Python höherer Ordnung

Marek Kubica

μPy

12. März 2009

Warum höhere Ordnung

Funktionale Programmierung

- viel Arbeit mit Funktionen
- Funktionen die Funktionen transformieren
- so wenig Zustand (State) wie möglich

Vorteile

- Klarer, kürzerer Code
- Elegante Lösungen

Warum höhere Ordnung

Funktionale Programmierung

- viel Arbeit mit Funktionen
- Funktionen die Funktionen transformieren
- so wenig Zustand (State) wie möglich

Vorteile

- Klarer, kürzerer Code
- Elegante Lösungen
- Spaß!

Dekoratoren

Was ist das?

- diese Teile mit der @-Syntax (Python 2.4)
- werden genutzt um Funktionen zu verarbeiten
- Vorher auch möglich, nur ohne @-Syntax
- fast wie Unix-Pipes nur mit Funktionen statt Strings
- ab Python 3.x gibt es Klassendekoratoren

Ein einfacher Dekorator

```
def logged(fun):
    def _inner(*args, **kwargs):
        print 'Function %s entry' % fun.__name__
        fun(*args, **kwargs)
        print 'Function %s exit' % fun.__name__

    return _inner

@logged
def foo():
    print 'Doing something'

foo()
```

Probleme

Funktionssignatur

Siehe `help(foo)`

- Falscher Name (`_inner`)
- Falsche Parameter (`*args, **kwargs`)
- Falscher Docstring

Lösung

`functools.update_wrapper`

Etwas ausgebessert

```
import functools

def logged(fun):
    def _inner(*args, **kwargs):
        print 'Function %s entry' % fun.__name__
        fun(*args, **kwargs)
        print 'Function %s exit' % fun.__name__
    functools.update_wrapper(_inner, fun)
    return _inner

@logged
def foo():
    print 'Doing something'

foo()
```

Parameter dennoch falsch

functools.wraps

Ein Dekorator für Dekoratoren

```
import functools

def logged(fun):
    @functools.wraps(fun)
    def _inner(*args, **kwargs):
        print 'Function %s entry' % fun.__name__
        fun(*args, **kwargs)
        print 'Function %s exit' % fun.__name__
    return _inner

@logged
def foo():
    print 'Doing something'

foo()
```

Dekoratoren mit Parametern

```
import sys
def logged(stream):
    def _inner(fun):
        def _wrapper(*args, **kwargs):
            stream.write('Function %s entry\n' %
                    fun.__name__)
            fun(*args, **kwargs)
            stream.write('Function %s exit\n' %
                    fun.__name__)
        return _wrapper
    return _inner

@logged(stream=sys.stderr)
def foo():
    print 'Doing something'
foo()
```

Mehr Spaß mit Dekoratoren

Das `decorator`-Modul

- Bietet noch eine Menge Features über `functools` hinaus
- Behält bei Dekoratoren die Signatur bei
- Dekorator-Dekoratoren
- wird von TurboGears 2 verwendet
- 16-seitige Dokumentation
- `http://pypi.python.org/pypi/decorator`

`DecoratorTools`

- Dekorator-Syntax (aber nicht `@decorator`) für Python 2.3
- Achtung: PJE-Ware ☺
- `http://peak.telecommunity.com/DevCenter/DecoratorTools`

with-Statement

Was isses?

- Syntaktischer Zucker für `try/except/finally`
- neu in Python 2.5, Standard in Python 2.6
- `from __future__ import with_statement`
- Eignen sich um Ressourcen vorzubereiten und freizugeben

with-Statement

Was isses?

- Syntaktischer Zucker für `try/except/finally`
- neu in Python 2.5, Standard in Python 2.6
- `from __future__ import with_statement`
- Eignen sich um Ressourcen vorzubereiten und freizugeben (Also wie Konstruktoren und Destruktoren, nur dass sie funktionieren)
- ein neues *Protokoll*

Beispiele für with-Statement

- Dateizugriff
- Threads
- Datenbankzugriff

Beispiel mit with

```
with open('/etc/passwd', 'r') as handle:
    for line in handle:
        print line,
```

Beispiel ohne with

```
handler = open('/etc/passwd', 'r')
try:
    for line in handler:
        print line,
finally:
    handler.close()
```

Context Manager

Was ist das?

