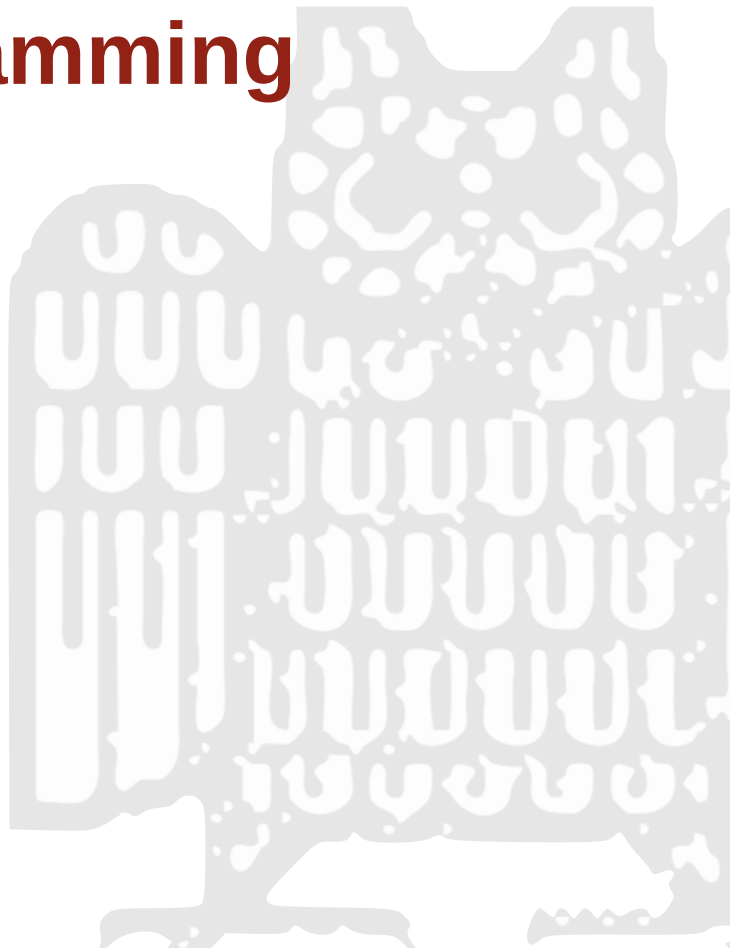
Introduction to Python Programming

12 – Object Orientation I

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23



Object-oriented Programming OOP

Procedural/imperative programming

- Turing complete

OOP: a different way of looking at things …

Many ways to look at OOP:

- Data structures & operations: what you can do with the data …
- Modularity and large scale-software engineering
- Data encapsulation …
- Look at the world in terms of objects and their relationships
- Taxonomy and inheritance

Recap: Imperative Programming

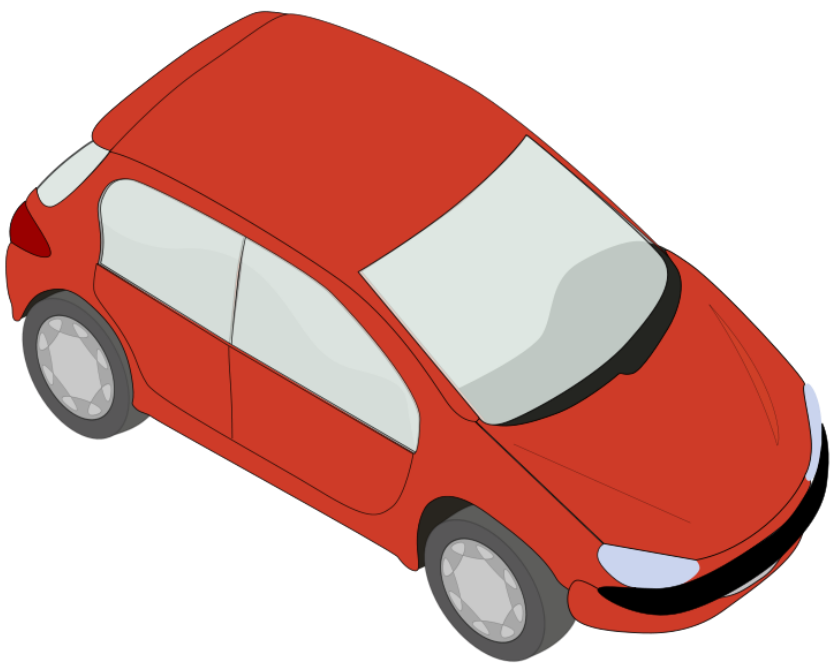
- Imperative paradigm: first do this, then do that.
- Control structures define the order in which computational steps are executed.
  - What could be a computational step in Python?
  - Which control structures do you know?
- State of the program changes as a function of time.
- Commands can be grouped into procedures/functions.

