

Informatik I: Einführung in die Programmierung

17. Funktionale Programmierung

Albert-Ludwigs-Universität Freiburg



UNI
FREIBURG

Prof. Dr. Peter Thiemann
21.01.2026

1 Funktionale Programmierung



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Programmierparadigmen



UNI
FREIBURG

- 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

Komprehen-
sionen

Imperative Programmierung

- Zum Programm gehört ein Zustand (aktuelle Werte der Variablen, Laufzeitkeller, etc), der sich während der Ausführung ändert.
- Denkansatz beim Programmieren: Modifikation des Zustands.
- Programm besteht aus **Anweisungen** (Zuweisung, ...).

Organisation von imperativen Programmen

- **Prozedural:** Die Aufgabe wird in kleinere Teile – Prozeduren – zerlegt, die auf den Daten arbeiten. (Sprachen: Pascal, C, ...)
- **Objekt-orientiert:** Die Aufgabe wird in Klassen zerlegt, die lokal Daten und die Methoden darauf enthalten. (Sprachen: Smalltalk, Eiffel, Java, ...)

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Deklarative Programmierung

- Keine explizite Bearbeitung eines Berechnungszustands.
- **Logische** Programmierung (LP) beschreibt die Aufgabe durch logische Formeln: Prolog, constraint programming, ASP.
- **Funktionale** Programmierung (FP) beschreibt die Aufgabe 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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Eigenschaften funktionaler Programmierung



- 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 sind als Ausdrücke definierbar.
- In **rein funktionalen Sprachen**: **keine Zuweisungen** und **keine Seiteneffekte**.
 - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
 - ⇒ Alle Datenstrukturen sind unveränderlich.
 - ⇒ **Referentielle Transparenz**: Eine Funktion liefert bei gleichen Argumenten immer das gleiche Ergebnis.
- Die meisten funktionalen Sprachen besitzen ein **starkes statisches Typsystem**, sodass zur Laufzeit kein `TypeError` auftreten kann.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

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 statische Typüberprüfung, aber meist weniger effizient!

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

2 Funktionale Programmierung in Python



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

- Funktionen werden durch Objekte repräsentiert.
- Funktionen höherer Ordnung werden voll unterstützt.
- Python besitzt ein starkes dynamisches Typsystem.

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 automatisch in Schleifen umwandeln.
- Ausdrücke.
Python erlaubt bei `lambda`-Funktionen nur einen Ausdruck statt eines Blocks von Anweisungen.

3 Funktionen definieren und verwenden



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Funktionsdefinition und -verwendung



- Eine Funktion ist ein Python-Objekt.

```
>>> def simple() -> None:  
...     print('invoked')  
...  
>>> simple    # keine Klammern -> Funktionsobjekt  
<function simple at 0x10df3ec00>  
>>> simple() # mit Klammern -> Funktionsaufruf  
invoked
```

- Es kann **zugewiesen** werden, als **Argument** übergeben werden und als **Funktionsresultat** zurückgegeben werden.
- Und es ist **aufrufbar** vom Typ Callable...

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Funktionsverwendung



```
>>> from typing import Callable
>>> spam = simple; print(spam)
<function simple at 0x10df3ec00>
>>> def call_twice(fun : Callable[[],None]) -> None:
...     fun(); fun()
...
>>> call_twice(spam) # keine Klammern hinter spam
invoked
invoked
>>> def gen_fun() -> Callable[[], None]:
...     return spam
...
>>> gen_fun()
<function simple at 0x10df3ec00>
>>> gen_fun()()
invoked
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

4 Lambda-Notation



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Funktionen mit Lambda-Notation definieren



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

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at 0x10df3ea20>
>>> (lambda x, y: x * y)(3, 8)
24
>>> mul = lambda x, y: x * y
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Der Typ Callable



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

```
>>> 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 die Liste der Typen der Parameter,
 - *RetType* ist der Typ des Rückgabewerts.
- Wird auch für Funktionsparameter verwendet, die selbst Funktionen sind.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Verwendung von Lambda-Funktionen (1)



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

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

- `mul2` ist äquivalent zu `mul`!
- Lambda-Funktionen werden hauptsächlich als Argumente für Funktionen (höherer Ordnung) benutzt.
- Solche Funktionen werden oft nur einmal verwendet und sind kurz, sodass sich die Vergabe eines Namens nicht lohnt.

