Introduction to Python Programming

13 – Object Orientation II

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

WS 2022/23

Recap

```
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)
```

Recap

```
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)
```

Recap

```
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)
```

Recap

```
a1.Account(1, 'Stephan')
a2.Account(2, 'Josef')
…
a1.get_print_info()
a2.get_print_info()
…
a1.set_deposit(100)
a1.set_deposit(50)
…
a1.get_print_info()
a2.get_print_info()
…
```

Objects and Classes

- Classes are objects, too.
- One (a single one) class object per class, created when Python evaluates (reads) the class for the first time.
- Instances are created by “calling the class”

```
>>> class Account:
>>>     def __init__(self, ...):
>>>         ...
>>>     ...
>>>
>>> Account
<class '__main__.Account'>
```

Objects and Classes

- Instance variables (“`self.xxx`”) belong to an individual object; the values can differ for different objects of the same class.
- Class variables belong to the class and are shared among all instances of the class

```
class Account:

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

Instance variables …

Objects and Classes

- Instance variables ("self.xxx") belong to an individual object; the values can differ for different objects of the same class.
- Class variables belong to the class and are shared among all instances of the class

```
class Account:
    num_of_accounts = 0 # class attribute

    def __init__(self, number, holder):
        self.number = number
        self.holder = holder
        self.balance = 0
        Account.num_of_accounts += 1
    ...
```

A class variable

Instance variables …

Objects and Classes

- Instance variables ("self.xxx") belong to an individual object; the values can differ for different objects of the same class.
- Class variables belong to the class and are shared among all instances of the class
- Instance methods are also class variables

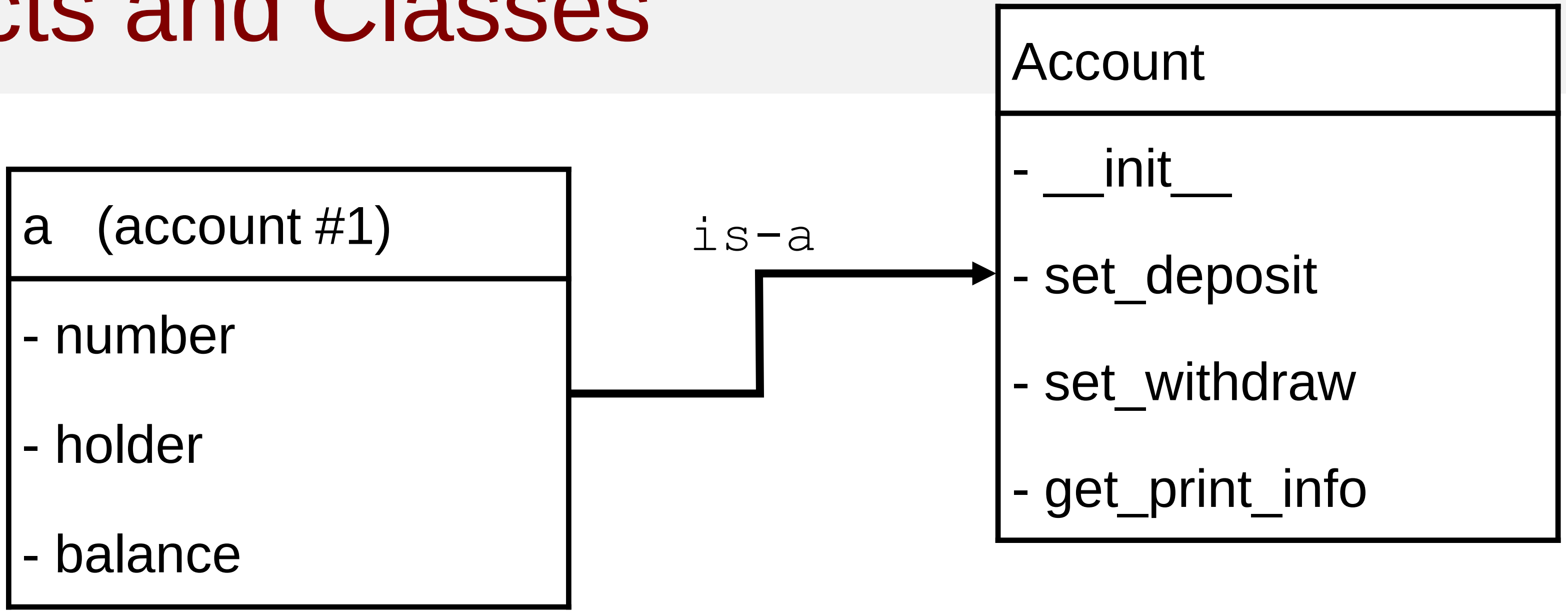
```
class Account:
   def __init__(self, number, holder):
      self.number = number
      self.holder = holder
      self.balance = 0

