Object-oriented Programming II

Barbara Plank

Symbolische Programmiersprache
MaiNLP / CIS LMU

WS2022/2023

Credits

These slides were extended and adapted from earlier versions created by a.o. Katerina Kalouli, Annemarie Friedrich, Benjamin Roth

Outline

Recap

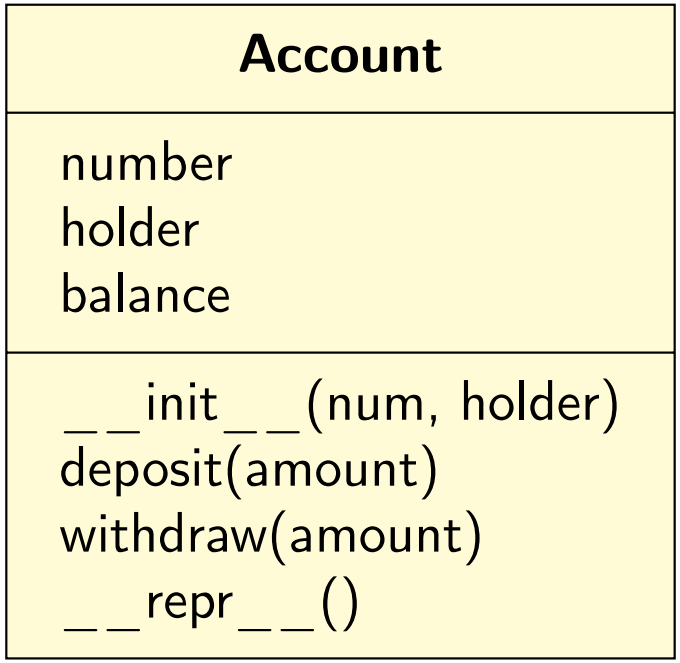
Recap: software objects / real-life objects

Attributes/Instances	`annesAccount`	`stefansAccount`
number	1	2
holder	'Anne'	'Stefan'
balance	200	1000

Attributes

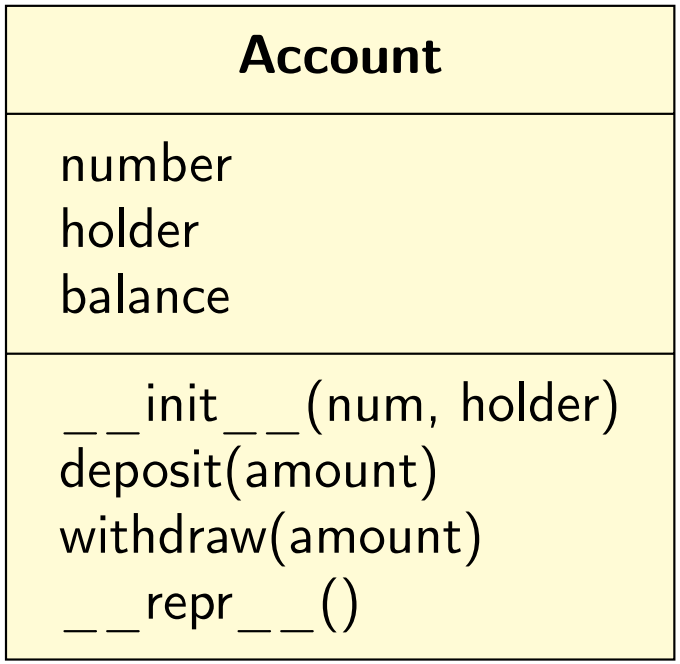
- describe the *state* of the object
- contain the *data* of an object
- can change with time

Recap: UML (Unified Modeling Language) diagram



Recap: Constructor, Attributes and Methods

```
1  class Account:
2    def __init__(self, num, holder):
3      self.num = num
4      self.holder = holder
5      self.balance = 0
6
7    def deposit(self, amount):
8      self.balance += amount
9
10   def withdraw(self, amount):
11     if self.balance < amount:
12         amount = self.balance
13     self.balance -= amount
14     return amount
15
16   def __repr__(self):
17     return "Account {} Holder: {} Balance: {}"\
18           .format(self.num, self.holder, self.balance)
```



Recap: Constructor / Initialization Methods

- The constructor is called for the creation of a new object (*self* points to the new object)
- *self.num*, *self.holder*: attributes of the object
- Note: *num* und *holder*: would be local variables of the method

```
1  class Account:
2    # Constructor (here object requires 2 arguments)
3    def __init__(self, num, holder):
4      self.num = num
5      self.holder = holder
6
7  annesAcc = Account(1, "Anne")
8  stefansAcc = Account(2, "Stefan")
9  bensAcc = Account() # error
```

Recap: Constructor / Initialization Methods

- Note that the following code is bad practice
- Instead use functions to manipulate objects and use the constructor.
- For example, expose object attributes via getter and setter methods (cf. lecture 2), better: python properties (next slide)

```
1  from accounts import Account
2
3  if __name__ == "__main__":
4      annesAcc = Account()
5      annesAcc.balance = 200 #bad
```

