Informatik I

24. Funktionale Programmierung in Python

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

21.01.2014

Informatik I

21.01.2014 — 24. Funktionale Programmierung in Python

24.1 Was ist funktionale Programmierung?

Programmierparadigmen

- Man unterscheidet bei der Programmierung und bei Programmiersprachen verschiedene Programmierparadigmen oder Programmierstile.
- Zuerst einmal gibt es eine generelle Unterscheidung in:
  - imperative Programmierung: Man beschreibt, wie etwas erreicht werden soll;
  - deklarative Programmierung: Man beschreibt, was erreicht werden soll.

Imperative Programmierparadigmen

- Imperative Programmierparadigmen:
  - Allen imperativen Programmierstilen ist gemeinsam, dass der Zustand der Berechnung explizit repräsentiert und modifiziert wird (Variablen und Zuweisungen).
  - Prozedurale Programmierung, wie wir sie im ersten Teil der Vorlesung kennen gelernt haben: Die Aufgabe wird in kleinere Teile – Unterprogramme – zerlegt, die auf den Daten arbeiten.
    Beispielsprachen: PASCAL, C
  - Objekt-orientierte Programmierung: Im Gegensatz zur prozeduralen Programmierung bilden Daten und die darauf arbeitenden Unterprogramme eine Einheit. Ihre Struktur wird durch Klassen beschrieben, die die Wiederverwendbarkeit unterstützen.
    Beispielsprachen: JAVA, C++.

Deklarative Programmierparadigmen

- Deklarative Programmierparadigmen:
  - Allen deklarativen Programmierstilen ist gemeinsam, dass kein Berechungs-Zustand explizit repäsentiert wird.
  - Logische Programmierung: Man beschreibt das Ziel mit Hilfe einer logischen Formel, z.B. in PROLOG.
  - Funktionale Programmierung: Man beschreibt das Ziel durch Angabe von (mathematischen) Funktionen, wie z.B. in Haskell, ML, LISP
  - Abfragesprachen wie SQL oder XQuery sind auch deklarative Programmiersprachen, allerdings nur für Spezialzwecke einsetzbar.
  - Gleiches gilt für viele Auszeichnungssprachen-Sprachen (*Markup*-Sprachen) wie HTML

Was ist funktionale Programmierung?

Wichtige Eigenschaften von funktionaler Programmierung (FP) sind:

- Funktionen sind Bürger erster Klasse (*first-class citizens*). Alles was man mit Daten machen kann, kann man mit Funktionen machen.
- Es gibt Funktionen höherer Ordnung – Funktionen, die auf Funktionen operieren, die womöglich auf Funktionen operieren.
- Rekursion ist die wesentliche Art, den Kontrollfluss zu organisieren.
- In funktionalen Programmiersprachen gibt es oft keine Anweisungen, sondern nur auswertbare Ausdrücke.
- In reinen funktionalen Sprachen gibt es keine Zuweisungen (und damit auch keine Seiteneffekte) $\rightarrow$ referentielle Transparenz: Eine Funktion gibt immer das gleiche Ergebnis bei gleichen Argumenten.

24.2 FP in Python

FP in Python

- Funktionen sind „Bürger erster Klasse“.
- Funktionen höherer Ordnung werden voll unterstützt.
- Viele andere Konzepte aus funktionalen Programmiersprachen werde unterstüzt, wie die Listen-*Comprehension*.
- In vielen funktionalen Programmiersprachen ist *Lazy Evaluation* ein wichtiger Punkt:
  - Die Auswertung von Ausdrücken wird solange verzögert, bis das Ergebnis benötight wird.
  - Damit lassen sich unendliche Sequenzen repräsentieren.
- Das Letztere unterstützt Python (und andere Sprachen) durch Iteratoren und Generatoren.