   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)
```

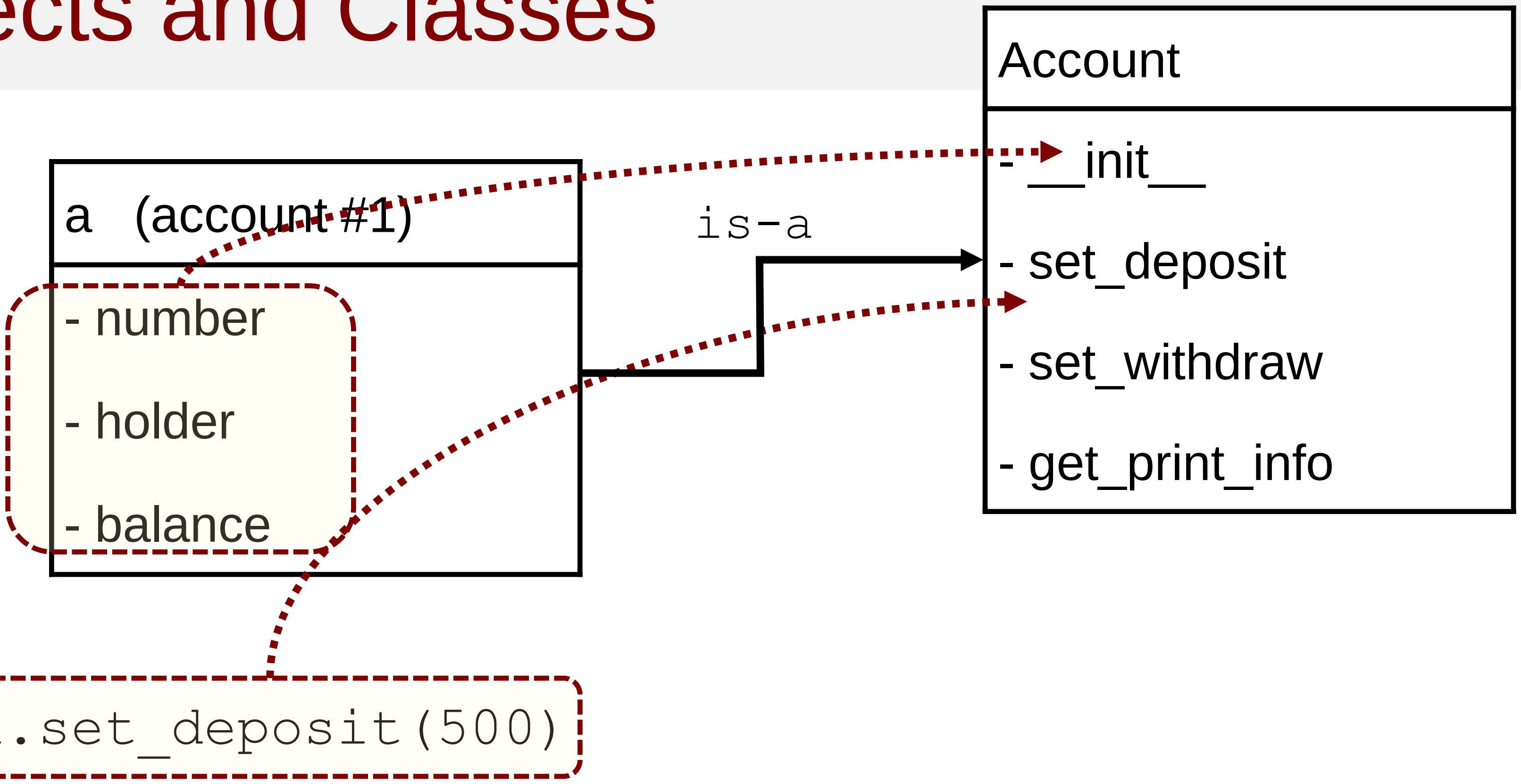
Objects and Classes