New: Python Properties

- By convention you can hide attributes from user by declaring them as private (self._balance).
- Expose class attributes through "getters" (@property) and "setters" (@method.setter), allows for input validation.

```
1   class Account:
2     def __init__(self):
3       self._balance = 0 # private attribute
4
5     # GETTER AND SETTER
6     @property
7     def balance(self):
8       return self._balance
9
10    @balance.setter
11    def balance(self, amount):
12      self._balance += amount
13
14  if __name__ == "__main__":
15    annesAcc = Account()
16    annesAcc.balance = 200 # safe
17    annesAcc._balance = 0 # works, but bad practice
```

Object Types

Recap: Object Types

All values in Python have types, also objects:

- 1.5: type float
- 'Stefan': type str
- the type of the instance/object *stefansAcc* is the class from which the object originates

```
>>> from accounts import Account
>>> stefansAcc = Account(2, "Stefan")
>>> type(stefansAcc)
<class 'accounts.Account'>
```

Python Modules

Python Modules

- modules: files with Python-Code, e.g., with functions, variables, classes
- modules: group together code that belongs together $\Rightarrow$ better comprehensibility
- modules: also objects; contain references to the function/class objects that the module defines; functions/classes can be imported from one module into another
- module name = file name without .py

```
1  # import all functions / classes from <modulename>
2  import modulename
3  # import all functions and rename as
4  import modulename as mn
5  # import specific function
6  from modulename import somefunction
7  # import specific class
8  from modulename import someclass
```

Python Modules

- when a module is imported, its methods and classes are available
- execute a module: `python3 someModule.py`
- *someModule* can also be imported from other files (modules)
- if-Statement checks whether this module should be executed as the main module ⇒ can write the tests for the module here
- without "main", any code when importing a module is executed

```
1  # imports
2
3  # some more function / class definitions
4
5  # main
6  if __name__ == "__main__":
7      # this code is executed when executing THIS module
8      # from the shell, i.e., python3 modulename.py
```

Project Structure

- be aware of naming modules (overwriting existing python modules)
- classes can be imported in the main application:
- `from modulename import classname`

```
1  from accounts import Account
2
3  if __name__ == "__main__":
4      annesAcc = Account(1, "Anne")
5      annesAcc.deposit(200)
```

Motivation

- Today we will learn about inheritance and composition in Python
- These are concepts about **relationships** of two classes
- Important concepts in object-oriented programming, as they help write less redundant and more reusable code

Inheritance

Inheritance

- Inheritance models an **is-a** relationship.
- If a Derived class inherits from a Base class, this means we created a relationship where Derived is a specialized version of Base.
- Classes from which other classes are derived are called super classes.

```
1  # Parent class
2  class Parent :
3              # Constructor
4              # Variables of Parent class
5              # Methods of Parent
6
7  # Child class inheriting Parent class
8  class Child(Parent) :
9              # constructor of child class
10             # variables of child class
11             # methods of child class
```

Src: https://www.geeksforgeeks.org/python-oops-aggregation-and-composition/

Inheritance: UML Diagram

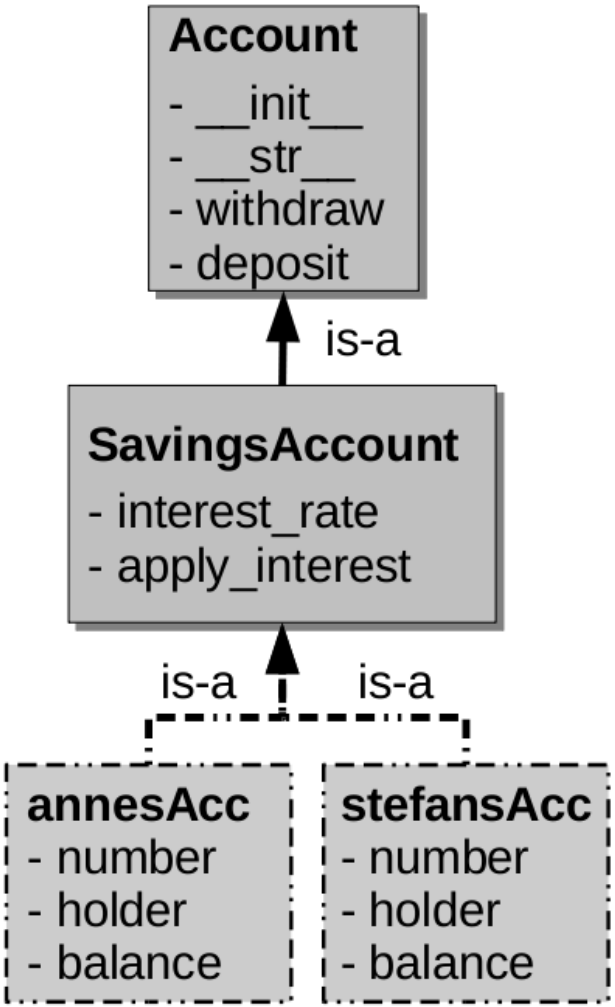
Inheritance: bank example

- **savings account**: for each account we store the account number, the account holder and the account balance. The account balance should be ≥ 0. An interest rate, which is defined for all saving accounts, can also be applied. Money can be deposit into the account. The account statement, which can be printed, contains the account number, the account holder and the account balance.
- **checking account**: for each account we store the account number, the account holder and the account balance. The account balance should lie within the credit limit defined for each client. If Anne's credit limit is \$500, then her account balance can be a minimum of -\$500. Money can be deposit into the account. The account statement, which can be printed, contains the account number, the account holder and the account balance.

