Programmierkurs Python I

Michaela Regneri

2009-11-19

(Folien basieren auf dem gemeinsamen Kurs mit Stefan Thater)

Objektorientierte Programmierung

- Prozedurale / imperative Programmierung: Daten und Operationen sind separat
- Objektorientiere Programmierung: Daten und Operationen werden in Objekten (bzw. Klassen) zusammengefasst
  - Daten werden in Feldern (≈ Variablen) gespeichert
  - Methoden (≈ Funktionen) definieren Operationen auf den Feldern
  - Felder und Methoden werden auch Attribute genannt
- Objekte sind Instanzen von Klassen: Klassen definieren gleichartige Objekte mit ihren Operationen

Ein erstes Beispiel: Rationale Zahlen

- Daten
  - Zähler und Nenner
- Operationen
  - in einen String konvertieren
  - Addieren
  - Multiplizieren
  - [...]

Rationale Zahlen: Imperativ

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

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

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

...
```

Rationale Zahlen: Objektorientiert

```
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 * other.den
      return Rat(num, den)
```


Rationale Zahlen: Objektorientiert

- Rationale Zahlen instantiieren (erzeugen) und an r1 bzw. r2 binden

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

- r1 mit r2 multiplizieren; Ergebnis an r3 binden

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

- Als String ausgeben

```
print(r3.tostring())
```

Warum OOP?

- Objektorientierte Programmierung (OOP) ermutigt den Programmierer dazu, Programme in Klassen aufzuteilen.
- Für viele Projekte sind Klassen gute Gliederungsebene, die zu Dingen in der wirklichen Welt passen.
- in einer guten Klassenhierarchie ist die Komplexität einzelner Klassen überschaubar, das macht den Code übersichtlicher

Warum OOP?

- Man kann Implementierungsdetails von Klassen nach außen verbergen
- Andere Programmierer (Benutzer der Klassen) können die Klassen direkt weiterverwenden, oder erweitern, ohne sie zu verändern
- Die Implementierung kann jederzeit verändert werden, ohne das Gesamtprogramm zu stören

Warum OOP?

- Klassen können von anderen Klassen abgeleitet werden.
- Abgeleitete Klassen erben alle Attribute der Basisklase, können neue dazutun und die geerbten Methoden überschreiben
- Objekte der abgeleiteten Klasse können überall eingesetzt werden, wo Objekte der Basisklasse akzeptiert werden

Übersicht

- Namensräume und Geltungsbereiche
- Klassen, Methoden, Objekte
- Spezielle Methoden zum Überladen von Operatoren

Geltungsbereiche & Namensräume

- Ein Namensraum (Namespace) ist eine Abbildung von Bezeichnern (Namen) auf Objekte
- Namen in verschiedenen Namensräumen können auf verschiedene Objekte referieren
- Man kann sich Namensräume als Dictionaries vorstellen; Schlüssel sind dabei eingeschränkt auf zulässige Namen
- Qualifizierter Zugriff auf Namen (bzw. Objekte) in einem Namensraum: `namespace.attr`

Funktionen und Namensräume

- Bei einem Funktionsaufruf wird ein lokaler Namensraum erzeugt, lokale Variablen existieren (nur) in diesem Namensraum
- Beim Verlassen der Funktion wird der Namensraum wieder gelöscht (bzw. „vergessen“)
- Bei Rekursion hat jeder rekursive Aufruf der Funktion seinen eigenen Namensraum

Geltungsbereich (Scope)

- Ein Geltungsbereich ist ein Bereich im Programm, innerhalb dessen man direkt auf Namen zugreifen kann („direkt“ = ohne andere Schküsselwörter)
- Drei (verschachtelte) Namensräume:
  - Eingebaute Namen (z.B. print)
  - Globale Namen
  - Lokale Namen
- Innerhalb von Funktionen referenzieren wir Namen in separaten lokalen Namensräumen
- Ausserhalb von Funktionen: Global = Lokal

Klassen

- Klassen in Python *brauchen* nichts außer einem Namen; definiert werden sie mit dem Schlüsselwort `class`

