Compact Course Python

Michaela Regneri & Andreas Eisele

Lecture 3

Object-oriented programming

- Procedural / imperative programming: data is kept separate from operations
- Object oriented programming: data and operations are combined to objects (or classes)
  - data is stored in fields (≈ variables)
  - methods (≈ functions) define operations on the fields
  - fields and methods are also called attributes
- Objects are instances of classes: classes define objects with similar properties

A first example: rational numbers

- Data
  - Numerator and denominator
- Operations
  - add
  - multiply
  - convert to a string
  - [...]

Rational numbers: Imperative

```
def rat_make(num, den):
    return (num, den)

def rat_tostring(rat):
    return str(rat[0]) + "/" + str(rat[1])

def rat_mul(rat1, rat2):
    num = rat1[0] * rat2[0]
    den = rat2[1] * rat2[1]
    return rat_make(num, den)

...
```

Rational numbers: Object

```
class Rat:
  def __init__(self, num, den):
      self.num = num
      self.den = den

  def toString(self):
      return str(self.num) + "/" + str(self.den)

  def mul(self, other):
      num =  self.num * other.num
      den =  self.den * othder.den
      return Rat(num, den)
```


Rational numbers: Object

- *Instatiate* two new `Rat` objects and bind them to `r1` and `r2`

```
r1 = Rat(1,2)
r2 = Rat(2,3)
```

- multiply to r1 and r2, bind the result to r3

```
r3 = r1.mul(r2)
```

- Output as String

```
print(r3.toString())
```

Why OOP?

- Object-oriented programming (OOP) encourages the programmer to divide programs into classes.
- for many projects, the class level is an appropriate level of granularity, and classes correspond to intuitive concepts
- in a good class hierarchy, the complexity of individual classes is manageable, which makes the code more readable and handable

Why OOP?

- You can hide implementation details of classes (and just show functions with their parameters and return values)
- Other programmers (users of the classes) may continue to use the classes directly, or expand, without changing it
- The implementation can be changed at any point in time, it won‘t affect the remainder of the program

Why OOP?

- Classes can be derived from other classes.
- Derived classes inherit all the attributes of the base class, can add new attributes and may override the inherited methods
- Objects of the derived class can be used anywhere where objects of the base class are accepted

Overview

- Namespaces and scope
- Classes, methods, objects
- Special methods for operator overloading

Scopes and Namespaces

- A namespace is a mapping of identifiers (names) to objects
- the same names in different namespaces can refer to different objects
- One can think of namespaces as dictionaries, whereas the keys are restricted to valid variable (or function) names
- Direct access to names (or objects) in a namespace: `namespace.attr`

Functions and namespaces

- with each function call , a local namespace is created in which there are local variables (only)
- when the function is exited, the namespace is deleted (resp. „forgotten“)
- in the case of recursion, each recursive call to the function has its own namespace

Scope

- *scope* is the part of a program in which you can access certain names directly ("directly" means without other keywords)
- there are 3 (nested) namespaces:
  - built-in names (eg. print)
  - global names
  - local names
- within functions, we refer to local names in separate namespaces
- outside of functions: global = local

Classes

- Classes in Python *need* nothing other than a name. They are defined with the keyword `class`

```
class <name>:
  [Statement1]
  ...
  [Statementn]
```

- classes can define methods; they are functions within the class, that have *self* as their first argument (*self* will be the object calling the function)

```
class <name>:
  def fun1(self[,...]):
     ...
```

- The class has its own namespace

Classes

- The class definition in the Python program must happen before you can use the class
- In the global name space, there will be a `class` object that has the name of the class
- classes (more precisely, `class` objects) support exactly two operations:
  - referencing attributes
  - instantiation (creation of instance objects)

Classes

```
class K:
  def fun(self):
    self.x = 2
    ...
```

```
k = K()
k.fun()
print(k.x)
```

- Instantiation: with `k = K()` instance object of K is created (and bound to `k`).
- assignments from „outside“ are allowed (as k.a = 8)

Instance objects

- Instance objects can use attributes of the class
- We distinguish:
  - data attributes ("instance variables")
  - methods
- methods are called directly (without self)
- Namespace resolution: if the attribute is not found in the instance, python looks for it in the class definition

Method calls

- The body functions defined in the class are the methods of the instance
- The first argument (`self`) Of the function is bound to the instance:
  - In the example, `k.f()` is equivalent to `MyClass.f(k)`

```
class MyClass:
   i = 123
   def f(self):
     print(MyClass.i)
```

```
>>> k = K()
>>> k.f()
123
>>> MyClass.f(k)
123
```

A simple example

```
class MyClass:
    i = 123
    def f(self):
        print(MyClass.i)

>>> k = MyClass()
>>> k.f()
123
>>> k.f
<bound method MyClass.f of ...>
```


A simple example

```
class MyClass:
    i = 123
    def f(self):
        print(MyClass.i)

>>> k = MyClass ()
>>> print(k.i)
123
>>> k.i = 321
>>> MyClass.i = 17
>>> k.f()
17
```

`__init__` 2 underscores!