- Ist grundsätzlich nur ein *Protokoll*
- Das Protokoll ist kompatibel zu alten Python-Versionen
- Ein Context Manager-Objekt hat `__enter__` und `__exit__`
- `__enter__` wird vor dem Betreten des `with`-Blocks ausgeführt
- `__exit__` wird nach dem Verlassen ausgeführt
- Objekte in Stdlib unterstützen es bereits

Die contextlib

contextlib.closing

- Klassen die vor Python 2.5 da waren haben kein `__enter__`/`__exit__`
- aber oftmals ein `close()`
- `contextlib.closing` wrappt Instanzen zu Context Managern

contextlib.nested

- Wenn man mehrere Dateien öffnen will (etwa Eingabe und Ausgabe-Datei)
- Verschachtelung nötig, aber recht hässlich
- `contextlib.nested` verbindet mehrere Context Manager zu einem
- Nur ein `with`-Statement nötig

functools.partial

Eine Funktion die Parameter in Funktionen belegt

- Eingabe: Funktion, Parameter die belegt werden sollen (positional und keyword)
- Ausgabe: Funktion, mit weniger Parametern
- Nutzen: Macht einige `lambdas` und Wrapperfunktionen überflüssig

```
next = property(partial(get_object_at_offset, offset=1))
```

Das itertools-Modul

Tools für Generatoren

Vorteil von Generatoren gegenüber Listen: Prozessor- und Speicherschonend.

- `imap`: Generator-Version von `map`. Ruft eine Funktion auf jedem Element eines *Iterable* auf und gibt dessen Rückgabe in einen Generator
- `izip`: Generator-Version von `zip`. Verbindet mehrere *Iterables* zu einem Generator.
- `count`: Generator der eine unendliche Reihe zurückgibt (0, 1, 2, 3...) oder (5, 6, 7, 8...) etc.
- `cycle`: Generator der ein *Iterable* durchläuft und dessen Werte zurückgibt. Nach dem letzen Wert fängt er wieder von vorne an
- Beispiel folgt, aber zunächst erstmal das `operator`-Modul

Das operator-Modul

Die Python-Operatoren als Funktionen

Das Eldorado für funktionale Programmierer in Python

- Bietet alle Operatoren wie `+`, `-`, `*` als Funktionen
- Etwa `add`, `sub`, `mul`
- Auch exotischere Sachen wie `itemgetter`, `attrgetter` und `truth`

Wozu sind die gut?

- Können nun mit den `itertools` kombiniert werden, die Funktionen erwarten
- Können mit `partial` kombiniert werden
- Können in `dicts` gesteckt werden

Das Beispiel

```
from operator import mul
from itertools import izip, imap, count, cycle
from functools import partial

lang = ['Python', 'Scheme', 'Haskell']
nums = izip(imap(partial(mul, 2), count()), lang)
ages = izip(cycle(['new', 'old']), lang)
print list(nums)
print list(ages)

# output
# [(0, 'Python'), (2, 'Scheme'), (4, 'Haskell')]
# [('new', 'Python'), ('old', 'Scheme'),
#  ('new', 'Haskell')]
```

Sortier-Beispiel

```
from operator import itemgetter

unsorted = [(3, 1, 'Python'), (1, 3, 'Haskell'),
        (2, 1, 'Scheme')]
# default: sorts by first item
print sorted(unsorted)
# custom: sorts by second item
print sorted(unsorted, key=itemgetter(1))

# output
# [(1, 3, 'Haskell'), (2, 1, 'Scheme'), (3, 1, 'Python')]
# [(3, 1, 'Python'), (2, 1, 'Scheme'), (1, 3, 'Haskell')]
```

Noch mehr itertools

```
from operator import eq
from functools import partial
from itertools import takewhile

import functools, itertools, operator
s = 'ZZZ123ZZZZ'
print len(list(takewhile(partial(eq, 'Z'), s)))
print len(list(takewhile(partial(eq, 'Z'), reversed(s))))

# output
# 3
# 4
```

`takewhile` hat auch noch einen Bruder, `dropwhile`

any() & all()

Quantoren

- `any`: Der Existenzquantor $\exists$; gibt `True` zurück wenn *irgendein* Wert `True` war, ansonsten `False`
- `all`: Der Allquantor $\forall$; gibt `True` zurück, wenn *alle* Werte `True` waren, ansonsten `False`

Nutzen

Wenn man viele gleichartige Daten auf irgendeine Eigenschaft untersuchen will.

Beispiele aus dem echten Leben

```
from os import listdir
from os.path import exists
from itertools import imap

print all(imap(exists, ['/etc/passwd',
    '/usr/bin/python']))

print any(item.endswith('.py') for item in listdir('.'))
```

sum()

Generische Summations-Funktion

- Nimmt als ersten Parameter ein *Iterable*
- summiert jedes Element des Iterables auf
- Zweiter, optionaler Parameter: Startwert
- Funktioniert mit allen Datentypen auf denen + definiert ist
- Achtung: Strings ausgenommen (“There should be one...”)

Auch hier noch ein Beispiel

```
from os import listdir
from itertools import imap

print sum(int(i) for i in "123456")
print sum(imap(int, "123456"))
print sum(1 for e in listdir('.') if e.endswith('.py'))
print sum([["Python", "Smalltalk"],
    ["Scheme", "Haskell"]], [])
```

Fertig und aus

Quellen und Inspirationen

- Forum – einige hübsche Beispiele
- Autoren der `functools` & `itertools`

Was gibt es noch in der Richtung?

- Klassendekoratoren
- Metaklassen
- Continuations (Stackless)
- Tail-Call Optimization (IronPython)