FP in Python: Defizite

Einige der Anforderungen an FP sind in Python nicht erfüllt:

- Referentielle Transparenz – kann man natürlich selbst erzwingen: Keine globalen Variablen nutzen, keine Mutuables ändern.
- Rekursion als wesentliche Steuerung des Kontrollflusses wird in Python nur eingeschränkt unterstützt: Keine Optimierung bei Endrekursion!
  - Beachte: Maximale Rekursionstiefe kann mit `sys.setrecursionlimit(n)` gändert werden.
  - Mit `sys.getrecursionlimit()` kann man sie abfragen.
- Ausdrücke statt Anweisungen: Wird in Python nicht unterstützt. Allerdings gibt es konditionale Ausdrücke!
  - *true-value* `if` *cond* `else` *false-value* hat den Wert *true-value*, falls *cond* wahr ist. Ansonsten hat der Ausdruck den Wert *false-value*.

Exkurs: Konditionale Ausdrücke

Konditionale Ausdrücke

```
>>> "a" if True else "b"
'a'
>>> "a" if False else "b"
'b'
>>> cond = True
>>> 2 * 3 if cond else 2 ** 3
6
>>> cond = False
>>> 2 * 3 if cond else 2 ** 3
8
>>> res = 2 * 3 if cond else 2 ** 3

>>> def mult_or_exp(cond):
...     return 2 * 3 if cond else 2 ** 3

>>> mult_or_exp(False)
8
```

24.3 Funktionen definieren und verwenden

Funktionsdefinition und -verwendung

- Funktionen existieren in dem Namensraum, in dem sie definiert wurden.

Python-Interpreter

```
>>> def simple():
...     print('invoked')
...
>>> simple # beachte: keine Klammern!
<function simple at 0x10ccbdcb0>
>>> simple() # Aufruf!
invoked
```

- Eine Funktion ist ein normales Objekt (wie andere Python-Objekte).
- Es kann zugewiesen werden, als Argument übergeben werden und als Funktionsresultat zurück gegeben werden.
- Und es ist aufrufbar ...

Funktionsverwendung

Python-Interpreter

```
>>> spam = simple; print(spam)
<function simple at 0x10ccbdcb0>
>>> def call_twice(fun):
...     fun(); fun()
...
>>> call_twice(spam) # Keine Klammern hinter spam
invoked
invoked
>>> def gen_fun()
...     return spam
...
>>> gen_fun()
<function simple at 0x10ccbdcb0>
>>> gen_fun()()
invoked
```

Wie stellt man fest, ob ein Objekt *aufrufbar* ist?

- Funktionsobjekte haben wie alle Instanzen eine Menge von Attributen, z.B. die magische Methode `__call__`.
- Teste, ob das Objekt das Attribut `__call__` besitzt!
- Funktion sobjekte sind Instanzen einer bestimmten Klasse, nämlich `collections.Callable`.

Python-Interpreter

```
>>> hasattr(spam, '__call__')
True
>>> import collections
>>> isinstance(spam, collections.Callable)
True
```

Klasseninstanzen aufrufbar machen

- Wir hatten gesehen, dass Funktionen einfach Objekte sind, die eine `__call__`-Methode besitzen.
- Was passiert, wenn wir eine Klasse mit dieser magischen Methode definieren?

→ In stanzen dieser Klasse werden aufrufbar!

Python-Interpreter

```
>>> class CallMe:
...     def __call__(self, msg=None):
...         if msg: print('called:', msg)
...         else: print('called')
...
>>> c = CallMe()
>>> c()
called
>>> c('hi')
called: hi
```

24.4 Lambda-Notation

Funktionen mit Lambda-Notation definieren

- Statt mit Hilfe der `def`-Anweisung eine benannte Funktion zu definieren, kann man mit dem `lambda`-Operator eine kurze, namenlose Funktionen definieren:

Python-Interpreter

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at 0x107cf4950>
>>> (lambda x, y: x * y)(3, 8)
24
>>> mul = lambda x, y: x * y
```

- Etwas andere Syntax
- Als Funktionskörper ist nur ein einziger Ausdruck (arithmetisch, Boolesch, . . .) zulässig!

Verwendung von Lambda-Funktionen (1)

Python-Interpreter

