Python-Crashkurs

Andreas Abel

Lehrstuhl für Theoretische Informatik
Ludwig-Maximilians-Universität München

Wintersemester 2008/09
6.-10. Oktober 2008

Klassendefinition

- Klassendefinition folgen dieser Syntax:

```
class ClassName (SuperClass_1, ..., SuperClass_n):
  statement_1
  ...
  statement_m
```

- Die Ausführung der Klassendefinition erzeugt ein Klassenobjekt, referenziert durch Variable `ClassName`.
- Die Anweisungen `statement_i` erzeugen Klassenattribute.
- Zusatzlich werden die Attribute der Elternklassen `SuperClass_i` geerbt.
- Klassenobjekte sind wie Funktionen aufrufbar (*callable*).
- Ein Aufruf eines Klassenobjektes erzeugt eine Instanz.

Klassenattribute

- Erzeuge eine Klasse mit 2 Attributen: Eine Klassenvariable und eine Methode.

```
class C:
  "Purpose-free demo class."
  classVar1 = 42
  def method1 (self):
    "Just a random method."
    print "classVar1 = %d" % C.classVar1

X = C                    # alias the class object
x = X()                  # create an instance of C
X.method1(x)             # call method (class view)
x.method1()              # call method (instance view)
```

- `dir(C)` listet alle Attribute der Klasse.

Nachträgliches Setzen von Klassenattributen

- Eine Klasse ist im wesentlichen ein Dictionary seiner Attribute.
- Attribute kann man nachträglich hinzufügen/modifizieren (ist aber kein guter Stil!).

```
class D: pass                  # empty class object

def method(self):              # just a function
  print D.classVar             # not-yet existing attribute
  print D.__dict__['classVar']   # same effect
  print self.classVar            # ditto

d = D()                        # create an instance
D.method = method              # add new class attributes
D.classVar = 42
d.method()                     # prints 42 (thrice)
```

Instanzenvariablen

- Klasse von Binärbäumen

```
class BinTree:
  "Binary trees."
  def __init__(self, label, left=None, right=None):
    self.left = left
    self.label = label
    self.right = right
  def inorder(self):
    if self.left != None: self.left.inorder()
    if self.label != None: print self.label,
    if self.right != None: self.right.inorder()
```

- `__init__` ist Konstruktor, erzeugt Instanzenattribute.
- Auch innerhalb der Methode nur qualifizierter Zugriff `self.attr`.

Instanzenattribute

- Instanzenattribute können auch nachträglich gesetzt werden.

```
x = C()
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

- `x.__class__` verweist auf das Klassenobjekt von `x`.
- `x.__dict__` listet die Attribute von `x`.
- `dir(x)` listet den Namensraum von `x`.

Methodenobjekte

- *Gebundene Methoden* kennen schon die Instanz, auf der sie arbeiten sollen.

```
>>> c = C()
>>> c.method1
<bound method C.method1 of <__main__.C instance at
>>> c.method1()
```

- *Ungebundene Methoden* benötigen die Instanz noch als zusätzliches erstes Argument.

```
>>> C.method1
<unbound method C.method1>
>>> C.method1(c)
```

Vererbung

- Einfachvererbung:

```
class EmptyTree(BinTree):
  def __init__ (self):
    BinTree.__init__ (self, None)

class Leaf(BinTree):
  def __init__ (self, label):
    BinTree.__init__ (self, label)

l1 = Leaf(6)
l1.printinorder()
```

- Konstruktor der Überklasse muss explizit aufgerufen werden.

Vererbung

- Unterklassen können neue Attribute hinzufügen.

```
class MemberTree(BinTree):
  def member(self, x):
    return bool(self.label == x or
                (self.left and self.left.member(x)) or
                (self.right and self.right.member(x)))
```

- Konstruktor `__init__` wird geerbt.
- Mehrfachvererbung `class C(C1,C2,...,Cn)`: Klassenattribute werden erst in `C` selbst gesucht, dann rekursiv (Tiefensuche) in `C1`,...,`Cn`.

Überschreiben

- Attribute können in Unterklassen überschrieben werden.
- Falls der Baum sortiert ist, kann man in logarithmischer Zeit suchen:

```
class SearchTree(MemberTree):
  '''Ordered binary tree.'''
  def member(self, x):
    return bool(self.label == x or
                (self.label > x and
                 self.left and self.left.member(x)) or
                (self.label < x and
                 self.right and self.right.member(x)))
```

Private Variablen

- Attribute der Form `__ident` sind Klassen-privat.
- Werden intern umbenannt in `_ClassName__ident`.

