Informatik I: Einführung in die Programmierung

15. Funktionale Programmierung

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

29.01.2019

1 Funktionale Programmierung

Programmierparadigmen

- Es gibt verschiedene Programmierparadigmen oder Programmierstile.
- Imperative Programmierung beschreibt, wie etwas erreicht werden soll.
- Deklarative Programmierung beschreibt, was erreicht werden soll.

Imperative Programmierparadigmen

Imperative Programmierung

- Eine Programmausführung besitzt einen Zustand (aktuelle Werte der Variablen, Laufzeitstack, etc).
- Die Anweisungen des Programms modifizieren den Zustand.
- Zentrale Anweisung ist die Zuweisung.

Organisation von imperativen Programmen

- Prozedural: Die Aufgabe wird in kleinere Teile – Prozeduren – zerlegt, die auf den Daten arbeiten. Beispielsprachen: Pascal, C
- Objekt-orientiert: Daten und ihre Methoden bilden eine Einheit, die gemeinsam zerlegt werden. Die Zerlegung wird durch Klassen beschrieben. Beispielsprachen: Java, C++.

Deklarative Programmierparadigmen

Deklarative Programmierung

- Keine explizite Bearbeitung eines Berechnugszustands.
- Logische Programmierung beschreibt das Ziel durch logische Formeln, wie z.B. in Prolog.
- Funktionale Programmierung beschreibt das Ziel durch mathematische Funktionen, wie z.B. in Haskell, ML, Racket
- Abfragesprachen wie SQL oder XQuery sind ebenfalls deklarativ und bauen auf der Relationenalgebra bzw. der XML-Algebra auf.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*). D.h. Funktionen sind auch Daten!
- Es gibt Funktionen höherer Ordnung, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- Keine Schleifen, sondern nur Rekursion.
- Keine Anweisungen, sondern nur Ausdrücke.
- In rein funktionalen Sprachen gibt es keine Zuweisungen und keine Seiteneffekte.
  - $\Rightarrow$ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
  - $\Rightarrow$ referentielle Transparenz: Eine Funktion gibt immer das gleiche Ergebnis bei gleichen Argumenten.
- Die meisten funktionalen Sprachen besitzen ein starkes statisches Typsystem, sodass TypeError zur Laufzeit ausgeschlossen ist.

2 FP in Python

FP in Python

- Funktionen sind „Bürger erster Klasse“.
- Funktionen höherer Ordnung werden voll unterstützt.
- Viele andere Konzepte aus funktionalen Programmiersprachen werden unterstützt, wie die Listen-*Comprehension*.
- In vielen funktionalen Programmiersprachen ist *Lazy Evaluation* ein wichtiger Punkt:
  - Die Auswertung eines Ausdrucks wird nur dann angestoßen, wenn das Ergebnis benötigt wird.
  - Das gleiche gilt für Datenstrukturen, die sich erst entfalten, wenn ihre Inhalte benötigt werden.
- Spezialfall: unendlich Sequenzen mit Generatoren.

FP in Python: Defizite

- Referentielle Transparenz kann in Python verletzt werden.
  Abhilfe: lokale Variablen nur einmal zuweisen, keine globalen Variablen nutzen, keine Mutables ändern.
  Die meisten Beispiele sind “mostly functional” in diesem Sinn.
  Vereinfacht Überlegungen zum aktuellen Zustand der Berechnung.
- Rekursion.
  Python limitiert die Rekursionstiefe, während funktionale Sprachen beliebige Rekursion erlauben und Endrekursion intern automatisch als Schleifen ausführen.
- Ausdrücke.
  Python verlangt Anweisungen in Funktionen, aber viel Funktionalität kann in Ausdrücke verschoben werden.

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
```

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
```

4 Lambda-Notation

Funktionen mit Lambda-Notation definieren

- Der `lambda`-Operator definiert eine kurze, namenlose Funktion, deren Rumpf durch einen Ausdruck gegeben ist.

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
```

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 werden hauptsächlich als Argumente für Funktionen (höherer Ordnung) benutzt.
- Diese Argument-Funktionen werden oft nur einmal verwendet und sind kurz, sodass sich die Vergabe eines Namens nicht lohnt.

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)
```

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

- Funktionen können Funktionen zurückgeben. Auch die Ergebnisfunktion kann durch einen Lambda-Ausdruck definiert werden.
- 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
```

5 Nützliche Funktionen höherer Ordnung: `map`, `reduce` und `filter`

map: Anwendung einer Funktion auf Iteratierbares

- map hat zwei Argumente: eine Funktion und ein iterierbares Objekt.
- map wendet die Funktion auf jedes Element der Eingabe an und liefert die Funktionswerte als Iterator ab.

Python-Interpreter

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

Anwendungsbeispiel für `map`

- Wir wollen eine Liste `c_list` von Temperaturen von Celsius nach Fahrenheit konvertieren. Nach dem Muster zur Verarbeitung von Sequenzen:

`ctof.py`

