

Informatik I: Einführung in die Programmierung

16. Funktionale Programmierung

Albert-Ludwigs-Universität Freiburg



Prof. Dr. Peter Thiemann

18.01.2022

1 Funktionale Programmierung



Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Imperative Programmierung

- Eine Programmausführung besitzt einen Zustand (aktuelle Werte der Variablen, Laufzeitkeller, etc).
- Die Anweisungen des Programms modifizieren den Zustand.
- Zentrales Programmelement 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: Smalltalk, Eiffel, Java.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehension

Dekoratoren

Schachtelung
und
Scope

Deklarative Programmierung

- Keine explizite Bearbeitung eines Berechnungszustands.
- **Logische** Programmierung beschreibt das Ziel durch logische Formeln: Prolog, constraint programming, ASP.
- **Funktionale** Programmierung beschreibt das Ziel durch mathematische Funktionen: Haskell, OCaml, Racket, Clojure, Lisp
- Abfragesprachen wie SQL oder XQuery sind ebenfalls deklarativ und bauen auf der Relationenalgebra bzw. der XML-Algebra auf.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Funktionen sind **Bürger erster Klasse** (*first-class citizens*).
- 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.
 - Auch Funktionen als Ausdrücke definierbar.
- In **rein** funktionalen Sprachen: **keine Zuweisungen** und keine Seiteneffekte.
 - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
 - ⇒ **Referentielle Transparenz**: Eine Funktion liefert immer das gleiche Ergebnis bei gleichen Argumenten.
- Die meisten funktionalen Sprachen besitzen ein **starkes statisches Typsystem**, sodass TypeError zur Laufzeit ausgeschlossen ist.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Stark vs. schwach

- In einem starken Typsystem besitzt jeder Wert einen unveränderlichen Typ.
- In einem schwachen Typsystem kann ein Wert je nach Kontext unterschiedliche Typen annehmen.

Statisch vs. dynamisch

- In einem statischen Typsystem wird vor Ausführung eines Programms eine Typüberprüfung durchgeführt. Das Programm kommt nur zur Ausführung, wenn diese Prüfung erfolgreich ist.
- In einem dynamischen Typsystem erfolgt die Typüberprüfung zur Laufzeit, vor Ausführung jeder Operation.
 - Flexibler als statisch, aber meist ineffizienter!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehension

Dekoratoren

Schachtelung
und
Scope

2 FP in Python



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Funktionen werden durch Objekte repräsentiert: „**Bürger erster Klasse**“.
- **Funktionen höherer Ordnung** werden voll unterstützt.
- Python besitzt ein starkes **dynamisches Typsystem**.
- Rein funktionale Programmiersprachen verwenden **Lazy Evaluation**:
 - 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.
- Einige Anwendungen von lazy evaluation können mit Generatoren modelliert werden. Beispiel: unendliche Sequenzen

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- **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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

3 Funktionen definieren und verwenden



Funktionale
Programmierung

FP in Python

**Funktionen
definieren
und
verwenden**

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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

Python-Interpreter

```
>>> def simple():  
...     print('invoked')  
...  
>>> simple # beachte: keine Klammern!  
<function simple at 0x10ccbdc0>  
>>> simple() # Aufruf!  
invoked
```

- Eine Funktion ist ein Python-Objekt.
- Es kann **zugewiesen** werden, als **Argument** übergeben werden und als **Funktionsresultat** zurück gegeben werden.
- Und es ist **aufrufbar** ...

