

Carsten Knoll

Chair of Fundamentals of Electrical Engineering

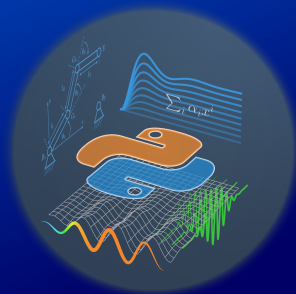
Python for Engineers

Pythonkurs für Ingenieur:innen

Object-Oriented Programming in Python
Objektorientierte Programmierung in Python

Dresden (Online), 2024-11-15

<https://tu-dresden.de/pythonkurs>
<https://python-fuer-ingenieure.de>



Programming Paradigms (Overview)

- desired: reusability of already implemented functionality
 - copy & paste?
 - frequent source of errors (forgetting some necessary changes)
 - later developments → changes in many places required → avoidable effort
- problem can be addressed with different **paradigms**

Programming Paradigms (Overview)

- desired: reusability of already implemented functionality
- copy & paste?
 - frequent source of errors (forgetting some necessary changes)
 - later developments → changes in many places required → avoidable effort

→ problem can be addressed with different **paradigms**

- What is a programming paradigm?
 - set of rules for formal and structural code design
 - so far in this course: “procedural programming”
 - here: “object-oriented programming”
 - also existing: “functional programming” (Lisp ♥ *recursive* functions calls)
- What is it used for?
 - support for creation of **good** code
 - suggest/prioritize a certain approach
- No dogma!
 - paradigm application depending on concrete problem (and taste)
 - unlike e.g. Java, Python does not enforce a certain paradigm
 - in Python: combinations of paradigms possible

Object-Oriented Programming (OOP)

- description of a complex system as an interaction of **objects**
- objects consist of
 - data ("**attributes**")
 - associated functions ("**methods**")
- objects are **instances** of a **class**
i.e. an object is the concrete variable, the class is the data type

object-orientation is a large field

- enough details for a whole semester
- here: only clarify most important terms and principles

Simple Example

Suppose, you own a personal soccer ball

- The ball has the properties (attributes)
 - radius
 - material
 - color
 - ...
 - (nouns)
- The ball provides certain actions (methods)
 - throw ball (in direction (x, y, z))
 - roll ball (in direction x, y)
 - ...
 - (verbs)



Object Orientation in Python

Your personal soccer ball is a particular sphere

Listing: example-code/ex01_sphere.py

```
class Sphere():
    """A class modeling a spherical objects"""

    def __init__(self, radius, midpoint=(0, 0, 0)):
        """
        Initialization method. Automatically executed when an object is created.
        Corresponds (approximately) to the 'constructor' in other programming
        languages.
        """

        # set attributes:
        self.radius = radius
        self.midpoint = midpoint

    def calc_volume(self):
        r = self.radius
        return (4/3)*np.pi*(r**3)
```

Instantiation

Other balls are spheres, too:

```
# Definition of the class
class Sphere():
    def __init__(self, radius, midpoint=(0,0,0)):
        ...

# instantiation of the class (create variables of that type)
# (arguments are passed to the `__init__` method)
soccer_ball = Sphere(radius=21)
tennis_ball = Sphere(3)
rubber_ball = Sphere(1)

print("radius of soccer:", soccer_ball.radius) # access to attribute
V = tennis_ball.calc_volume() # access to method
```

- class `Sphere` only is the “construction plan” or “blueprint”
- instantiation: creation of **concrete objects** according to the blueprint
- each object gets its own memory section
- attribute values are independent of each other
(each `sphere` instance has its own radius, center, ...)
- each object has unique memory address (readable with `id(..)`)

Inheritance (in general)

- creation of a new class based on an existing one
- only limited analogy to biological inheritance
- typical case: inheritance from the abstract to the specific
- representation: `base class` \leftarrow `child class` (" \leftarrow " $\hat{=}$ "inherits from")
- example: `animal` \leftarrow `mamal` \leftarrow `dog`
- child class has all attributes/methods of base class
 - value/implementation can be overridden
- additional attributes/methods possible in child class

What is inheritance good for?