```
>>> a.set_deposit(500)
```

- Python starts to search for the deposit method at the object
- The object contains only instance variables, no methods
- Continue search for method in class from which the object was created

Objects and Classes



```
>>> a.set_deposit(500)
```

- Python starts to search for the deposit method at the object
- The object contains only instance variables, no methods
- Continue search for method in class from which the object was created

Achtung: Shadowing …!

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

The code does not work: the instance variable `balance` overrides / shadows the method `balance(…)`

Achtung: Shadowing …!

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

The code does not work: the instance variable `balance` overrides / shadows the method `balance(…)`

Achtung: Shadowing …!

```
class Account:
   def __init__(self, ...):
      self._balance = 0
      ...
   def balance(self):
      return self._balance
```

Convention: use “_” as the first character of the name of instance variables to avoid name clashes.

Achtung: Shadowing …!

```
class Account:
   def __init__(self, ...):
      self.__balance = 0
      ...
   def balance(self):
      return self.__balance
```

Convention: or use “__” as the first character of the name of instance variables to avoid name clashes and for complete data encapsulation (no access to data from outside except through instance methods that come with the class).

Objects and Classes

- We can also assign attributes to a class object: class attributes/variables
- Achtung: this is **rarely** needed, but useful in example below:

```
class Account:
    num_of_accounts = 0 # class attribute
    def __init__(self, number, holder):
        self.__number = number
        self.__holder = holder
        Account.num_of_accounts += 1
    ...
```

Objects and Classes

- We can read class attributes via instance objects that were created from this class
- Better: always access them via class name