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

4 Lambda-Notation



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

**Lambda-
Notation**

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Der Typ von `mul` kann nicht wie bei einer Definition geschrieben werden. Stattdessen verwende `typing.Callable`:

Python-Interpreter

```
>>> from typing import Callable
>>> mul: Callable[[int, int], int] = lambda x, y: x * y
```

- Der allgemeine Typ einer Funktion ist `Callable[ArgTypes, RetType]` mit
 - *ArgTypes* ist eine Liste von Typen für die Parameter,
 - *RetType* ist der Typ des Rückgabewerts.
- Solch ein Typ wird für Funktionsparameter verwendet, die selbst Funktionen sind.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehension

Dekoratoren

Schachtelung
und
Scope

Python-Interpreter

```
>>> def mul2(x: int, y: int) -> int:  
...     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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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 : int) -> Callable[[int], int]:  
...     return lambda x: x + c  
...  
>>> add5: Callable[[int], int] = gen_adder(5)  
>>> add5(15)  
20
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

5 Nützliche Funktionen höherer Ordnung: map, filter und reduce



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, filter
und reduce**

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

map: Anwendung einer Funktion auf Iterierbares



- `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]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- 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 : float) -> float:
    return ((9 / 5) * temp + 32)
def list_ctof(cl : list[float]) -> list[float]:
    result = []
    for c in cl:
        result += [ctof(c)]
    return result
c_list = [16, 3, -2, -1, 2, 4]
f_list = list_ctof(c_list)
```

- Ausgabe ist eingeschränkt auf Listen!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehension

Dekoratoren

Schachtelung
und
Scope

Anwendungsbeispiel für map (2)

- Als Generator: effizientere Ausgabe, weniger Einschränkungen

```
def gen_ctof (cl: Iterable[float]) -> Iterator[float]:  
    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))
```

- In diesem Fall: besser die benannte Funktion ctof verwenden (bessere Dokumentation, was die Funktion bedeuten soll)

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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 (vgl. `convolute0`)

```
def convolute_0(  
    xs :list[float], ys :list[float]) -> float:  
    return sum(map(lambda x, y: x*y,  
                  xs, reversed(ys)))
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- 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)]
```

- Volle zip-Funktionalität selbst gemacht:

```
def myzip(*args):  
    return map(lambda *args: args, *args)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

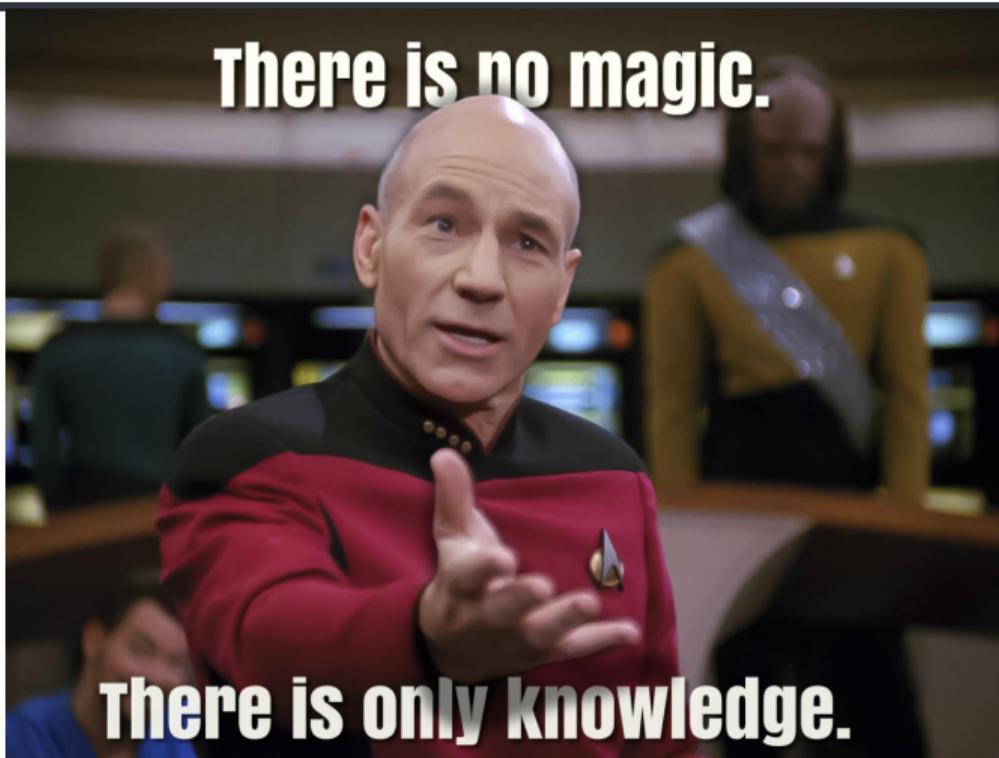
Dekoratoren

Schachte-
lung und
Scope

*arg?



UNI
FREIBURG



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope



- Eine Funktion kann eine variable Zahl von Argumenten akzeptieren.
- Schreibweise dafür