```
>>> def mul2(x, y):
...     return x * y
...
>>> mul(4, 5) == mul2(4, 5)
True
```

- `mul2` ist äquivalent zu `mul`!
- Lambda-Funktionen benutzt man gerne für
  - einfache Prädikatsfunktionen (Boolesche Tests)
  - einfache Konverter
  - Objektdestruktoren
  - Lazy Evaluation ermöglichen
  - Sortierordnung bestimmen

Verwendung von Lambda-Funktionen (2)

cookie_lib.py

```
# add cookies in order of most specific
# (ie. longest) path first
cookies.sort(key=lambda arg: len(arg.path),
      reverse=True)
```

- Ohne Lambda-Notation hätte man hier erst eine Funktion definieren müssen und dann benutzen können.
- Da die Funktion nur einmal benutzt wird und sehr klein ist, wäre das ziemlich umständlich.
- Weitere Beispiel kommen noch . . .
- Übrigens: Mit Lambda-Notation definierte Funktionen sind Seiteneffekt-frei (wenn die aufgerufenen Funktionen es sind)!

Verwendung von Lambda-Funktionen (3): Funktions-Fabriken

- Funktionen können ja Funktionen zurück geben. Zur Erzeugung der Funktion kann man natürlich Lambda-Ausdrücke benutzen.
- Beispiel: Ein Funktion, die einen Addierer erzeugt, der immer eine vorgegebene Konstante addiert:

Python-Interpreter

```
>>> def gen_adder(c):
...     return lambda x: x + c
...
>>> add5 = gen_adder(5)
>>> add5(15)
20
```

24.5 Funktionen höherer Ordnung: map, reduce und filter

map: Anwendung einer Funktion auf Listen und Iteratoren

- map hat mindestens zwei Argumente: eine Funktion und ein iterierbares Objekt.
- Es liefert einen Iterator zurück, der über die Anwendungen der Funktion auf jedes Objekt des übergebenen Arguments iteriert.

Python-Interpreter

```
>>> list(map(lambda x: x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Wird mehr als ein iterierbares Objekt angegeben, dann muss die Funktion entsprechend viele Argumente besitzen.

Python-Interpreter

```
>>> list(map(lambda x, y, z: x + y + z,
...   range(5), range(0, 40, 10), range(0, 400, 100)))
[0, 111, 222, 333]
```

Anwendungsbeispiel für map

- Wir wollen eine Liste c_list von Temperaturen von Celsius nach Fahrenheit konvertieren. Konventionell:

ctof.py

```
def ctof(temp):
    return ((9.0 / 5) * temp + 32)
def list_ctof(cl):
    result = []
    for c in cl:
        result.append(ctof(c))
    return result
f_list = list_ctof(c_list)
```

- Mit map:

ctof.py

```
list(map(lambda c: ((9.0 / 5) * c + 32), c_list))
```

reduce: Reduktion eines iterierbaren Objekts auf ein Element

- reduce ist eine weitere Funktion höherer Ordnung, die man oft in funktionalen Sprachen findet.
- Sie wendet eine Funktion mit zwei Argumenten auf ein iterierbares Objekt an.
- Es werden jeweils die ersten beiden Objekte genommen und zu einem Objekt reduziert, das dann das neue Anfangsobjekt ist.
- reduce wurde allerdings aus dem Sprachkern von Python 3 entfernt und findet sich nun im Modul functools.