Similarities/General Functionality: Parent Class

```
1  class Account:
2  " a class providing general functionality for accounts"
3  # CONSTRUCTOR
4    def __init__(self, num, person):
5      self.balance = 0
6      self.number = num
7      self.holder = person
8  # METHODS
9    def deposit(self, amount):
10     self.balance += amount
11   def withdraw(self, amount):
12     if amount > self.balance:
13           amount = self.balance
14           self.balance -= amount
15           return amount
16   def __str__(self):
17     res = ...
18         return res
```

Special Cases: Child Classes



- `SavingsAccount` "is based on" `Account`
- methods of the parent class are available in the child class
- (Note: `__str__` is like `__repr__`)

```
1 annesAcc = SavingsAccount(1, "Anne")
2 annesAcc.deposit(200)
3 annesAcc.apply_interest()
4 print(annesAcc)
```

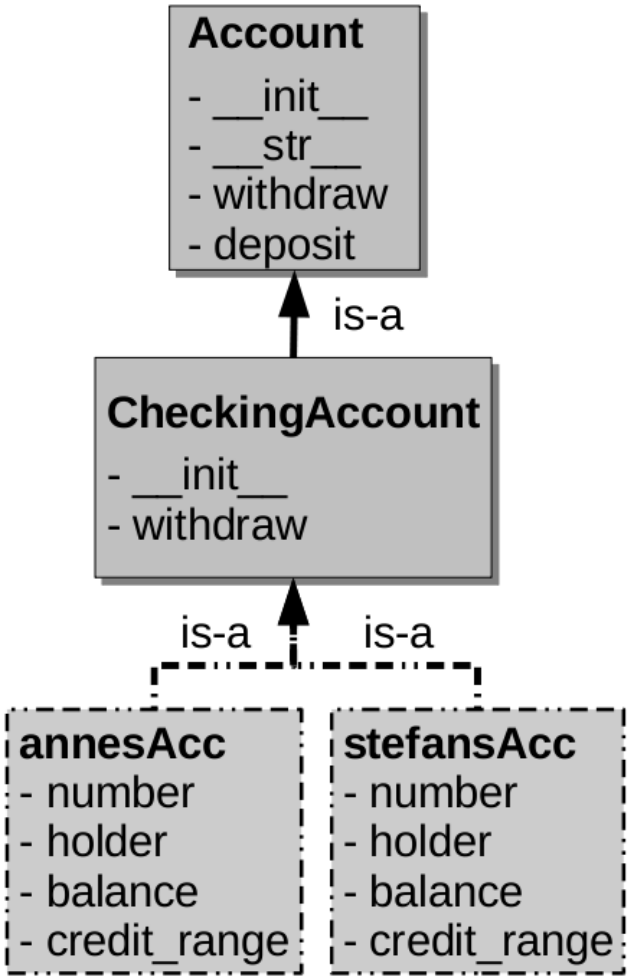
Special Cases: Child Classes

- SavingsAccount is based on Account
- SavingsAccount is derived from Account
- SavingsAccount extends Account
- SavingsAccount provides further functionalities:

```
1  class SavingsAccount(Account):
2  ''' class for objects representing savings accounts.
3  shows how a class can be extended. '''
4      # ATTRIBUTES
5      interest_rate = 0.035
6      # METHODS
7      def apply_interest(self):
8          self.balance *= (1+SavingsAccount.interest_rate)
```

Method Overwriting/ Polymorphism/Redundancy

Overwriting Methods



- the `Account` class also has the `__init__` and `withdraw` methods
- the `CheckingAccount` class overwrites them

```
1 annesAcc = CheckingAccount
2 (1, "Anne", 500)
3 annesAcc.deposit(200)
4 annesAcc.withdraw(350)
5 print(annesAcc)
```

Which methods get called on annesAcc? `deposit`, `withdraw`

Polymorphism

- method is called on the object $\rightarrow$ what happens depends on class hierarchy, i.e., one (same) call, different behaviour possible