```
>>> a1 = Account(1, "Timo")
>>> a2 = Account(2, "Stefan")
>>> a1.num_of_accounts
2
>>> Account.num_of_accounts
2
```

Better style!

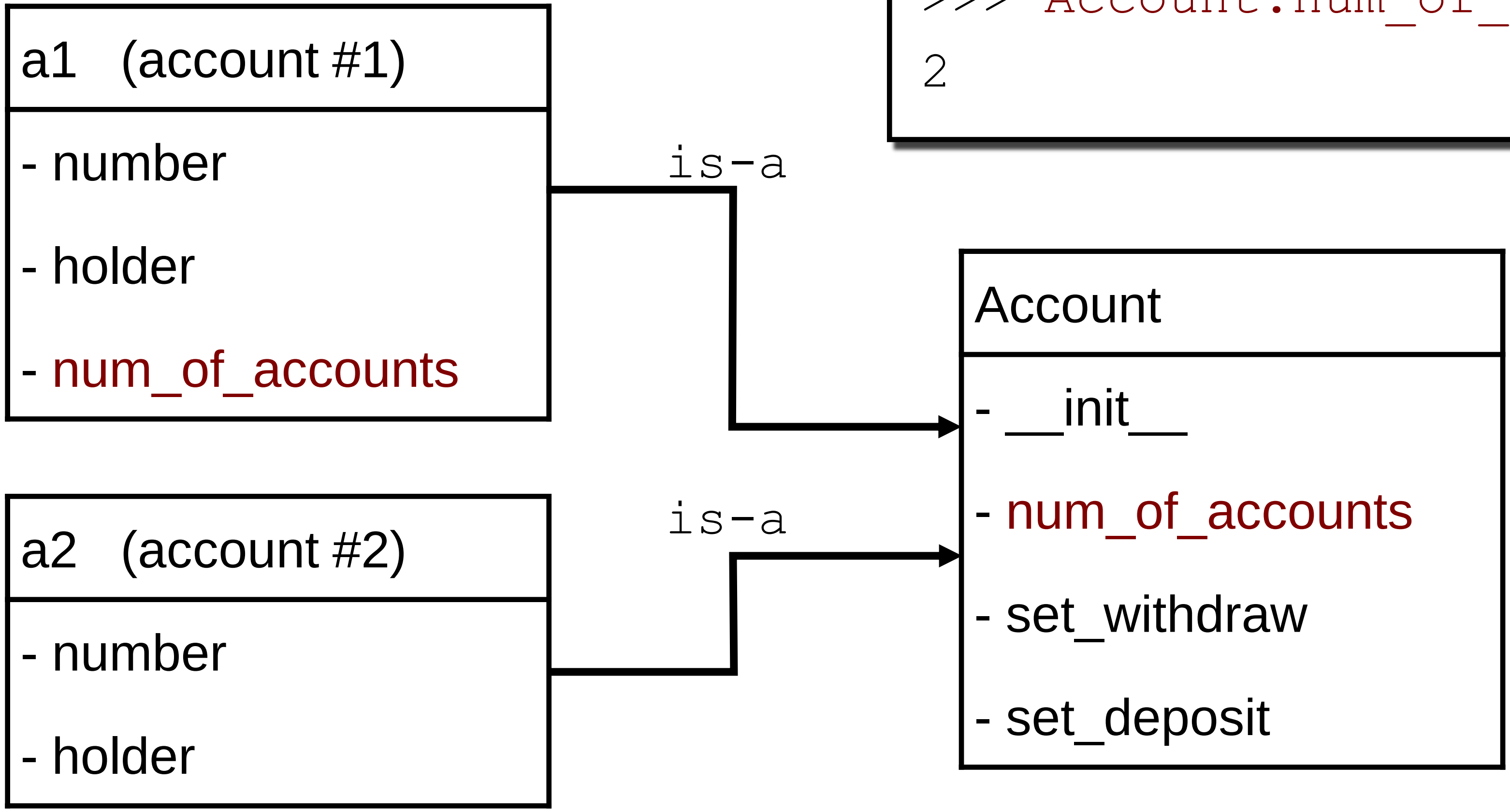
Objects and Classes

- We **CAN NOT** write class attributes via instance objects that were created from this class.
- Line 3: we create an instance variable that locally overrides the class variable!

```
>>> a1 = Account(1, "Timo")
>>> a2 = Account(2, "Stefan")
>>> a1.num_of_accounts += 1
>>> a1.num_of_accounts
3
>>> Account.num_of_accounts
2
```

Objects and Classes

```
>>> a1 = Account(1, "Timo")
>>> a2 = Account(2, "Stefan")
>>> a1.num_of_accounts += 1
>>> a1.num_of_accounts
3
>>> Account.num_of_accounts
2
```



Coding Style

- Never give the same (“shadowing”) name to class attributes and instance attributes.
- Always access class attributes via the class, never via instance objects.

Recap: Methods

- All methods we have seen so far are instance methods.
  - the method itself “belongs” to the class, …, its actually a class variable …
  - but the method operates on the instance
- Designed to be called on an object of the class.
- The object on which it is called is automatically assigned to `self`
  - All other parameters need to be given in method call.

Static Methods

- Class objects can also have **static methods**.
  - Designed to be called on the class object.
  - Methods that do not involve a particular instance object, but that somehow involve the class (e.g., class attributes)
  - Methods creating objects of this class.

Static Methods

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

  @staticmethod
  def accounts_info():
    print(Account.num_of_accounts,
      "accounts have been created.")

if __name__ == "__main__":
   a1 = Account(1, "John")
   a2 = Account(2, "Jane")
   Account.accounts_info()
```