Python-Interpreter

```
>>> from functools import reduce
>>> reduce(lambda x, y: x * y, range(1, 5))
24 # ((1 * 2) * 3) * 4
```

Anwendung von reduce (1)

- Guido von Rossum schrieb zu reduce:

  *This is actiually the one I've always hated most, because, apart from a few examples involving + and *, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what reduce() is supposed to do.*

- Hier ein nicht-triviales Beispiel:

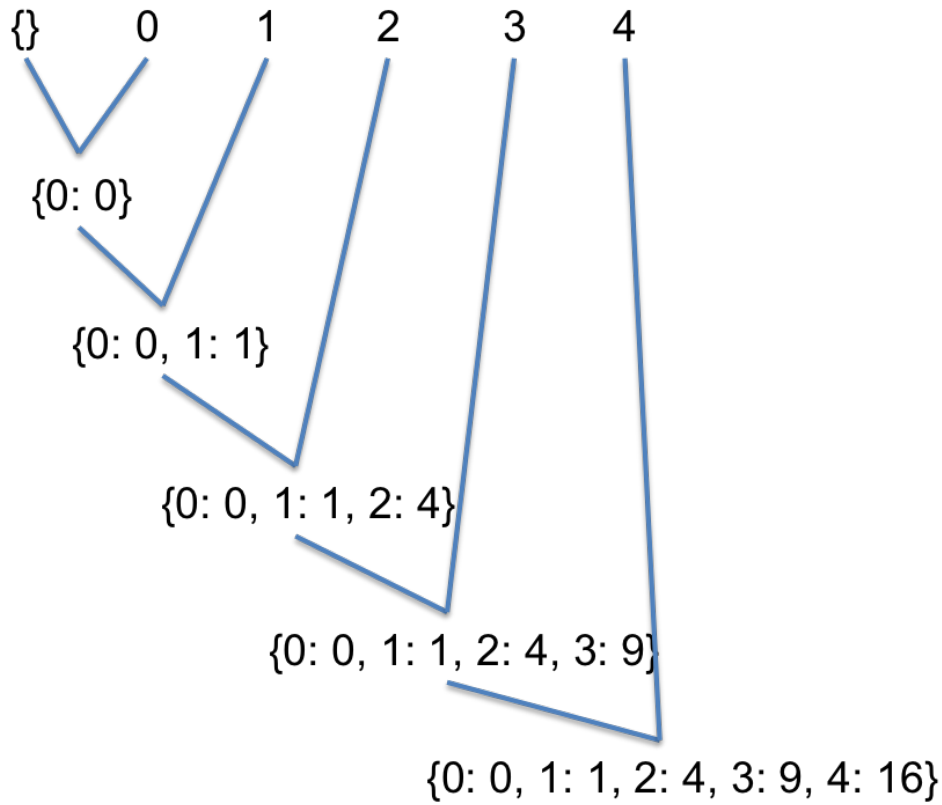
Python-Interpreter

```
>>> def to_dict(d, key):
...     d[key] = key**2
...     return d
...
>>> reduce(to_dict, [{}] + list(range(5)))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- Es wird also ein dict aufgebaut, das als Werte die Quadrate seiner Schlüssel enthält.

Anwendung von reduce (2)

- Was genau wird da schrittweise reduziert?



- Eleganter Aufbau des dicts.
- Allerdings ist dict-*Comprehension* (kommt noch) eleganter und lesbarer!

filter: Filtert nicht passende Objekte aus

- filter erwartet als Argumente eine Funktion mit einem Parameter und ein iterierbares Objekt.
- Es liefert einen Iterator zurück, der die Objekte aufzählt, bei denen die Funktion nicht False (oder äquivalente Werte) zurück gibt.

Python-Interpreter

```
>>> list(filter(lambda x: x > 0, [0, 3, -7, 9, 2]))
[3, 9, 2]
```

24.6 Listen-, Generator-, `dict`- und Mengen-*Comprehension*

Listen-*Comprehension*

- In Python 2.7 wurden die sogenannten *Comprehensions* (im Deutschen auch Abstraktionen) eingeführt (aus der funktionalen Programmiersprache Haskell entlehnt).
- Mit diesen kann man ähnlich wie mit `lambda`, `map` und `filter`, Listen u.a. deklarativ und sehr kompakt beschreiben.
- Der Stil ist ähnlich dem, den man in der mathematischen Mengenschreibweise findet: $\{x \in U : \phi(x)\}$ (alle x aus U, die die Bedingung ϕ erfüllen). Beispiel:

Python-Interpreter