Example

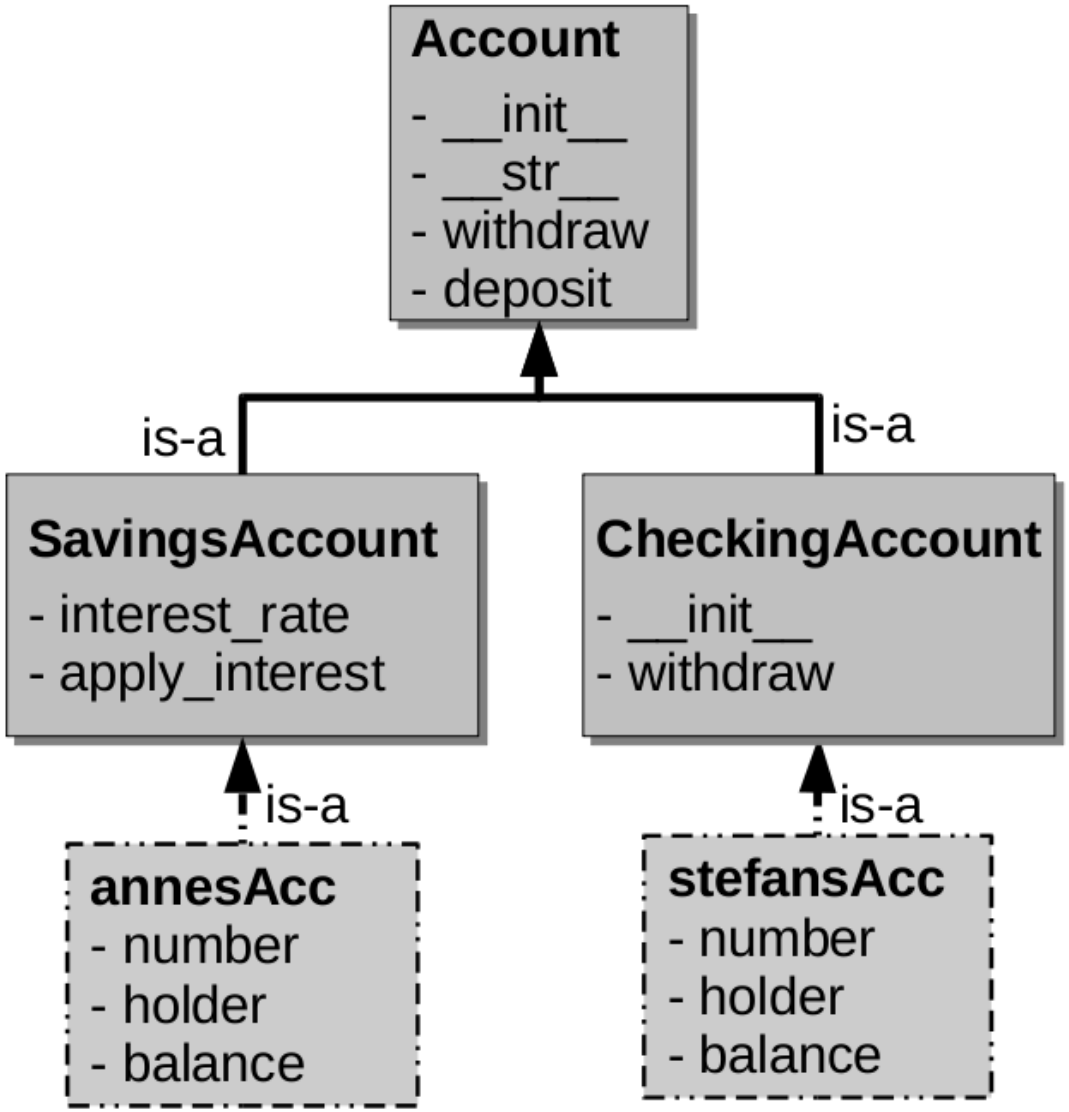
Application: `annesAcc.withdraw(400)`

- Python follows the class hierarchy and outputs the desired result

Overwriting Methods

```
1  class CheckingAccount(Account):
2    # CONSTRUCTOR
3    def __init__(self, num, person, credit_range):
4      print("Creating a checkings account")
5      self.number = num
6      self.holder = person
7      self.balance = 0
8      self.credit_range = credit_range
9    # METHODS
10   def withdraw(self, amount):
11     amount = min(amount, abs(self.balance +
12                  self.credit_range))
13     self.balance -= amount
14     return amount
```