```
def func(a1, a2, a3, *args):  
    for a in args:  
        pass # process arguments 4, 5, ...  
    goo(a1, *args)
```

- func muss mit **mindestens drei** Argumenten aufgerufen werden.
- Weitere Argumente werden als **Tupel** zusammengefasst der Variablen args zugewiesen.
- Der *-Operator kann auch in einer Liste von Ausdrücken auf ein iterierbares Argument angewendet werden.
- Er fügt die Elemente aus dem Iterator der Liste hinzu.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

filter: Filtert unpassende 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]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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: 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`

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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')
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, filter
und reduce**

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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 als **akkumulierender Parameter**:
 - Bei jedem Iterationsschritt wird der Startwert durch (alter Startwert \oplus nächster Iterationswert) ersetzt.
 - Am Ende ist der Startwert das Ergebnis.
- Falls kein Startwert angegeben, 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: Iterable[float]) -> float:
...     return reduce (lambda x,y: x*y, it, 1)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Anwendung von reduce (1)



Python-Interpreter

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, filter
und reduce**

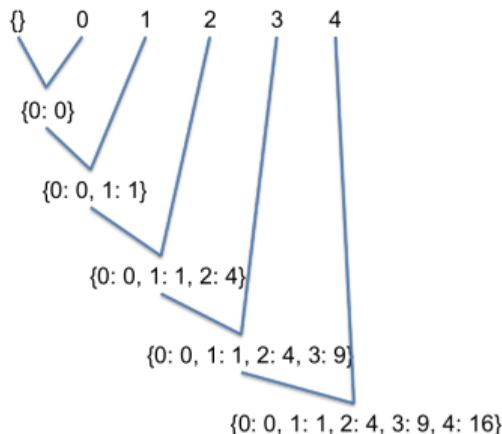
Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Anwendung von reduce (2)

■ Was genau wird da schrittweise **reduziert**?



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, filter
und reduce**

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Einschub: Der echte Reduktionsoperator ist parallel!



- Python's `reduce` ist ein sogenannter **Fold Operator**.
[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))
- Das echte `reduce($\oplus, [x_0, \dots, x_{m-1}]$)` rechnet **parallel** und zwar so:
 - Arbeitet auf einem Array mit $m = 2^n$ Elementen.
 - Parameter ist **assoziative Funktion** \oplus .
 - Berechnet $r = ((x_0 \oplus x_1) \oplus x_2) \cdots \oplus x_{m-1}$.
- Anstatt r mit \oplus -Operationen in $m - 1$ Schritten zu berechnen ...
- Beginne mit $x_0, x_2, \dots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \dots, (x_{m-2} \oplus x_{m-1})$
- 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.

Funktionale Programmierung

FP in Python

Funktionen definieren und verwenden

Lambda-Notation

`map`, `filter` und `reduce`

Comprehension

Dekoratoren

Schachtelung und Scope

6 Listen- und Generator-Comprehensionen



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

**Comprehen-
sion**

Dekoratoren

Schachte-
lung und
Scope

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. **deklarativ** und kompakt beschrieben werden.
- Entlehnt aus der funktionalen Programmiersprache Haskell (Miranda, KRC).
- Inspiriert von 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 eine Liste aus allen `str(x)`, wobei `x` über das iterierbare Objekt `range(10)` läuft und nur die geraden Zahlen berücksichtigt werden.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehension

Dekoratoren

Schachtelung
und
Scope

Generelle Syntax von Listen-Comprehensions