```
>>> [str(x) for x in range(10) if x % 2 == 0]
['0', '2', '4', '6', '8']
```

- Bedeutung: Erstelle aus allen `str(x)` eine Liste, wobei `x` über das iterierbare Objekt `range(10)` läuft und nur die geraden Zahlen berücksichtigt werden.

Generelle Syntax von Listen-*Comprehensions*

```
[ expression for expr1 in seq1 if cond1
              for expr2 in seq2 if cond2
              ...
              for exprn in seqn if condn ]
```

- Die `if`-Klauseln sind dabei optional.
- Ist `expression` ein Tupel, muss es in Klammern stehen!
- Damit kann man ganz ähnliche Dinge wie mit `lambda`, `map`, `filter` erreichen.

Python-Interpreter

```
>>> [str(x) for x in range(10) if x % 2 == 0]
['0', '2', '4', '6', '8']
>>> list(map(lambda y: str(y), filter(lambda x: x%2 == 0,
range(10))))
['0', '2', '4', '6', '8']
```

Geschachtelte Listen-*Comprehensions* (1)

- Wir wollen eine zweidimensionale Matrix der Art `[[0,1,2,3],[0,1,2,3],[0,1,2,3]]` konstruieren.
- Imperative Lösung:

Python-Interpreter

```
>>> matrix = []
>>> for y in range(3):
...     matrix.append(list(range(4)))
...
>>> matrix
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

- Lösung mit Listen-*Comprehensions*:

Python-Interpreter

```
>>> [[x for x in range(4)] for y in range(3)]
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

Geschachtelte Listen-*Comprehensions* (2)

- Wir wollen `[[1,2,3],[4,5,6],[7,8,9]]` konstruieren.
- Imperativ:

Python-Interpreter

```
>>> matrix = []
>>> for rownum in range(3):
...     row = []
...     for x in range(rownum*3, rownum*3 + 3):
...         row.append(x+1)
...     matrix.append(row)
...
```

- Lösung mit Listen-*Comprehensions*:

Python-Interpreter

```
>>> [[x+1 for x in range(y*3, y*3 + 3)] for y in range(3)]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Listen-*Comprehensions*: Kartesisches Produkt

- Wir wollen das kartesische Produkt aus `[0, 1, 2]` und `['a', 'b', 'c']` erzeugen.
- Imperativ:

Python-Interpreter

```
>>> prod = []
>>> for x in range(3):
...     for y in ['a', 'b', 'c']:
...         prod.append((x, y))
...
```

- Lösung mit Listen-*Comprehensions*:

Python-Interpreter

```
>>> [(x, y) for x in range(3) for y in ['a','b','c']]
[(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c'),
(2, 'a'), (2, 'b'), (2, 'c')]
```

Generator-*Comprehension*

- Es gibt auch eine Variante der Listen-Comprehension, die die Liste nicht explizit aufbaut, sondern einen Iterator erzeugt, der alle Objekte nacheinander generiert.
- Einziger Unterschied zur Listen-*Comprehension*: Runde statt eckige Klammern: Generator-*Comprehension*.
- Diese können weggelassen werden, wenn der Ausdruck in einer Funktion mit nur einem Argument angegeben wird.

Python-Interpreter

```
>>> sum(x**2 for x in range(11))
385
```

- Ist Speichplatz-schonender als `sum([x**2 for x in range(11)])`.