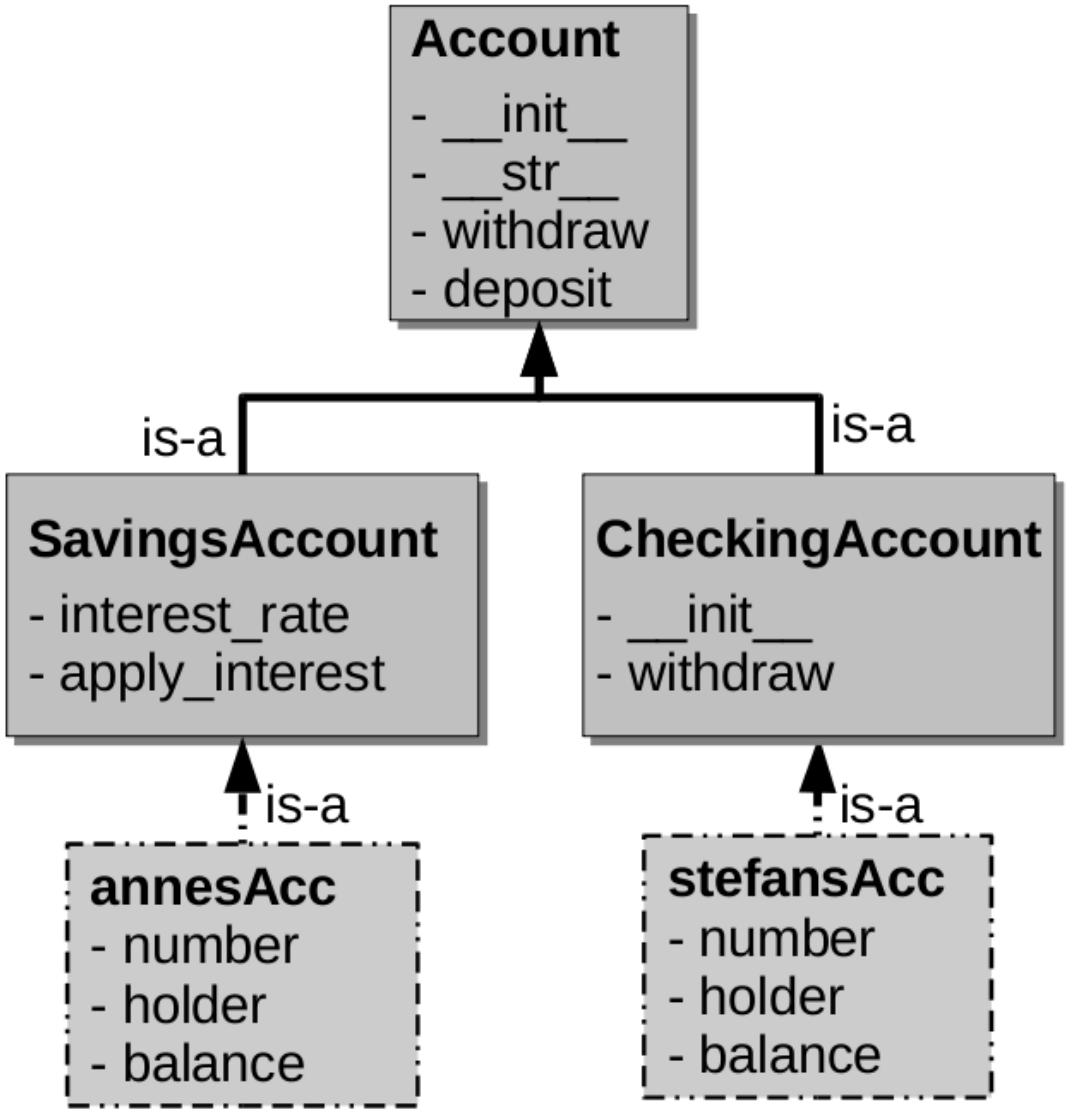
Class Hierarchy



Which methods are called, when the following methods are called on `annesAcc` and `stefansAcc`?

- `deposit`
- `withdraw`
- `apply_interest`

Class Hierarchy: Design



We could also have defined two separate `withdraw` methods (one in each child class)

Why is it useful to have it in the `Account` class?

Redundancy

- when the data for an object exist in multiple places (e.g., the account holder information exists separately for each type of account)
  $\Rightarrow$ possible inconsistencies
- the same code is written multiple times
  $\Rightarrow$ hard to maintain

```
1  class Account:
2    def __init__(self, num, person):
3      self.balance = 0
4      self.number = num
5      self.holder = person
6
7  class CheckingAccount(Account):
8    def __init__(self, num, person, credit_range):
9      self.number = num
10     self.holder = person
11     self.balance = 0
12     self.credit_range = credit_range
```

Minimizing Redundancies

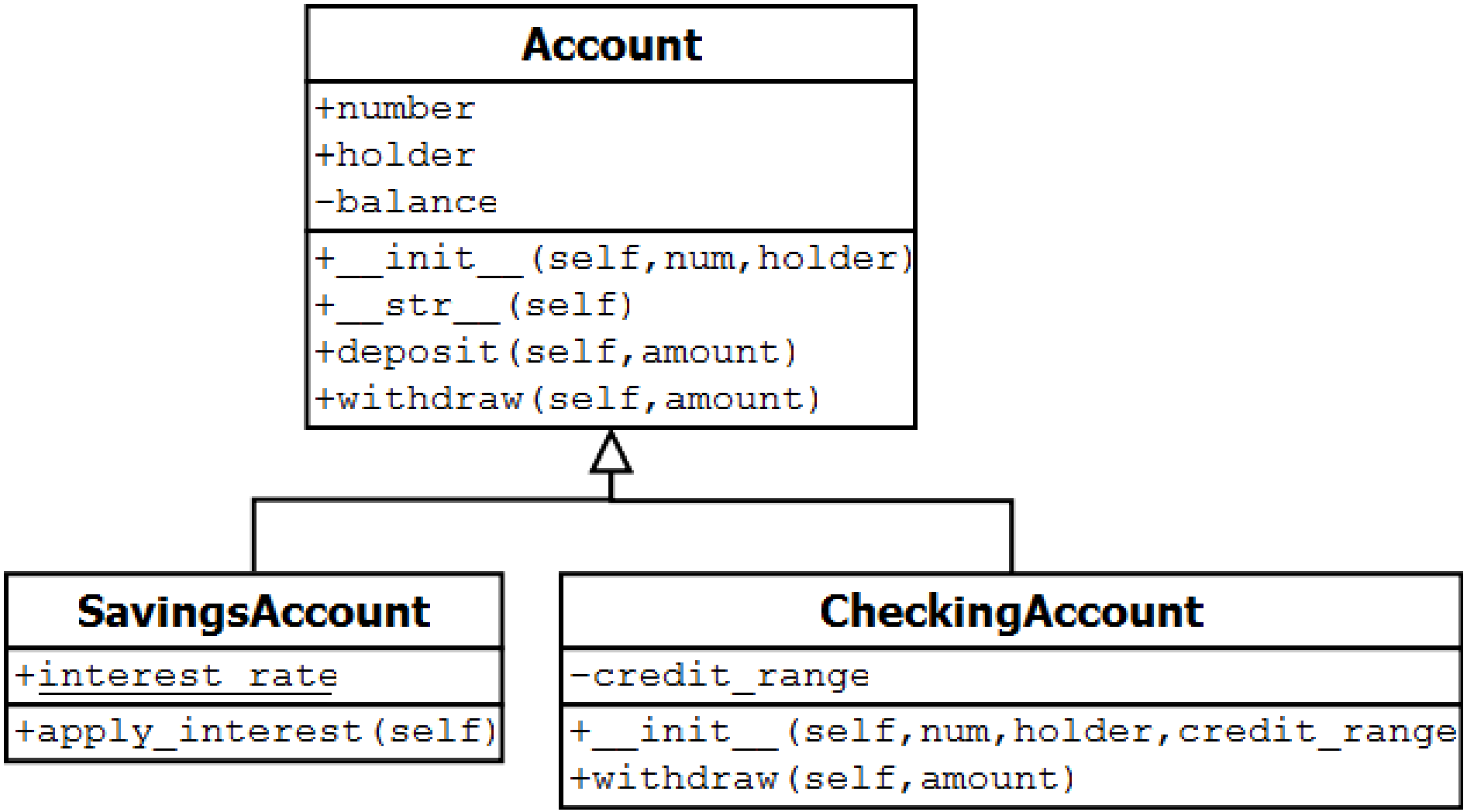
- here: call parent method from within a child and extend it with additional functionality
- method is called on the class ⇒ here, the reference to the instantiated object has to be explicitly given to `self`

```
1  class Account:
2    def __init__(self, num, person):
3      self.balance = 0
4      self.number = num
5      self.holder = person
6
7  class CheckingAccount(Account):
8    def __init__(self, num, person, credit_range):
9      Account.__init__(self, num, person)
10     self.credit_range = credit_range
```

