Python-Crashkurs

Andreas Abel

Lehrstuhl für Theoretische Informatik
Ludwig-Maximilians-Universität München

Wintersemester 2008/09
6.-10. Oktober 2008

Typen

- `type(v)` liefert den Typ von `v`.
- Typmitgliedschaft testet man mit `isinstance(val,typ)`.

```
>>> isinstance(5, float)
False
>>> isinstance(5., float)
True
```

- Dabei wird Mitgliedschaft in einer Vaterklasse akzeptiert.

```
>>> isinstance(NameError(), Exception)
True
```

- `issubclass` fragt die Klassenhierarchie ab.

```
issubclass(NameError, Exception)
issubclass(int, object)
```

Manuelle Klassenerzeugung

- `type(name,superclasses,attributes)` erzeugt ein Klassenobjekt mit Namen `name`, Vaterklassen `superclasses`, und Attributen `attributes`.
- `C = type('C',(),{})` entspricht `class C: pass`.
- Methoden können als Attribute übergeben werden.

```
def f (self, coord):
  self.coord = coord

Vec = type('Vec, (object,), {'__init__' : f})
```

- Manuelle Klassenerzeugung nützlich für Meta-Programmierung.

Eigenschaften

- *Properties* sind Felder, für die Lesen, Schreiben, und Löschen besonders definiert ist.
- Konstruktion:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

```
class Rectangle(object):
  def __init__(self, width, height):
    self.width  = width
    self.height = height
  area = property(
    lambda self: self.width * self.height,
    doc="Rectangle area (read only).")

Rectangle(5,2).area
```

Feldzugriff kontrollieren

- Der Zugriff auf Attribute kann komplett neu definiert werden.
- Dazu die folgende Methoden überschreiben.

```
__getattribute__(self, attr)
__setattr__(self, attr, value)
__delattr__(self, attr)
```

- Beispiel: Listen ohne Append

```
class listNoAppend(list):
  def __getattribute__(self, name):
    if name == 'append': raise AttributeError
    return list.__getattribute__(self, name)
```

Statische Methoden

- Einer Klasse können Methoden gegeben werden, die nicht auf die Instanz zugreifen.
  - Klassenmethoden können auf Klassenattribute zugreifen.
  - Statische Methoden nicht einmal das.

```
class Static:
  # static method
  def __bla(): print "Hello, world!"
  hello = staticmethod(__bla)
```

- Aufruf aus Klassen- und Instanz-Sicht möglich.

```
Static.hello()
Static().hello()
```

Klassenmethoden

- Eine Klassenmethode nimmt als erstes Argument eine Klassenobjekt anstatt einer Instanz.

```
class Static:
  val = 5
  # class method
  def sqr(c): return c.val * c.val
  sqr = classmethod(sqr)

Static.sqr()
Static().sqr()
```

- Es ist üblich, wie oben die ursprüngliche Definition von `sqr` zu überschreiben.

Dekoration von Funktionen

- Für das Muster

```
def f(args): ...
f = modifier(f)
```

gibt es eine eigene Syntax:

```
@modifier
def f(args): ...
```

- Beispiele für Modifikatoren: Memoisation, Typprüfung.

Interpretation

- Strings können mittels der Funktion `eval` als Python-Ausdrücke ausgewertet werden.

```
>>> x = 5
>>> eval ('x')
5
>>> f = lambda x: eval('x * x')
>>> f(4)
16
```

- Der Befehl `exec` führt Anweisungen aus:

```
>>> exec 'print x'
5
```

Übersetzung

- Übersetzung von Strings byte-code.

```
c = compile('map(lambda x:x*2,range(10))', # code
  'pseudo-file.py',       # filename for error msg
  'eval')  # or 'exec' (module) or 'single' (stm)
eval(c)
```

- Achtung Einrückung!

```
>>> c2 = compile('''
... def bla(x):
...   print x*x
...   return x
... bla(5)
... ''', 'pseudo', 'exec')
>>> exec c2
25
```

Überladen

- Operatoren wie `+`, `<=` und Funktionen wie `abs`, `str` und `repr` können für eigene Typen definiert werden.

```
class Vector(object):
  def __init__(self, coord):
    self.coord = coord
  def __str__(self):
    return str(self.coord)

v1 = Vector([1,2,3])
print v1
```

Überladen

```
import math      # sqrt
import operator # operators as functions

class Vector(object):
  ...
  def __abs__(self):
    '''Vector length (Euclidean norm).'''
    return math.sqrt(sum(x*x for x in self.coord))
  def __add__(self, other):
    '''Vector addition.'''
    return map(operator.add, self.coord, other.coord)

print abs(v1)
print v1 + v1
```

Überladen nicht-symmetrischer Operationen

- Skalarmultiplikation für Vektoren kann entweder als `v1 * 5` oder `5 * v1` geschrieben werden.

```
class Vector(object):
  ...
  def __mul__(self, scalar):
    'Multiplication with a scalar from the right.'
    return map(lambda x: x*scalar, self.coord)

  def __rmul__(self, scalar):
    'Multiplication with a scalar from the left.'
    return map(lambda x: scalar*x, self.coord)
```

- `v1 * 5` ruft `v1.__mul(5)` auf.
- `5 * v1` ruft `v1.__rmul(5)` auf.

Überladen von Indizierung

- Auch die Index- und Abschnittsnotation ist überladbar.

```
class Vector(object):

  def __getitem__(self, index):
    '''Return the coordinate with number index.'''
    return self.coord[index]

  def __getslice__(self, left, right):
    '''Return a subvector.'''
    return Vector(self.coord[left:right])

print v1[2]
print v1[0:2]
```

Übung

- Definieren Sie eine Klasse `Matrix` und überladen Sie die Operationen `+` und `*` für Matrizen.
- Definieren Sie weitere Operationen für Matrizen, wie `m.transpose()`, `str(m)`, `repr(m)`.