no “self”!
preceeding line must be
@staticmethod
called a “decorator“

Works on class variable

Static Methods

- Python also provides an additional more powerful mechanism: class methods.
- They are beyond the scope of our lecture for now.

Types of Objects

- Values in Python have types:
  - `1.5` has type `float`
  - `'Stefan'` has type `str`
  - ...
- The type of the `stefansAcc` instance object is the class from which it was created
  - `stefansAcc = Account(1, "Stefan")`
  - `type(stefansAcc) == Account` ⇒ `True`
  - `isinstance(stefansAcc, Account)` ⇒ `True`

Modules

- For big programmes, Python code can get very long
- Modules are Python files that group related code
  - Purpose: structure your code into modules to make it easier to maintain.
- We can then import these modules in other modules.
  - `import my_module`
  - `from my_module import my_function`
  - `from my_module import *`
- The module’s name is the name of the file name minus “`.py`”

Modules

- When importing a module, the code in the module is evaluated (Python executes the code in the file).
- Since modules are just Python files, we can execute them in the usual way:
  - F5 in IDLE
  - `python3 my_module.py` (from the command line)
- Also: `__name__ == "__main__"`
  - evaluates to True when the module is run, but False when it is imported
  - ⇒ We can use this in each module for automatic tests, but at runtime, only the main module defines what happens!

Modules

- Importable code should have no function calls outside of the "`if __name__ == '__main__'`" block.

```
# my_module.py
def f(x):
  print(x)

f('Hello')
```

```
# main.py
import my_module

my_module.f('World')
```

- What happens when we execute the main module?

Compositionality

- The attributes of objects can be of any type
- They can be objects themselves
- This ability to create objects out of objects is called compositionality.
  - Access with dot notation: a1.holder.name

Compositionality

```
class Person:
   def __init__(self, name, surname):
      self.name = name
      self.surname = surname

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

```
>>> p1 = Person("Jane", "Doe")
>>> a1 = Account(1, p1)
>>> a1.holder.name
"Jane"
>>> a1.holder.surname
"Doe"
>>>
```

Shared References

- Need to be aware of where attributes point to
- In example below, modifying `a1.holder` also modifies `a2.holder`
- Desired here, but may be a bug in another setting.

```
>>> p1 = Person("Jane", "Doe")
>>> a1 = Account(1, p1)
>>> a2 = Account(2, p1)
>>> a2.holder.surname
"Doe"
>>> a1.holder.surname = "Smith"
>>> a2.holder.surname
"Smith"
```

Shared References

- Need to be aware of where attributes point to
- In example below, modifying `a1.holder` also modifies `a2.holder`
- Desired here, but may be a bug in another setting.

```
>>> p1 = Person("Jane", "Doe")
>>> a1 = Account(1, p1)
>>> a2 = Account(2, p1)
>>> a2.holder.surname
"Doe"
>>> a1.holder.surname = "Smith"
>>> a2.holder.surname
"Smith"
```

Remember:
bad style!

Composition vs Aggregation

- Composition = contained objects exist only within a complex object
  - Example: Object balance (of type int) does not exist without an Account object. If the Account object is deleted, we don’t have any reference to the balance any more.
- Aggregation = no implied ownership; contained objects can exist independently; can also be used in more than one object.
  - Example: Account object has a reference (attribute) to a Customer/Person object. If Account object is deleted, the Customer/Person object can still exist (e.g., customer has two accounts).