```
class <name>:
   [statement1]
   ...
   [statementn]
```

- Klassen können Methoden definieren; das sind Funktionen innerhalb der Klasse, die als erstes Argument *self* haben (*self* ist später das Objekt, das die Methode gerade aufruft)

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

- Die Klasse hat ihren eigenen Namensraum

Klassen

- Die Klassendefinition muss im Python-Programm ausgeführt werden, bevor man die Klasse verwenden kann
- Im globalen Namensraum befindet sich dann ein Klassen-Objekt, das den Namen der Klasse hat
- Klassen (genauer: Klassen-Objekte) unterstützen genau zwei Operationen:
  - Referenzieren von Attributen
  - Instantiierung (erzeugen von Instanz-Objekten)

Klassen

```
class K:
  a = 83
  def fun(self):
    ...
```

- Instantiierung: mit `k = K()` erzeugt man ein Instanz-Objekt von K (und bindet es an k).

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

- Referenzieren von Attributen: Wenn K ein Klassen-Objekt ist, kann man mit `K.a` auf das Attribute a zugreifen
- Zuweisungen sind erlaubt (so wie K.a = 8)

Ein einfaches Beispiel

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

>>> MyClass.i
123
>>> MyClass.f
<unbound method MyClass.f>
```


Instanz-Objekte

- Instanz-Objekte können Attribute der Klasse benutzen
- Wir unterscheiden:
  - Daten-Attribute („Instanz-Variablen“)
  - Methoden
- Methoden werden nach dem referenzieren direkt aufgerufen
- Namensraum-Auflösung: wenn das Attribut nicht in der Instanz gefunden wir, wird in der Klasse gesucht

Methodenaufrufe

- Die im Klassenkörper definierten Funktionen sind die Methoden der Instanz
- Das erste Argument (`self`) der Funktion ist an die Instanz gebunden:
  - Im Beispiel ist `k.f()` äquivalent zu `MyClass.f(k)`.

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

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


Ein einfaches Beispiel

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

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

Ein einfaches Beispiel

```
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!

- Instantiierung erzeugt zunächst ein „leeres“ Objekt.
- Die Methode `__init__` wird automatisch mit den bei der Instantiierung verwendeten Argumenten aufgerufen.
- Typischer Code:

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

entspricht einem sog **Konstruktor**

Vererbung

- In objektorientierten Sprachen kann man (normalerweise) Klassen von anderen Klassen ableiten
- Die abgeleitete Klasse *erbt* Attribute von der Basisklasse
- Alle Klasse haben automatisch eine Basisklasse (`object`) in Python, die vererbt auch Dinge
  - `object` vererbt eine Methode, die einen *Hash-Code* erzeugt -- das heißt, man darf selbst erzeugte Klassen standardmäßig in Mengen und Dictionaries benutzen
  - was sonst von `object` vererbt wird, sehen wir in späteren Vorlesungen

Vererbung: Ein Beispiel

```
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('Etienne')
>>> f.sayHello()
Bonjour Etienne
```

Vererbung: Methoden überschreiben

- Manchmal möchte man eine Basisklasse nicht nur neue Methoden hinzufügen, sondern existierende Methoden modifizieren (häufig: `__init__`).
- Man kann Methoden einfach überschreiben, indem man sie neu definiert
- Wenn man auf die gleichnamige Methode der Basisklasse zugreifen möchte, kann man die eingebaute Methode **`super`** benutzen:
  `super().methode(...)` tut das gleiche wie
  `BasisKlasse.methode(self,...)`

Methoden überschreiben: Beispiel

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

Abstrakte Klassen

- Ein gängiges Konzept aus der objektorientierten Programmierung sind abstrakte Klassen
- Abstrakte Klassen enthalten nicht implementierte Methoden (ohne Körper) und müssen abgeleitet werden, um sinnvoll verwendet zu werden.
- Python kennt keine abstrakten Klassen – man kann sie jedoch einfach nachbilden: die Basisklasse definiert eine „Platzhalter“-Methode, die gar nichts tut, oder eine Ausnahme wirft.
- Pythons Schlüsselwort für „nichts tun“ ist pass

Eine „abstrakte“ Klasse