```
def ctof(temp):
    return ((9 / 5) * temp + 32)
def list_ctof(cl):
    result = []
    for c in cl:
        result + [ctof(c)]
    return result
f_list = list_ctof(c_list)
```

- Ausgabe ist eingeschränkt auf Listen!

Anwendungsbeispiel für `map` (2)

- Als Generator: effizientere Ausgabe, weniger Einschränkungen

```
def gen_ctof (cl):
    for c in cl:
        yield ctof(c)
f_list = list (gen_ctof (c_list))
```

- Mit `map`: Vorteile wie Generator, noch knapper

```
f_list = list(map(lambda c: 1.8 * c + 32, c_list))
```

map mit mehreren Eingaben

- `map` kann auch mit einer k-stelligen Funktion und k weiteren Eingaben aufgerufen werden ($k > 0$).
- Für jeden Funktionsaufruf wird ein Argument von jeder der k Eingaben angefordert. Stop, falls eine der Eingaben keinen Wert mehr liefert.
- Ein Beispiel

Python-Interpreter

```
>>> list(map(lambda x, y, z: x+y+z,
...   range(5), range(0, 50, 10), range(0, 500, 100)))
[0, 111, 222, 333, 400]
```

- Ein einfaches `zip` (das Original funktioniert auch mit > 2 Argumenten):

Python-Interpreter

```
>>> list(map(lambda x, y: (x, y),
...   range(5), range(0, 50, 10)))
[(0, 0), (1, 10), (2, 20), (3, 30), (4, 40)]
```

`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]
```

`partial`: Partielle Anwendung von Funktionen

- `from functools import partial`
- `partial (f, *args, **kwargs)` nimmt eine Funktion *f*, Argumente für *f* und Keywordargumente für *f*
- Ergebnis ist eine neue Funktion, die die verbleibenden Argumente und Keywordargumente für *f* nimmt und dann *f* mit sämtlichen Argumenten aufruft.

Beispiel

- `int` besitzt einen Keywordparameter `base=`, mit dem die Basis der Zahlendarstellung festgelegt wird.
- `int ("10011", base=2)` liefert `19`
- Definiere `int2 = partial (int, base=2)`
- `assert int2 ("10011") == 19`

partial: Beispiel

```
def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print(subsystem, ': ', message)
    ...

server_log = partial(log, subsystem='server')
server_log('Unable to open socket')
```

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

- `from functools import reduce`
- `reduce` wendet eine Funktion $\oplus$ mit zwei Argumenten auf ein iterierbares Objekt und einen Startwert an.
- Der Startwert fungiert wie ein Akkumulator:
  - Bei jedem Iterationsschritt wird der Startwert durch (alter Startwert $\oplus$ nächster Iterationswert) ersetzt.
  - Am Ende ist der Startwert das Ergebnis.
- Falls kein Startwert vorhanden, verwende das erste Element der Iteration.

Python-Interpreter

```
>>> reduce(lambda x, y: x * y, range(1, 5))
24 # ((1 * 2) * 3) * 4
>>> def product(it):
...   return reduce (lambda x,y: x*y, it, 1)
```

Anwendung von `reduce` (1)

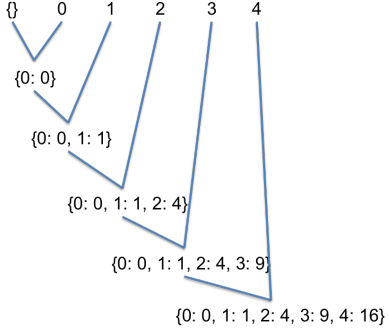
Python-Interpreter

```
>>> 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 ein `dict` aufgebaut, das als Werte die Quadrate seiner Schlüssel enthält.

Anwendung von `reduce` (2)

- Was genau wird da schrittweise reduziert?



Einschub: Das echte, parallele Reduce

- Pythons `reduce` ist in Wirklichkeit ein sogenannter **Fold Operator**.
  `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`
- Das echte `reduce` ist eine Operation im **parallelen Rechnen**.
  - Arbeitet auf einem Array $x_0, \ldots, x_{m-1}$ mit $m = 2^n$ Elementen.
  - Parameter ist assoziative Funktion $\oplus$.
  - Berechnet $r = \bigoplus_{i=0}^{m-1} x_i$.
- Anstatt $r = ((x_0 \oplus x_1) \oplus x_2) \cdots \oplus x_{m-1}$ nacheinander mit $m-1$ $\oplus$ Operationen zu berechnen,
- Beginne mit $x_0, x_2, \ldots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \ldots, (x_{m-2} \oplus x_{m-1})$
- D.h. $m/2$ Operationen parallel in einem Schritt!
- Dann weiter so bis $x_0 \leftarrow (x_0 \oplus x_{m/2-1})$ das Ergebnis liefert.
- Falls m keine Zweierpotenz, werden fehlende Argumente durch die (Rechts-) Einheit von $\oplus$ ersetzt.

6 Listen-, Generator-, `dict`- und Mengen-Comprehensionen

Listen-*Comprehension*

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. deklarativ und kompakt beschrieben werden.
- Entlehnt aus der funktionalen Programmiersprache Haskell.
- Ähnlich der mathematischen Mengenschreibweise: $\{x \in U \mid \phi(x)\}$ (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 pat1 in seq1 if cond1
              for pat2 in seq2 if cond2
              ...
              for patn in seqn if condn ]
```

- Die `if`-Klauseln sind dabei optional.
- Ist `expression` ein Tupel, muss es in Klammern stehen!
- Kurze Schreibweise für Kombinationen aus `map` und `filter`.