Object-Oriented Programming: OOP

- Classes, objects and methods
- Classes describe concepts (data and/or operations) of the domain of interest
- Classes are abstract blueprints
- Objects are instantiations of classes
- Methods are functions associated with objects/classes
- Object-orientation: Send messages between objects to simulate the temporal evolution of a set of real-world phenomena.

Object-Oriented Programming

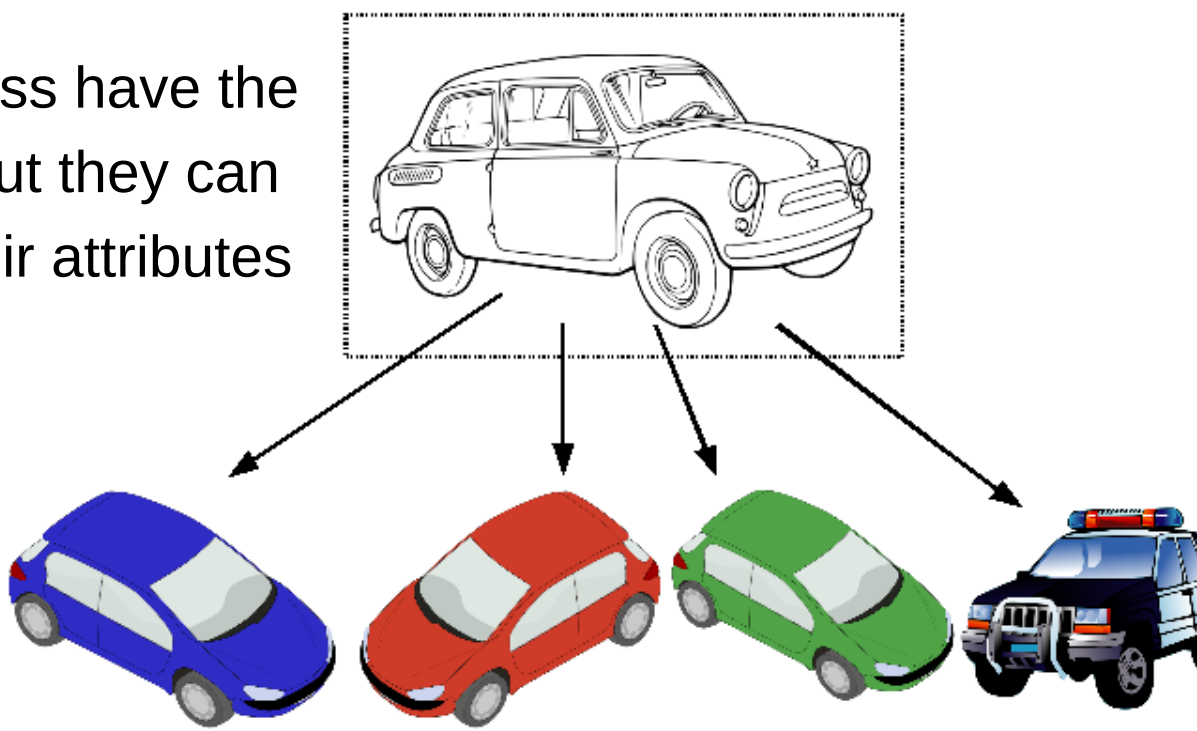
- Real-life objects modelled as software objects.
- Objects combine characteristics (attributes/features) and behaviors (methods).