- Instantiation first generates an "empty" object.
- The method `__init__` is automatically called with the arguments used in the instantiation.
- Typical code:

```
class SomeClass:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  ...
inst = SomeClass(1, 2)
```

corresponds to a so called *constructor*

Inheritance

- In object oriented languages, classes can inherit from other classes
- The derived class *inherits* attributes from the base class
- All class automatically have a base class (`object`) In Python, the inherited things
  - `object` inherited a method that one *Hash code* generated - that is, one may use self-generated classes in standard quantities and Dictionaries
  - what else of `object` is inherited, we see in later lectures

Inheritance: an example

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

class FrenchGuy(Person):
    def sayHello(self):
        print("Bonjour " + self.name)

class GermanGuy(Person):
    def sayHello(self):
        print("Hallo " + self.name)
```

```
>>> g = GermanGuy('Stefan')
>>> g.sayHello()
Hallo Stefan
>>> f = FrenchGuy('Romain')
>>> f.sayHello()
Bonjour Romain
```


Inheritance: override methods

- Sometimes you want not only add new methods to the base class, but also modify existing ones (most often: `__init__`).
- You can override methods simply by redefining them
- If you want to access the corresponding method of the base class, you can use the built-in method `super` :

  `super().method(...)` does the same as `BaseClass.method(self,...)`

Override methods: Example

```
class Person:
    def __init__(self, name):
        self.name = name
    ...
class Employee(Person):
    def __init__(self, name, salary):
        super().__ init__(name)
        self.salary = salary
    ...
```


Abstract Classes

- *abstract* classes are a popular concept in object-oriented programming are
- abstract classes contain unimplemented methods (without body) and must be implemented in derived classes to make them work
- Python has no abstract classes - but you can simulate them: the base class defines a "placeholder" method, which does nothing, or throws an exception.
- Python keyword for "doing nothing" is **`pass`**

An "abstract" class

```
class Xmlparser:

  def parse(self):
        ...
        self.handleElement(someElement)
        ...

  def handleElement(self, someElement):
    pass
    # alternative: raise NotImplementedError


class MyXmlParser(Xmlparser):
    def handleElement(self, someElement):
        ...
```


Private variables (name mangling)

- In Python there is no "real" private variables and methods that are accessible only within the class
- To avoid naming conflicts, names can be "mantled": identifiers of the form `__foo` are automaticall replaced by `__klassenname_foo` (for calls outside the class)

Name conflicts & convention

- data attributes override method attributes with the same name.
- Common convention for the avoidance of conflict: data attributes start with an underscore: `_foo`.

Hooks

- In the last few lectures were presented to operators: +, -, ...
- Strictly speaking, there are no operators in Python, just operations:
  - The "+" operator, for example, calls internally the `__add__` method of the first operand
  - you can define those special methods ("hooks") yourself in order to change or extend the functionality.

Rational numbers with operators

```
class Rat:
    def __init__(self, num, den):
        self.num = num
        self.den = den
    def __mul__(self, other):
        num = self.num * other.num
        den = self.den * other.den
        return Rat(num, den)
    def __repr__(self):
        return "Rat(" + str (self.num)+","+ str
        (self.den) + ")"
    def __str__(self):
        return str(self.num) + "/" + str (self.den)
```

```
>>> r1 = Rat(1,2)
>>> r2 = Rat(3,4)
>>> r1 * r2
Rat(3, 8)
>>> print(r1 * r2)
3 / 8
```


Some special methods

- relational operators:
  - `__eq__` `==`
  - `__ge__` `>=`
  - `__gt__` `>`
  - `__le__` `<=`
  - `__lt__` `<`
  - `__ne__` `!=`

with 2 underscores!

- `__bool__` : :is the object as True or False?

Some special methods

- Numerical operations:
  - `__add__, __iadd__` +, + =
  - `__truediv__, __itruediv__` /, /=
  - `__mul__, __imul__` *, *=
  - `__sub__, __isub__` -, -=
  - `__mod__, __imod__` %,%=

For example: `dict` with default value

```
>>> d = Defaultdict(0)
>>> d[17]
0
>>> d[17] + = 1
>>> d[17]
1
```

```
class Defaultdict(dict):
    def __init__(self, default):
        self.default = default

    def __getitem__ (self, key):
        if key in self:
            return super().__getitem__(key)
        else:
            return self.default
```

Classes, modules, functions

- Classes should be used if you want to manage multiple states simultaneously.
- If any one condition is sufficient: Module (= a Python file)
- If you need no state: Features

Summary

- Classes & Objects
- Inheritance
- Methods & Operator Overloading
- Multiple inheritance
- Example