```
class Bla():
  __privateVar = 4
  def method(self):
    print self.__privateVar
    print self.__class__.__dict__[
      '_Bla__privateVar']

b = Bla()
b.method()                    # prints 4 (twice)
```

Ausnahmen

- Eine Ausnahmen kann man mit `try...except...` abfangen.

```
while True:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print "Not a valid number.  Try again..."
```

- Abfangen mehrerer Ausnahmen in einem `except`-Block.

```
except (RuntimeError, TypeError, NameError):
  pass
```

Ausnahmen

- Verschiedene Fehlerbehandlungsroutinen

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Ausnahmen: else

- Falls keine Ausnahme ausgelöst wurde, wird der optionale `else`-Teil ausgeführt.

```
for arg in sys.argv[1:]:
  try:
    f = open(arg, 'r')
  except IOError:
    print 'cannot open', arg
  else:
    print arg, 'has', len(f.readlines()), 'lines'
    f.close()
```

Raising Exceptions

- `raise Ex[, info]` wirft eine Ausnahme.
- `raise` wirft die eben gefangene Ausnahme nochmal.

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

Aufräumaktion

- Der Kode im `finally`-Block wird ausgeführt bei normaler Beendigung und auch vor einer Fehlerbehandlung.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Ausnahmen: Alle Elemente

- Hier ein `try`-Konstrukt mit allen features.

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print "division by zero!"
    else:
        print "result is", result
    finally:
        print "executing finally clause"
```

Vordefinierte Aufräumaktion

- `with` sorgt für Aufräumaktion (hier: Datei schliessen) im Fehlerfall.

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

- Vor Version 2.6 muss dieses Feature manuell aktiviert werden:

```
from __future__ import with_statement
```

Benutzerdefinierte Ausnahmen

- Der Benutzer kann sich eine Hierarchie von Ausnahmen definieren.
- Dies sind einfach Klassen.
- Sollten (indirekt) von `Exception` abgeleitet sein.
- Standardmässig speichert `__init__` die Argumente nach `args`.
- Werfen mit `raise Class, instance` (`instance` Instanz (einer Unterklasse) von `Class`).
- Oder `raise instance`, kurz für:

  ```
  raise instance.__class__, instance
  ```

- `instance` steht bei Bedarf für `instance.args`, e.g. `print instance`.

Benutzerdefinierte Ausnahmen

- Standardbehandlung von Argumenten kann abgeändert werden.
- Hier: benutze Attribut `value` statt `args`.

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError, e:
    print 'My exception occurred, value:', e.value
```

- Mit dem Konstruktor muss dann auch die Repräsentationsfunktion `__str__` neu definiert werden.

Benutzerdefinierte Ausnahmen

- Der folgende Kode druckt B, B, D (denn `except B` greift auch bei der Unterklasse `C` von `B`.

```
class B:    pass
class C(B): pass
class D(C): pass

for c in [B, C, D]:
    try: raise c()
    except D: print "D"
    except B: print "B"
    except C: print "C"
```

Iterator unter dem Mikroskop

- `it = iter(obj)` liefert den Iterator für das Objekt `obj`.
- `it.next()` liefert das nächste Element
- oder wirft eine `StopIteration`-Ausnahme.

Iterator selbst gebastelt

- Iterierbare Klasse: hat `__iter__()`-Methode, die ein Objekt mit einer `next()`-Methode liefert.

```
class Reverse:
    "Iterator for looping over sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0: raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Generatoren

- Eine Funktion, die `yield`-Anweisungen enthält, ist ein Generator.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

- Generatoren lassen sich iterieren.

```
>>> for char in reverse('golf'): print char,
...
f l o g
```

Generator-Ausdrücke

- Ähnlich Listen-Komprehension

```
>>> sum(i*i for i in range(10))
285
>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))
260
>>> unique_words = set(word
                       for line in page
                       for word in line.split())
```