Minimizing Redundancies: Another Example

```
1  class Account:
2    def withdraw(self, amount):
3      self.balance -= amount
4      return amount
5
6  class SavingsAccount(Account):
7    def withdraw(self, amount):
8      if amount > self.balance:
9        amount = self.balance
10       cash = Account.withdraw(self, amount)
11       return cash
12
13 class CheckingAccount(Account):
14   # METHODS
15   def withdraw(self, amount):
16     amount = min(amount,
17                  abs(self.balance + self.credit_range))
18     cash = Account.withdraw(self, amount)
19     return cash
```

UML Class Diagram: Inheritance



Multiple Inheritance

- in some object-oriented languages classes can only inherit from a single super class
- in Python a class can inherit from multiple classes
  $\Rightarrow$ Multiple Inheritance (Mehrfachvererbung)
- this is beyond the scope of this class
- recommendation: use max. one parent class for now

Everything in Python is an object

- we have been using objects all the time...
- lists and dictionaries are objects
- creation with special syntax but calling the class itself is also possible
- strings and integers/floats are also objects

```
1  # create a new list object
2  myList = []
3  # call a method of the list object
4  myList.append(4)
5  # create a new dictionary object
6  myDict = {}
7  # call a method of the dictionary object
8  myDict["someKey"] = "someValue"
```

- line 8 calls a magical method of the dictionary:
  `__setitem__(self, key, value)`

Everything in Python is an object

- we can also create child classes of the **built-in**-classes
- here: overwrite magical methods

```
1  class TalkingDict(dict):
2    # Constructor
3    def __init__(self):
4      print("Starting to create a new dictionary...")
5      dict.__init__(self)
6      print("Done!")
7    # Methods
8    def __setitem__(self, key, value):
9      print("Setting", key, "to", value)
10     dict.__setitem__(self, key, value)
11     print("Done!")
12
13 >>> print("We are going to create a talking dictionary!")
14 >>> myDict = TalkingDict()
15 >>> myDict["x"] = 42
```

Good understanding of OOP is important because:

- we rarely code from scratch in “real-life"
- Framework = collection of (Parent-) classes, which implement common programming tasks
- we need to understand how the classes of a framework work together
- we write child classes, which specify the behavior for our special application scenario

Terminology: Data Encapsulation

- Data Encapsulation is used for two reasons:
  1. data should be *hidden* and only available through instance methods
  2. the program-logic should be packaged in *interfaces* (class and method names) in such a way that each functionality is only defined once

OOP: Composition and Aggregation

Composition and Aggregation

- the attributes of an object can have any type
- they can themselves be (**complex**) objects
- **Composition** is a type of aggregation in which two entities are extremely reliant on one another.
  - complex objects are built based on other objects;
  - both entities are dependent on each other in composition.
  - the composed object **cannot exist** without the other entity when there is a composition between two entities.
  - In composition one class acts as a container of the other class (contents). If you destroy the container there is no existence of contents. That means if the container class creates an object or hold an object of contents.
  - Examples: `person` has a `heart`