Comprehension für Dictionaries, Mengen, etc. (1)

Comprehension-Ausdrücke lassen sich auch für Dictionaries, Mengen, etc. verwenden. Nachfolgend ein paar Beispiele:

Python-Interpreter

```
>>> evens = set(x for x in range(0, 20, 2))
>>> evenmultsofthree = set(x for x in evens if x % 3 == 0)
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Lorem ipsum'
>>>  res = set(x for x in text if x >= 'a')
>>>  print(res)
{'e', 'i', 'm', 'o', 'p', 'r', 's', 'u'}
>>>  d = dict((x, x**2) for x in range(1, 10))
>>>  print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Comprehension für Dictionaries, Mengen, etc. (2)

Python-Interpreter

```
>>> sqnums = set(x for (_, x) in d.items())
>>> print(sqnums)
{64, 1, 36, 81, 9, 16, 49, 25, 4}
>>> dict((x, (x**2, x**3)) for x in range(1, 10))
{1: (1, 1), 2: (4, 8), 3: (9, 27), 4: (16, 64), 5: (25,
125), 6: (36, 216), 7: (49, 343), 8: (64, 512), 9: (81,
729)}
>>> dict((x, x**2) for x in range(10)
...       if not x**2 < 0.2 * x**3)
0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25
```

Comprehension für Dictionaries, Mengen, etc. (3)

Mit `all` und `any` kann man über die Elemente eines iterierbaren Objekts oder eines Iterators quantifizieren:

- `all(iterable)` evaluiert zu `True` gdw. alle Elemente äquivalent zu `True` sind (oder das `iterable` leer ist).
- `any(iterbale)` ist `True` wenn ein Element äquivalent zu `True` ist.

Python-Interpreter

```
>>>  text = 'Lorem ipsum'
>>>  all(x.strip() for x in text if x < "b")
False
>>>  any(x.strip() for x in text if x < "b")
True
>>>  all(x for x in text if x > "z")
True
>>>  any(x for x in text if x > "z")
False
```

24.7 Dekoratoren

Dekoratoren

Dekoratoren sind Funktionen (*Callables*), die Funktionen (*Callables*) als Parameter nehmen und zurückgeben. Sie werden verwendet, um andere Funktionen oder Methoden zu „umhüllen“.
Dekoratoren, die uns schon früher begegnet sind: `staticmethod`, `classmethod`, `property`, etc.
Es gibt eine spezielle Syntax, um solche Dekoratoren anzuwenden. Falls der Dekorator `wrapper` definiert wurde:

```
def fct(*args):
    # do some stuff
fct = wrapper(fct)
```

können wir auch schreiben:

```
@wrapper
def fct(*args):
    # do some stuff
```

Dekoratoren: property, staticmethod (1)

decorators.py

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

    def getname(self):
        return self._name

    # def setname(self, x):
    #     self._name = 2 * x
    name = property(getname)

    def hello():
        print("Hello world")
    hello = staticmethod(hello)
```

lässt sich also mittels der @wrapper-Syntax schreiben . . .

Dekoratoren: property, staticmethod (2)

decorators.py

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

    @property
    def name(self):
        return self._name

    # @name.setter
    # def name(self, x):
    #     self._name = 2 * x

    @staticmethod
    def hello():
        print("Hello world")
```

Einen Dekorator selbst definieren (1)

Wie definiert man selbst einen solchen Dekorator?

Angenommen, wir wollen den Aufruf einer Funktion (mit seinen Argumenten) auf der Konsole ausgeben. Eine Lösung, bei der wie die Funktion direkt mit ein paar Zeilen Code erweitern

decorators.py

```
verbose = True

def mult(x, y):
    if verbose:
        print("--- a nice header ---------")
        print("--> call mult with args: %s, %s" % x, y)
    res = x * y
    if verbose:
        print("--- a nice footer ---------")
    return res
```

Das ist hässlich! Wir wollen eine generische Lösung . . .

Einen Dekorator selbst definierenm (2)

Eleganter ist die folgende Lösung:

decorators.py

```
def decorator(f):
    def wrapper(*args, **kwargs):
        print("--- a nice header ---------")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer ---------")
        return res
    # print("--> wrapper now defined")
    return wrapper

@decorator
def mult(x, y):
    return x * y
```

Einen Dekorator selbst definieren (3)

Angenommen, wir wollen (ferner) messen, wie lange die Ausführung einer Funktion dauert.

Auch dies kann man mit einem Dekorator einfach implementieren:

decorators.py