car
ATTRIBUTES: color motor … METHODS: start drive stop …

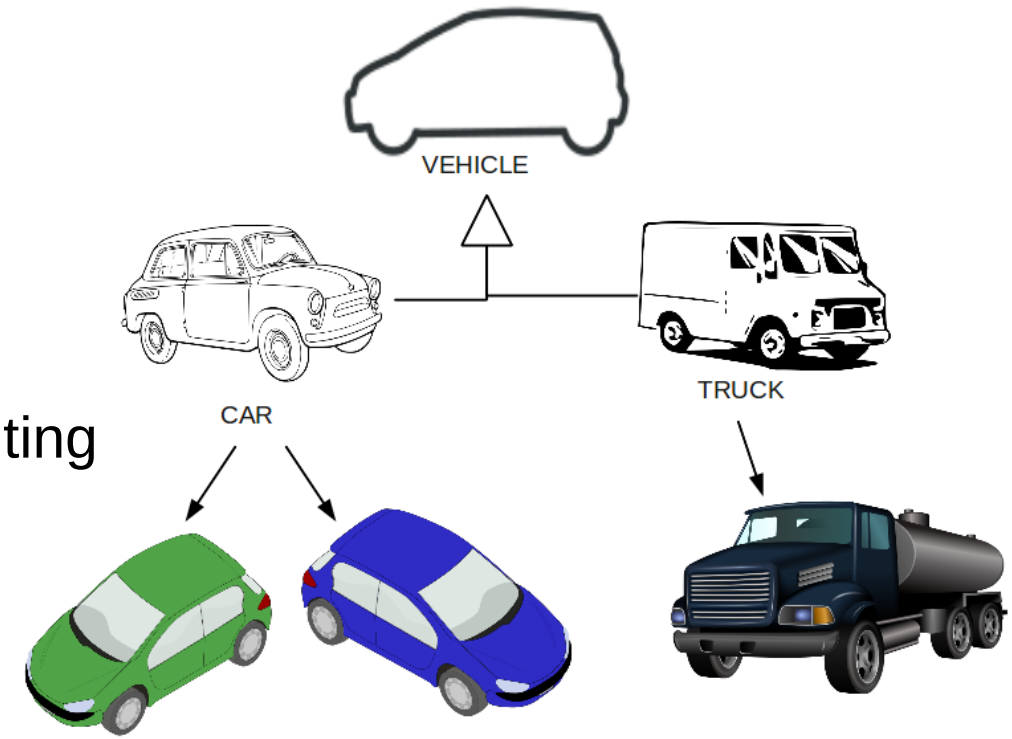
Classes = Blueprints

- Classes are blueprints/designs for objects.
- Creating objects using classes: objects instantiate a class.
- Objects are also called instances of a class.
- Objects of the same class have the same basic structure, but they can differ in what values their attributes have.



Inheritance

- Different classes can share characteristics / behaviors.
- A class hierarchy …
- Inheritance saves us from writing the same code over and over again.



OOP in Python

Let’s look at the details

... and how it’s done in Python.

Software objects to model real-life objects

	account #1	account #2
number	1	2
holder	„Timo“	„Stefan“
balance	200	1000

Attributes

- describe aspects of the object
- contain the data of an object
- may change over time

Classes = Blueprints for Objects

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def print_info(self):
        print('Balance:', self.balance)
```

	account #1	account #2
number	1	2
holder	„Timo“	„Stefan“
balance	200	1000

Classes = Blueprints for Objects

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def print_info(self):
        print('Balance:', self.balance)
```

A constructor …

Methods …

Classes = Blueprints for Objects

```
class Account:
    def __init__(self,
        self.number = n
        self.holder = h
        self.balance =

    def deposit(self,
        self.balance +=

    def withdraw(self,
        self.balance -= amount

    def print_info(self):
        print('Balance:', self.balance)
```

```
>>> a = Account(1, "Stefan")
>>> a.print_info()
Balance: 0
>>> a.deposit(100)
>>> a.print_info()
Balance: 100
>>> a.withdraw(50)
>>> a.print_info()
Balance: 50
>>>
```

Creating an object a from the class Account

Computing with/ up-dating a using methods

Classes = Blueprints for Objects

Instance methods

- operate on objects that have been created from this class, code to manipulate / use the object's attributes
- first parameter is the object itself
- called "self" (by convention)

```
class Ac
    def _
        se
        se
        se

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def print_info(self):
        print('Balance:', self.balance)
```

Initialization

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def deposit(self,
        self.balance +=

    def withdraw(self,
        self.balance -=

    def print_info(sel
        print('Balance:
```

`__init__(self, ...)`

- is called automatically right after an object has been created
- sets initial or default values of an object: the instance attributes
- … called a constructor (method)

Manipulate attributes only via instance methods!

- Instance attributes ("self.xxx") can be accessed "from outside the class"

```
>>> a = Account(2, "Stefan")
>>> a.balance -= 1000
```

Bad …!

- … but this is generally considered bad style.
- Attributes of an object should only be modified using code that was written *within* the class definition: that's the methods that come with the class …!
- So use methods defined with the class/object to manipulate attribute values

Programming style/hygene: manipulate attributes only via instance methods!

- Instance attributes ("self.xxx") can be accessed "from outside the class"

```
>>> a = Account(2, "Stefan")
>>> a.withdraw(1000)
```

Much better! Use method to update attribute!

- … but this is generally considered bad style.
- Attributes of an object should only be modified using code that was written *within* the class definition: thats the methods that come with the class …!
- So use methods defined with the class/object to manipulate attribute values

Programming style/hygene: manipulate attributes only via instance methods!

- Refining the methods … to what you want them to do …
- As an example:
  - Account restriction: balance may not be negative
  - Let's assume that my balance is $1000, and I want to withdraw $1500
  - If teller set my balance to -$500, the branch manager would not be very happy!
- We also use this example to illustrate Data Encapsulation below.
- Data Encapsulation/Protection = part of the modularity of OOP.

Data Encapsulation