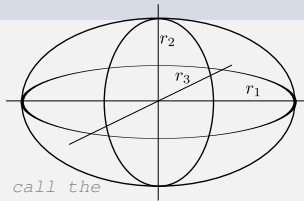
- sharing structure and code (attributes and methods)
 - reduces implementation effort
- documentation of similarities between classes

Inheritance (in Python)

The sphere is a special kind of ellipsoid

```
class Ellipsoid():  
    def __init__(self, r1, r2, r3):  
        ...
```

```
class Sphere(Ellipsoid):  
    def __init__(self, radius):  
  
        # In the constructor of child class we call the  
        # constructor of the parent class:  
  
        # Sphere is a Ellipsoid where all radii are equal  
        Ellipsoid.__init__(self, radius, radius, radius)
```



- class `Sphere` here is derived from class `Ellipsoid`
- attributes and methods are inherited (if not specified explicitly in the child class)
- constructor `__init__` is overridden so that it accepts only one radius

Inheritance Hierarchy

```
class GeometricObject: # (topmost) base class
    ...
class Cuboid(GeometricObject): # level 1 subclass
    ...
class Ellipsoid(GeometricObject): # level 1 subclass
    ...
class Sphere(Ellipsoid): # level 2 subclass
    ...
soccer_ball = Sphere(21) # instance
```

- base class `GeometricObject` implicitly is derived from `object` (builtin type)
- illustration with the help functions `isinstance(...)` und `issubclass(...)`

```
isinstance(soccer_ball, Sphere) # True
isinstance(soccer_ball, GeometricObject) # True
isinstance(soccer_ball, Cuboid) # False
issubclass(Sphere, Ellipsoid) # True
issubclass(Sphere, Cuboid) # False
```

Python-Speciality (1): `self`

- a **method** is a function belonging to an object
- when executed, it must be known to which **instance** this method belong
→ passing the instance as **implicit** first argument.
- Usually named (convention): `self`
- in other words:
`self` is placeholder for the concrete instance at a time when it does not yet exist

Listing: example-code/ex02_self.py

```
# Note: the id(...) function returns a unique identity-number for each object.
# Same number means: same object.

class ClassA():
    def m1(self):
        print(id(self))

    def m2(x): # !! self argument missing
        print(x)

a = ClassA() # create an instance

a.m1() # no explicit argument (but implicitly a is passed)
print(id(a)) # this gives the same number

a.m2() # no error (corresponds to print(id(a)), because x takes the role of self)
a.m2(123) # Error: too many arguments passed (a (implicitly) and 123 (explicitly))
```

Speciality (2): everything is an object

- no “primitive data types” (as in Java or C++)
- everything* is an object (i. e. an instance of a class):
 - numbers (instances of `int`, `float`, `complex`, ...)
 - strings (instances of `str`, `bytes`, ...)
 - functions and classes:

```
class ClassA():  
    pass  
  
def function1():  
    pass  
  
type(ClassA) # -> classobj  
type(function1) # -> function
```

- keywords, operators and other syntax elements are not objects!

```
type(while) # SyntaxError  
type(def) # SyntaxError  
type(class) # SyntaxError  
type(+) # SyntaxError
```

Summary

presented terms

- class
- instance (= object)
- attribute
- method
- constructor
- base class
- inheritance

presented Python constructs

- `class`,
`isinstance(...)`,
`issubclass(...)`, `self`,
`id(...)`, `type(...)`

other OOP related topics:

- multiple inheritance
- static methods
- operator overloading and "magic methods"
- ducktyping
- polymorphism
- encapsulation
- meta class programming
- difference between `__init__` and `__new__`
- class methods
- class variables
- data classes

OOP-related Links

- [official Python doc on classes, instances, `self` etc.](#)
- [demystifying `self`](#)
- [introductory blog post](#)