```
[ expr for pat1 in seq1 if cond1  
  for pat2 in seq2 if cond2  
  ...  
  for patn in seqn if condn ]
```

- Die *if*-Klauseln mit den booleschen Ausdrücken *cond1*, ... sind optional.
- Ist *expr* ein Tupel, muss es in Klammern stehen!
- Kurzschreibweise für Kombination 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']
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, *filter*
und *reduce*

Comprehension

Dekoratoren

Schachtelung
und
Scope

Zusammenhang Comprehensions vs map und filter



■ Betrachte

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

mit $pat ::= x_1, x_2, \dots, x_n$ für $n > 0$

■ Entspricht

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

■ Falls if *cond* fehlt, kann das Filter weggelassen werden:

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Geschachtelte Listen-Comprehensions (1)

- Konstruiere die Matrix `[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]`:

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> 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]]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Geschachtelte Listen-Comprehensions (2)



- Konstruiere `[[1,2,3], [4,5,6], [7,8,9]]`:

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> 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]]

```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Erzeuge das kartesische Produkt aus [0, 1, 2] und ['a', 'b', 'c']:

Python-Interpreter

```
>>> prod: list[tuple[int, str]] = []
>>> 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')]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehension

Dekoratoren

Schachtelung
und
Scope

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'), (2, '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

```
X = TypeVar('X')
def flatten (iix : Iterable[Iterable[X]]) -> Iterator[X]:
    """flattens a nested iterable to a single iterator"""
    for ix in iix:
        for x in ix:
            yield x
```

- 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'), (2, 'c')]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren
Schachte-
lung und
Scope

Allgemein: Elimination von Listen-Comprehensions



Elimination des innersten for

$[expr \text{ for } pat \text{ in } seq \text{ if } cond \text{ for } \dots] =$
`flatten(map(lambda pat : [expr for ...], filter(lambda pat : cond, seq)))`

Beispiel schematisch

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

Elimination von "for x" ergibt

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

Elimination von "for y" ergibt

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Eine Variante der Comprehension baut keine Liste 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 als Argument einer Funktion mit nur einem Parameter dient.

Python-Interpreter

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

- Braucht weniger Speicherplatz als `sum([x**2 for x in range(11)])`.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Comprehension für Dictionaries und Mengen



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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>> text = 'Management Training Course'
>>> res = {x for x in text if x >= 'a'}
>>> print(res)
{'a', 'o', 'm', 'n', 'e', 'i', 'g', 'r', 'u', 't', 's'}
>>> d = {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}
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

7 Dekoratoren



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Dekoratoren werden durch Funktionen, die Funktionen als Parameter nehmen und zurückgeben, implementiert.

Dekoratoren, die uns schon früher begegnet sind: `dataclass`, `property`, etc. Es gibt eine spezielle **Syntax**, um solche Dekoratoren anzuwenden. Falls der Dekorator `wrapper` definiert wurde:

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

können wir auch schreiben:

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Dekoratoren: property, staticmethod (1)



decorators.py

```
@dataclass
class C:
    _name : str

    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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Dekoratoren: property, staticmethod (2)



decorators.py

```
@dataclass
class C:
    _name : str

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

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

    @staticmethod
    def hello():
        print("Hello_□world")
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Definition eines Dekorators (1)

Aufgabe

Gib bei jedem Aufruf den Namen der Funktion mit ihren Argumenten aus.

decorators.py

```
verbose = True
def mult(x:float, y:float) -> float:
    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 ...

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Definition eines Dekorators (2)

Wiederverwendbare modulare Lösung



UNI
FREIBURG

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:float, y:float) -> float:
    return x * y
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Definition eines Dekorators (3)

Aufgabe

Wie lange dauert die Ausführung eines Funktionsaufrufs?