```
class Account:
   ...
   def withdraw(self, amount):
      if self.balance < amount:
         amount = self.balance
      self.balance -= amount
      return amount
   ...

a = Account(2, "Stefan")
a.deposit(1000)
...
cash = a.withdraw(1500)
print("Oh, I only got", cash)
```

One of the methods …

A refinement of the previous `withdraw()` method …

Data Encapsulation

```
class Account:
   ...
   def set_holder(self, holder):
      if not isinstance(holder, str):
         raise TypeError
      self.holder = holder
   ...
```

Provide a setter (set_...) method for each attribute that may otherwise have to be changed from the outside (bad style …)

- Allows for validation

Original Code

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def print_info(self):
        print('Balance:', self.balance)
```

Original Code, with Setter Methods

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def set_deposit(self, amount):
        self.balance += amount

    def set_withdraw(self, amount):
        self.balance -= amount

    def print_info(self):
        print('Balance:', self.balance)
```

Original Code, with Setter/Getter Methods

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def set_deposit(self, amount):
        self.balance += amount

    def set_withdraw(self, amount):
        self.balance -= amount

    def get_print_info(self):
        print('Balance:', self.balance)
```

Data Encapsulation – Coding Style

- Assign values to attributes only via instance methods (setters) or the constructor.
- Modify the values of attributes only via (setter) methods.
- Accessing (reading) the value of instance attributes from “outside“ is okay-ish …
  - e.g., `print(stefansAccount.balance)`
- Better with getter method: `stefansAccount.get_print_info()`

Data Encapsulation – Coding Style

- Actually, we can completely hide/shield instance attributes from the outside … often, that‘s exactly what we want to do.
- Make sure data stored in object created from a class cannot be changed accidentally from outside, but only through explicit use of instance methods
- The way we do that is we change any instance variable / attribute / feature from
  - `self.balance`

  to
  - `self.__balance`
- everywhere in the constructor method and instance methods

Data Encapsulation: Before

```
class Account:
    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0

    def set_deposit(self, amount):
        self.balance += amount

    def set_withdraw(self, amount):
        self.balance -= amount

    def get_print_info(self):
        print('Balance:', self.balance)
```

Data Encapsulation: After

```
class Account:
    def __init__(self, number, holder):
        self.__number = number
        self.__holder = holder
        self.__balance = 0

    def set_deposit(self, amount):
        self.__balance += amount

    def set_withdraw(self, amount):
        self.__balance -= amount

    def get_print_info(self):
        print('Balance:', self.__balance)
```

Data Encapsulation: After

```
class Account:
    def __init__(self, number, holder):
        self.__number = number
        self.__holder = holder
        self.__balance = 0

    def set_deposit(self, amount):
        self.__balance += amount

    def set_withdraw(self, amount):
        self.__balance -= amount

    def get_print_info(self):
        print('Balance:', self.__balance)
```

None of these work ….!

```
>>> a = Account(2, "Stefan")
>>> a.balance -= 1000

>>> a = Account(2, "Stefan")
>>> a.__balance -= 1000
```

But this does ….!

```
>>> a = Account(2, "Stefan")
>>> a.set_withdraw(1000)
```

General Issues: Class Design

- How can I describe the state of my object?
  ⇒ instance attributes.
- What do I know about the object before/when creating it
  ⇒ initializer/constructor method (`__init__`)
- What operations that change/access the object's attribute values will be performed on the object?
  ⇒ methods: `set_` … `get_` …
- How do I do full or partial data encapsulation?
  ⇒ instance attributes/variables: `self.__xxx` …

Hooks

```
class Account:
  def __str__(self):
     res = "Account ID:" + str(self.__number) + "\n"
     res += "Holder:" + self.__holder + "\n"
     res += "Balance: " + str(self.__balance) + "\n"
    return res
  ...
```

- Hooks = methods that are executed automatically by Python in particular circumstances
  - e.g., `__init__` and `__str__`
- For instance, `__str__` is called when an object is printed
  - e.g., `print(stefansAccount)`

Exercise #1 (see Moodle)

- Put together a file containing the complete Account class and create a main application where you create a number of accounts.
- Play around with depositing / withdrawing money.
- Change the withdraw function such that the minimum balance allowed is -1000.
- Write a method `apply_interest(self)` which applies an interest rate of 1.5% to the current balance and call it on your objects.

Exercise #2 (see Moodle)

- Write the complete code for an Employee class, including `__init__`, `__str__`, ...
- Create a few employee objects and show how you can manipulate them using the methods.