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']
```

Zusammenhang Comprehensions vs `map` und `filter`

- Betrachte

```
[ expr for pat in seq if cond ]
```

  mit `pat` ::= `x1,x2,...,xn` für $n > 0$

- Entspricht

```
list (map (lambda pat: expr, filter (lambda pat: cond, seq)))
```

- Falls `cond == True` bzw `if cond` fehlt, kann das Filter weggelassen werden:

```
list (map (lambda pat: expr, seq))
```

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 += [list(range(4))]
...
>>> matrix
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

- Lösung mit Listen-*Comprehensions*:

Python-Interpreter

```
>>> [list (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 += [x+1]
...     matrix += [row]
...
```

- Lösung mit Listen-*Comprehensions*:

Python-Interpreter

```
>>> [list (range (3*y+1, 3*y+4)) 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 += [(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')]
```

Kartesisches Produkt mit `map` und `filter`

- Erster Versuch

```
map (lambda y: map (lambda x: (x,y), range(3)), "abc")
```

- Ergebnis

```
<map object at 0x102dc3438>
```

- ... etwas später

```
[[(0, 'a'), (1, 'a'), (2, 'a')], [(0, 'b'), (1, 'b'), (2, 'b')], [(0, 'c'), (1, 'c
```

- eine Liste von Listen, weil das `map` von `map` einen Iterator von Iteratoren liefert.

Kartesisches Produkt mit `map`, `filter` und `flatten`

- Lösung: `flatten` entfernt eine Ebene von Iteration

```
def flatten (it):
    """flattens a nested iterator to a single iterator"""
    for i in it:
        yield from i
```

- Damit

```
list(flatten(map (lambda y: map (lambda x: (x,y), range(3)), "abc")))
```

- Ergebnis

```
[(0, 'a'), (1, 'a'), (2, 'a'), (0, 'b'), (1, 'b'), (2, 'b'), (0, 'c'), (1, 'c'), (
```

Allgemein: Elimination von Listen-Comprehensions

Basisfall

$$ELC([\texttt{expr}]) = [\texttt{expr}]$$

Elimination von `for`

$$ELC([\texttt{compr}\ \textbf{for}\ \texttt{pat}\ \textbf{in}\ \texttt{seq}\ \textbf{if}\ \texttt{cond}]) =$$
$$\texttt{flatten}(\textbf{map}(\textbf{lambda}\ \texttt{pat} : ELC(\texttt{compr}), \texttt{filter}(\textbf{lambda}\ \texttt{pat} : \texttt{cond}, \texttt{seq})))$$

Beispiel schematisch

```
[(x, y) for x in range(3) for y in "abc"]
```

Elimination von “`for y`” ergibt

```
flatten (map (lambda y: [(x,y) for x in range(3)], "abc"))
```

Elimination von “`for x`” ergibt

```
flatten (map (lambda y: flatten (map (lambda x: [(x,y)], range(3))), "abc"))
```

Generator-*Comprehension*

- Eine Variante der Comprehension baut die Liste nicht explizit auf, sondern liefert einen Iterator, der alle Objekte nacheinander generiert.
- Syntaktischer Unterschied zur Listen-Comprehension: Runde statt eckige Klammern: Generator-Comprehension.
- Die runden Klammern 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(range(0, 20, 2))
>>> evenmultsofthree = set(x for x in evens if x % 3 == 0)
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = set(x for x in text if x >= 'a')
>>>  print(res)
{'a', 'o', 'm', 'n', 'e', 'i', 'g', 'r', 'u', 't', 's'}
>>>  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(d.values())
>>>  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}
```

7 Funktionsschachtelung, Namensräume und Scope

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 Scope 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__`) mit allen vordefinierten Funktionen und Variablen;
  - 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 oft mit der Punkt-Notation zugegriffen werden (insbesondere bei Modulen).

Gültigkeitsbereiche

- Der Scope (oder Gültigkeitsbereich) einer Variablen ist der textuelle Bereich in einem Programm, in dem die Variable ohne die Punkt-Notation referenziert werden kann – d.h. wo sie sichtbar ist.
- Es gibt eine Hierarchie von Gültigkeitsbereichen, wobei der innerste Scope normalerweise alles äußeren überschattet!
- Wird ein Variablennamen referenziert, so versucht Python der Reihe nach:
  - 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 Scope, 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, werden die umgebenden Namensräume von innen nach außen durchsucht.
- 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
```

8 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 Python (2)

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x):
...     return lambda num: x+num
...         # returns a closure
...         # x is a free variable of the lambda
...
>>> add_5 = add_x(5); add_5
<function add_x.<locals>.<lambda> 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 im Beispiel.
- Innerhalb 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!