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Definition eines Dekorators (4)

Dekoratoren hintereinander schalten



`decorators.py`

```
@decorator
@timeit
def sub(x:float, y:float) -> float:
    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: 2.1457672119140625e-06 sec.
--- a nice footer -----
-2
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Dekoratoren: docstring und `__name__` (1)



- Beim Dekorieren gehen interne Attribute wie Name und docstring verloren.
- Ein guter Dekorator muss das wieder richtigstellen:

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Ein Standardproblem, das selbst durch einen Dekorator gelöst werden kann:

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Dekoratoren mit Parametern (1)



- Aufgabe: beschränke alle Stringergebnisse auf 5 Zeichen

```
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'
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Dekoratoren mit Parametern (2)



- Warum 5 Zeichen? Manchmal sollen es 3 sein, manchmal 6!

```
def limit(length:int):
    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'
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- 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 genau bei der geschachtelten Definition von Funktionen?

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

8 Funktionsschachtelung, Namensräume und Scope



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Im letzten Abschnitt sind uns **geschachtelte Funktionsdefinitionen** begegnet.
- Dabei müssen wir klären, auf welche Bindung sich die Verwendung einer Variablen bezieht.
- Dafür 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
 - **Built-in**-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
 - Namensraum von **Modulen**, die importiert werden
 - **globaler** Namensraum (des Moduls `__main__`)
 - **lokaler** Namensraum innerhalb einer Funktion
 - Namensräume haben verschiedene **Lebensdauern**; der **lokale Namensraum** einer Funktion existiert z.B. 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).

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehension

Dekoratoren

Schachtelung
und
Scope

- 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 alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
 - ihn im **lokalen** Scope aufzulösen;
 - ihn im **nicht-lokalen** Scope aufzulösen;
 - ihn im **globalen** Scope aufzulösen;
 - ihn im **Builtin**-Namensraum aufzulösen.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope



- Gibt es eine **Zuweisung** `var = ...` im aktuellen Scope, so wird von einem lokalen Namen ausgegangen und Referenzen auf `var` dürfen erst nach Ausführung einer Zuweisung erfolgen.
- Ausnahmen:
 - „`global var`“ bedeutet, dass `var` in der **globalen** Umgebung gesucht werden soll. Auch Zuweisungen wirken auf die globale Umgebung.
 - „`nonlocal var`“ bedeutet, dass `var` in einer **nicht-lokalen** Umgebung gesucht werden soll, d.h. in den umgebenden Funktionsdefinitionen. Auch Zuweisungen wirken dort.
- Gibt es keine Zuweisungen, werden die umgebenden Namensräume von innen nach außen durchsucht.
- Kann ein Namen nicht aufgelöst werden, dann gibt es eine Fehlermeldung.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Ein Beispiel für Namensräume und Gültigkeitsbereiche (1)



```
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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

9 Closures



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Eine **Closure** ist eine von einer anderen Funktion zurückgegebene lokale Funktion, die freie Variable (nicht-lokale Referenzen) enthält:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:  
...     def adder(num:float) ->float:  
...         # 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

Closures in Python (2)



- Dasselbe mit einer lambda Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:  
...     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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

```
>>> def clo() -> Callable[[], int]:  
...     x = 0  
...     f = lambda : x  
...     x = x + 1  
...     return f  
...  
>>> fx = clo()  
>>> fx(0)  
1           # nicht 0!
```

- Nachfolgende Zuweisungen ändern den Wert in der Closure...

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope

- Definition: Eine Variable tritt **frei** in einem Funktionsrumpf auf, wenn sie zwar vorkommt, aber weder in der Parameterliste noch in einer lokalen Zuweisung gesetzt wird.
- Jede Funktion mit freien Variablen wird durch eine *Closure* repräsentiert.
- Innerhalb einer Closure kann mit Hilfe der Anweisungen `nonlocal` oder `global` auf freie Variable schreibend zugegriffen werden.
- In den beiden letzteren Fällen wird die **Lebensdauer** eines Namensraums (nämlich der umschließenden Funktion) nicht notwendig bei Verlassen der Closure beendet!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Comprehen-
sion

Dekoratoren

Schachte-
lung und
Scope