```
import time

def timeit(f):
    def wrapper(*args, **kwargs):
        print("--> Start timer")
        t0 = time.time()
        res = f(*args, **kwargs)
        delta = time.time() - t0
        print("--> End timer:  %s sec." % delta)
        return res
    return wrapper
```

Einen Dekorator selbst definieren (4)

Wir können nun nicht nur einmalig Funktionen „transformieren“, sondern solche Transformationen auch hintereinander schalten:

decorators.py

```
@decorator
@timeit
def sub(x, y):
    return x - y

print(sub(3, 5))
```

liefert z.B.:

decorators.py

```
--- a nice header ---------
--> call wrapper with args: 3,5
--> Start timer
--> End timer:  2.145767211914062e-06 sec.
```

Dekoratoren: docstring und __name__ (1)

- Beim Dekorieren ändert sich ja der interne Name und der docstring.
- Allerdings könnte man ja ...

decorators.py

```
def decorator(f):
    def wrapper(*args, **kwargs):
        print("--- a nice header ---------")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer ---------")
        return res
    wrapper.__name__ = f.__name__
    wrapper.__doc__ = f.__doc__
    return wrapper
```

- ... die Funktionsattribute übernehmen.

Dekoratoren: docstring und __name__ (2)

- Zur Übernahme der Attribute gibt es natürlich schon einen Python-Dekorator

decorators.py

```
import functools
def decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print("--- a nice header ---------")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer ---------")
        return res
    return wrapper
```

Parametrische Dekoratoren (1)

- Wir wollen annehmen, dass wir einen Dekorator benötigen, der alle Stringausgaben auf 5 Zeichen beschränkt:

decorators.py

```
def trunc(f):
    def wrapper(*args, **kwargs):
        res = f(*args, **kwargs)
        return res[:5]
    return wrapper

@trunc
def data():
    return 'foobar'
```

- EIn aktueller Aufruf:

Python-Interpreter

```
>>> data()
'fooba'
```

Parametrische Dekoratoren (2)

- Manchmal sollen es 3 Zeichen sein, manchmal 6!
- Eine Dekorations-erzeugende Funktion:

decorators.py

```
def limit(length):
    def decorator(f):
        def wrapper(*args, **kwargs):
            res = f(*args, **kwargs)
            return res[:length]
        return wrapper
    return decorator

@limit(3)
def data_a():
    return 'limit to 3'

@limit(6)
def data_b():
    return 'limit to 6'
```

Parametrische Dekoratoren (3)

- Was passiert hier?
- Der Aufruf von `limit(3)` erzeugt einen Dekorator, der auf `data_a` angewandt wird; `limit(6)` wenden wir auf `data_b` an:

Python-Interpreter