```
class XmlParser:
   def parse(self):
      ...
      self.handleElement(someElement)
      ...
   def handleElement(self, someElement):
      pass # alternativ: raise NotImplementedError


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

Mehrfachvererbung

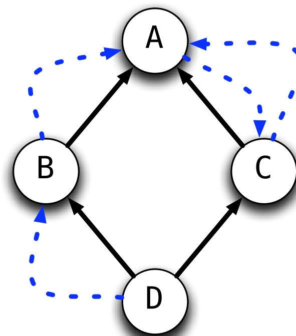
- Python unterstützt Mehrfachvererbung: Eine Klasse kann von mehreren Basisklassen abgeleitet werden:

```
class DerivedClass(Base1, ..., Basen):
    ...
```

- Resolution bei Attribut-Zugriff: depth-first, left-to-right
  - Zuerst wird das Attribut in $Base_1$ gesucht
  - dann rekursiv in den Basisklassen von $Base_1$
  - dann in $Base_2$,
  - [...]
  - (bis das Attribut gefunden wird).

Mehrfachvererbung

```
class A:
    def f(self): return 1

class B(A):
    def f(self): return 2

class C(A):
    pass

class D(B, C):
    pass

d = D()

d.f() ⇒  ?
```



Private Variablen (Name Mangling)

- In Python gibt es keine „echten“ privaten Variablen bzw. Methoden, die nur innerhalb der Klasse zugreifbar sind.
- Um Namenskonflikte zu vermeiden, können Namen „verstümmelt“ werden: Bezeichner der Form `__foo` werden automatisch durch `__klassenname_foo` ersetzt.

Namenskonflikte & Konvention

- Daten-Attribute überschreiben Methoden-Attribute mit gleichem Namen.
- Gängige Konvention zur Vermeidung von Konflikten: Daten-Attribute beginnen mit einem Unterstrich: `_foo`.

Bürger erster Klasse

- In Python sind Klassen „Bürger erster Klasse," d.h., sie unterliegen nur den Einschränkungen, die für wirklich alles in Python gelten
- Man kann beispielsweise Klassen auch innerhalb von Funktionen definieren
- oder Klassen selbst (nicht nur ihre Instanzen!) als Argument in Funktionsaufrufen verwenden

Hooks

- In den letzten Vorlesungen wurden einige Operatoren vorgestellt: +, -, ...
- Streng genommen gibt es in Python aber gar keine Operatoren, sondern nur Operationen:
  - Der „+"-Operator ruft beispielsweise intern die __add__ Methode des ersten Operanden auf.
  - Diese speziellen Methoden („hooks") kann man selbst definieren (bzw. überschreiben), um damit die Funktionalität zu ändern oder zu erweitern.

Rationale Zahlen mit Operatoren

```
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.den)+","+ str(self.num) + ")"
   def __str__(self):
      return str(self.den) + "/" + str(self.num)
```

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


Einige spezielle Methoden

- Vergleichsoperatoren:
  - `__eq__` `==`
  - `__ge__` `>=`
  - `__gt__` `>`
  - `__le__` `<=`
  - `__lt__` `<`
  - `__ne__` `!=`
- `__bool__` : gilt das Objekt als Wahr oder Falsch?

jeweils 2 underscores!

Einige spezielle Methoden

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

Ein Beispiel: dict mit Defaultwert

```
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
```

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

Klassen, Module, Funktionen

- Klassen sollten verwendet werden, wenn man mehrere Zustände gleichzeitig verwalten möchte.
- Wenn ein einziger Zustand ausreicht: Module (= eine Python-Datei)
- Wenn man gar keinen Zustand braucht: Funktionen

Zusammenfassung

- Klassen & Objekte
- Vererbung
- Methoden & Operatoren überladen
- Mehrfachvererbung
- Beispiel