Example - Composition

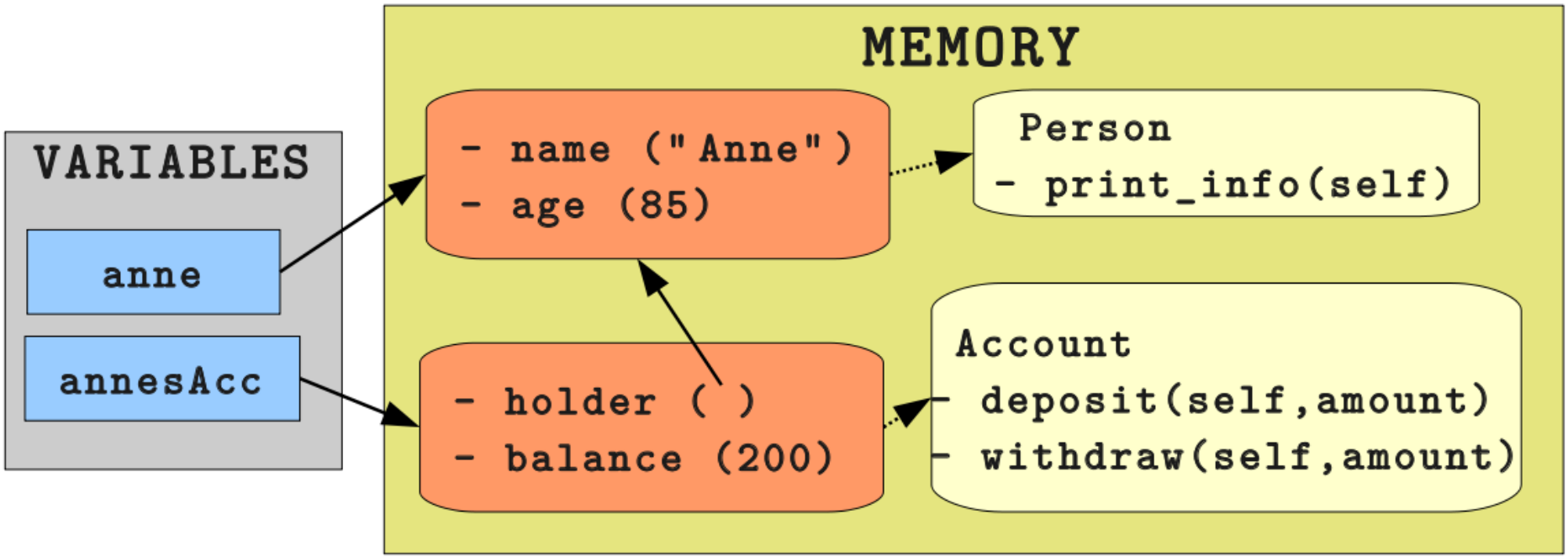
(Example by dineshmadhup)

```
1   class Heart:
2       def __init__(self, heartValves):
3           self.heartValves = heartValves
4       def display(self):
5           return self.heartValves
6   class Person:
7       def __init__(self, fname, lname, address, heartValves):
8           self.fname = fname
9           self.lname = lname
10          self.heartValves = heartValves
11          self.heartObject = Heart(self.heartValves) #composition
12      def display(self):
13          .....
14  p = Person("Adam", "syn", "876 Zyx Ln", 4)
15  p.display()
```

Composition and Aggregation

- **Aggregation** is a concept in which an object of one class can own or access another independent object of another class.
  - unidirectional has-a relationship;
  - one entity has a relationship to the other entity, but the other direction does not hold. For example, a department has students but it is vice versa is not possible (therefore unidirectional, i.e. one-way relationship)
  - the composed object **can exist** in isolation (in contrast to composition)
  - Examples: `department` has `students`