```
>>> data_a()
'lim'
>>> data_b()
'limit '
```

- Aber was passiert hier eigentlich bei der geschachtelten Definition von Funktionen?

24.8 Funktionsschachtelung, Namensräume und Skopus

Geschachtelte Funktionsdefinitionen

- Im letzten Abschnitt sind uns geschachtelte Funktionsdefinitionen begegnet.
- Es ist dabei nicht immer klar, auf was sich ein bestimmter Variablenname bezieht.
- Um das zu verstehen, müssen wir die Begriffe Namensraum (*name space*) und Skopus oder Gültigkeitsbereich (*scope*) verstehen.
- Dabei ergeben sich zum Teil interessante Konsequenzen für die Lebensdauer einer Variablen.

Namensräume

- Ein Namensraum ist eine Abbildung von Namen auf Werte (innerhalb von Python oft durch ein `dict` realisiert).
- Innerhalb von Python gibt es:
  - den `Built-in`-Namensraum (`__builtins__`), in dem alle vordefinierten Funktionen und Variablen eingeführt werden.
  - den Namensraumen von Modulen, die importiert werden;
  - den globalen Namensraum (des Moduls `__main__`);
  - den lokalen Namensraum innerhalb einer Funktion;
  - Namensräume haben verschiedene Lebensdauern; der lokale Namensraum einer Funktion existiert z.B. normalerweise nur während ihres Aufrufs.

  → Namensräume sind wie Telefonvorwahlbereiche. Sie sorgen dafür, dass gleiche Namen in verschiedenen Bereichen nicht verwechselt werden.
  - Auf gleiche Variablennamen in verschiedenen Namensräumen kann meist mit der Punkt-Notation zugegriffen werden.

Gültigkeitsbereiche

- Der Skopus (oder Gültigkeitsbereich) einer Variablen ist der textuelle Bereich in einem Programm, in dem die Variable ohne die Punkt-Notation zugegriffen werden kann – d.h. wo sie sichtbar ist.
- Es gibt eine Hierarchie von Gültigkeitsbereichen, wobei der innere den äußeren normalerweise überschreibt!
- Wird ein Variablennamen gefunden, so wird nacheinander versucht:
  - ihn im lokalen Bereich aufzulösen;
  - ihn im nicht-lokalen Bereich aufzulösen;
  - ihn im globalen Bereich aufzulösen;
  - ihn im `Builtin`-Namensraum aufzulösen.

- Gibt es eine Zuweisung im aktuellen Skopus, so wird von einem lokalen Namen ausgegangen (außer es gibt andere Deklarationen):
  - „`global` *varname*“ bedeutet, dass *varname* in der globalen Umgebung gesucht werden soll.
  - „`nonlocal` *varname*“ bedeutet, dass *varname* in der nicht-lokalen Umgebung gesucht werden soll, d.h. in den umgebenden Funktionsdefinitionen.
- Gibt es keine Zuweisungen, wird in den umgebenden Namensräumen gesucht.
- Kann ein Namen nicht aufgelöst werden, dann gibt es eine Fehlermeldung.

EIn Beispiel für Namensräume und Gültigkeitsbereiche (1)

scope.py

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
```

EIn Beispiel für Namensräume und Gültigkeitsbereiche (2)

Python-Interpreter

```
>>> scope_test()
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
>>> print(In global scope:", spam)
In global scope: global spam
```

24.9 Closures

Closures

Ein Closure (oder Funktionsabschluss) ist eine Funktion, bzw. eine Referenz auf eine Funktion, die Zugriff auf einen eigenen Erstellungskontext enthält. Beim Aufruf greift die Funktion dann auf diesen Erstellungskontext zu. Dieser Kontext (Speicherbereich, Zustand) ist außerhalb der Funktion nicht referenzierbar, d.h. nicht sichtbar.
Closure beinhaltet zugleich Referenz auf die Funktion und den Erstellungskontext - die Funktion und die zugehörige Speicherstruktur sind in einer Referenz untrennbar abgeschlossen (closed term).

Wikipedia

Closures in Python

- In Python ist eine Closure einfach eine von einer anderen Funktion zurückgegebene Funktion (die nicht-lokale Referenzen enthält):

Python-Interpreter

```
>>> def add_x(x):
...     def adder(num):
...         # adder is a closure
...         # x is a free variable
...         return x + num
...     return adder
...
>>> add_5 = add_x(5); add_5
<function adder at ...>
>>> add_5(10)
15
```

Closures in der Praxis

- *Closures* treten immer aus, wenn Funktionen von anderen Funktionen erzeugt werden.
- Manchmal gibt es keine Umgebung, die für die erzeugte Funktion wichtig ist.
- Oft wird eine erzeugte Funktion aber parametrisiert, wie in unserem Beispiel oder bei den parametrisierten Dekoratoren.
- Innehalb von Closures kann auch zusätzlich der Zustand gekapselt werden, wenn auf nonlocal Variablen schreibend zugegriffen wird.
- In den beiden letzteren Fällen wird die Lebenszeit eines Namensraum nicht notwendig bei Verlassen einer Funktion beendet!