strategy Pattern – Implementation (II)

- ▶ 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 paradigm 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!





University of  
Zurich UZH

Department of Physics



Extra