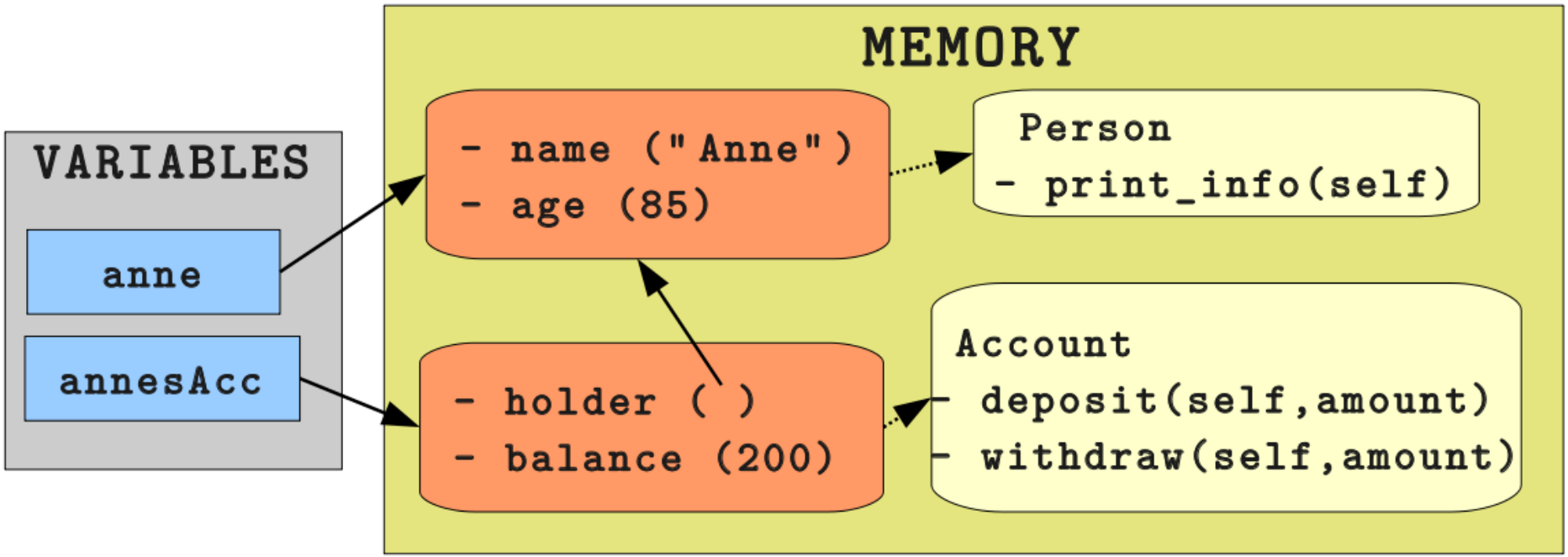
Composition and Aggregation



Example - version 1

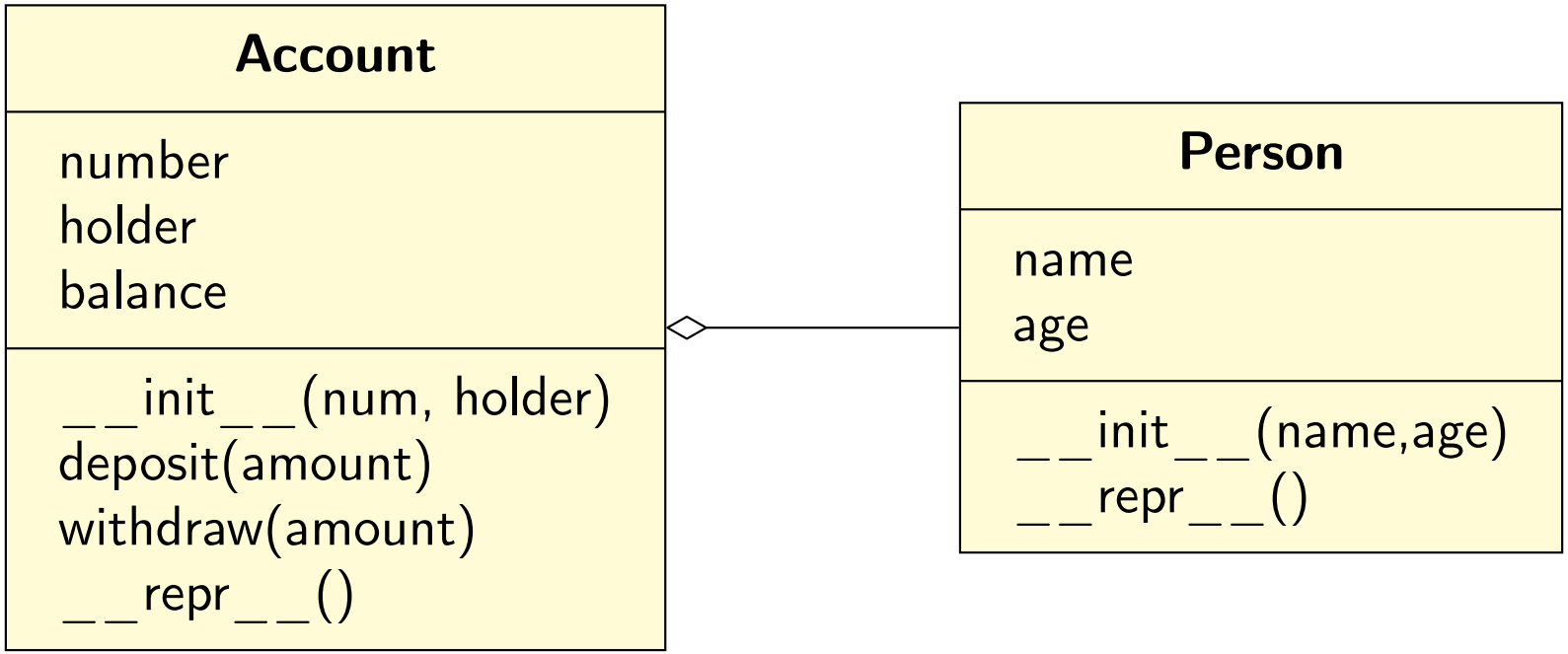
```
1  class Person:
2    def __init__(self, name, age):
3      self.name = name
4      self.age = age
5
6  class Account:
7    def __init__(self, num, person):
8      self.num = num
9      self.holder = person
10     self.balance = 0
11
12   def deposit(self, amount):
13     self.balance += amount
14
15 anne = Person("Anne", 85). # Aggregation
16 annesAcc = Account(1, anne)
17 annesAcc2 = Account(2, anne)
```

Shared References



- watch out where the attributes point: *annesAcc.holder.age += 1* also changes *annesAcc2.holder.age*
- this is ok here but could cause bugs in other cases

Aggregation: UML Diagram



N.B. we usually do not use 'self' in UML diagrams, as it is a general modeling language, and abstracts from Python-specifics

Composition: UML Diagram

Example - version 2 - What is the difference to version 1?

```
1  class Person:
2    def __init__(self, name, age):
3      self.name = name
4      self.age = age
5
6  class Account:
7    def __init__(self, num, name, age):
8      self.holder = Person(name, age) # composition
9      self.num = num
10     self.balance = 0
11
12   def deposit(self, amount):
13     self.balance += amount
14
15 annesAcc = Account(1, "Anne", 85)
16 annesAcc2 = Account(2, "Anne", 85)
```

Summary

- Python Modules
- Object Types
- Inheritance: parent and child classes
- Redundancy
- Multiple Inheritance
- Composition
- Aggregation

Questions

Questions?

Next Week: Representing documents and NLTK: Corpus linguistics