Verwendung von Lambda-Funktionen (2): Funktionsfabriken



- Funktionen können Funktionen zurückgeben. Auch das Ergebnis einer Funktion kann durch einen Lambda-Ausdruck definiert werden.
- Beispiel: Eine Funktion, die einen Addierer erzeugt, der immer eine vorgegebene Konstante addiert:

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

20

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

5 Nützliche Funktionen höherer Ordnung: map, filter und reduce



Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, filter
und reduce**

Komprehen-
sionen

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.

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

- Eine getypte Definition für `map`:

```
def map[A, B](f : Callable[[A], B]
              , xs : Iterable[A]) -> Iterator[B] :
    for x in xs:
        yield f(x)
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

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 : 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)
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Anwendungsbeispiel für map (2)



- Mit `map` wesentlich knapper:

```
f_list = list(map(ctof, c_list))
```

- Oder mit einer `lambda` Funktion:

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

map mit mehreren Eingaben



- Die eingebaute `map`-Funktion kann auch mit einer k -stelligen Funktion und k weiteren iterierbaren 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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Der wirkliche Typ von map

- Der Typ der eingebauten `map` Funktion kann mit den bisherigen Typannotationen nicht hingeschrieben werden.
- Wir brauchen eine unbekannte Anzahl von Typvariablen, die sich nach der Zahl der Argumente richtet.

```
def map[*As, B](f : Callable[[*As], B], *xs : Iterable[*As]
                 ) -> Iterator[B] :
```

 . . .
- Dabei ist `*As` eine **Tupeltypvariable**, die für ein Tupel von Typen steht. Sie kann nur zusammen mit Tupelunpacking (wie im Beispiel) verwendet werden.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `filter`
und `reduce`

Komprehen-
sionen

Zip aus map



- Ein einfaches `zip` mit `map` programmiert:

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

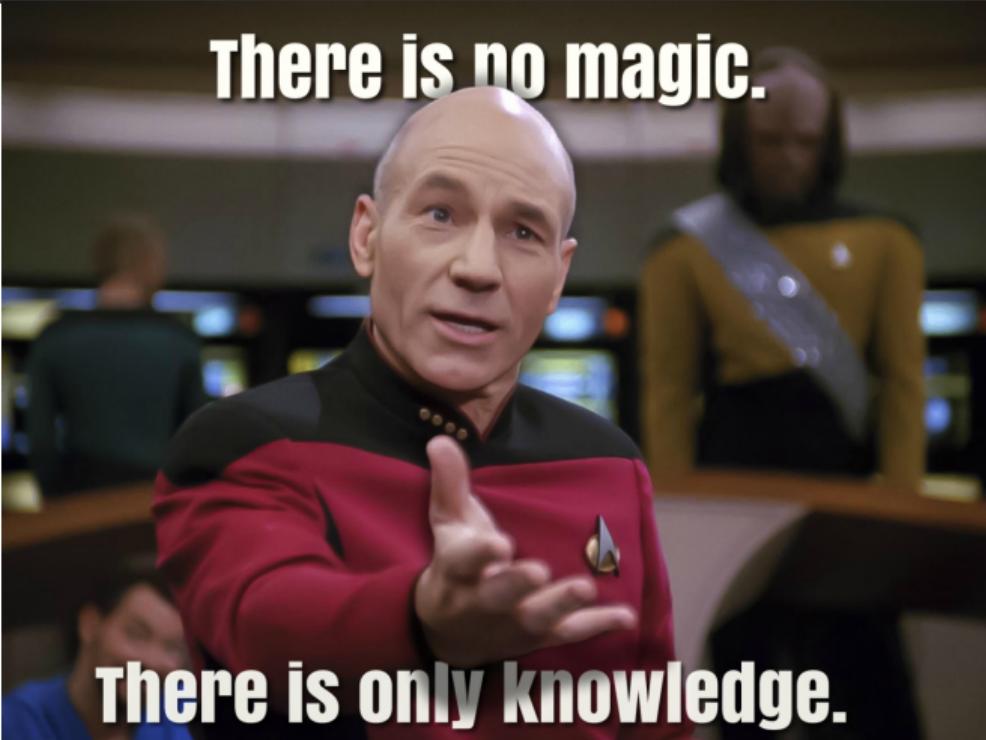
- Das originale `zip` funktioniert auch mit > 2 Argumenten...
- Volle Funktionalität von `zip` selbst gemacht:

```
def myzip[*As](*args : Iterable[*As]) -> Iterator[tuple[*As]]:  
    return map(lambda *args: args, *args)
```

*arg?



UNI
FREIBURG



Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map, filter
und reduce`

Komprehen-
sionen

Wiederholung: Variable Parameterlisten



- 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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

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.

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

- Eine Definition dazu

```
def filter[A](p : Callable[[A], bool], xs: Iterable[A]) -> Iterator[A]:  
    for x in xs:  
        if p(x):  
            yield x
```

partial: Partielle Anwendung von Funktionen



UNI
FREIBURG

- `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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

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 Akkumulator ersetzt durch (alter Akkumulator \oplus nächster Iterationswert).
 - Der finale Wert des Akkumulators ist das Ergebnis.
- Falls kein Startwert angegeben wird, verwende das erste Element der Iteration.

```
>>> from typing import Iterable
>>> reduce(lambda x, y: x * y, range(1, 5))
24
>>> def product(it: Iterable[float]) -> float:
...     return reduce(lambda x,y: x*y, it, 1)
...
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Anwendung von reduce (1)



UNI
FREIBURG

```
>>> 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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

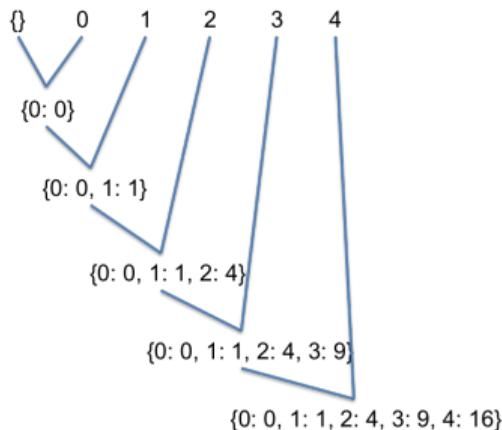
Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Anwendung von reduce (2)

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



Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, filter
und reduce**

Komprehen-
sionen

Einschub: Der echte Reduktionsoperator ist parallel!



- Pythons `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 $r = ((x_0 \oplus x_1) \oplus x_2) \dots \oplus x_{m-1}$.

- Anstatt r mit \oplus -Operationen in $m - 1$ Schritten zu berechnen ...

Einschub: Der echte Reduktionsoperator ist parallel (2)



- Berechne $r = ((x_0 \oplus x_1) \oplus x_2) \cdots \oplus x_{m-1}$ ($m - 1$ Operationen \oplus).
- 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: $x_0, x_4, \dots, x_{m-4} \leftarrow (x_0 \oplus x_2), (x_4 \oplus x_6), \dots, (x_{m-4} \oplus x_{m-2})$.
- Jetzt $m/4$ Operationen parallel in einem Schritt!
- Dann weiter so bis zum Ergebnis $x_0 \leftarrow (x_0 \oplus x_{m/2})$.
- ... in $n = \log_2 m$ Schritten
- Falls m keine Zweierpotenz, werden fehlende Argumente durch die (Rechts-) Einheit von \oplus ersetzt.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

6 Komprehensionen



Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Listen-Komprehension



- *Komprehensionen* können Listen **deklarativ** und kompakt beschreiben.

- Inspiriert von der mathematischen Mengenschreibweise:

$\{x \in U \mid \phi(x)\}$ (alle x aus U , die die Bedingung ϕ erfüllen).

Beispiel:

```
>>> [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.
- Kurzschreibweise für Kombination aus `map` und `filter`.

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

Allgemeine Syntax von Listen-Komprehensionen



UNI
FREIBURG

```
[ expr  for tup1  in seq1  if cond1
     for tup2  in seq2  if cond2
     ...
     for tupn  in seqn  if condn ]
```

- Jedes tup_i ist ein Tupel (vgl. Tupel-Unpacking).
- Jedes seq_i ist ein iterierbares Objekt.
- Die if-Klauseln mit den booleschen Ausdrücken $cond_1, \dots$ sind optional.
- Ist $expr$ ein Tupel, muss es in Klammern stehen!

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Zusammenhang Komprehensionen vs map und filter



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

■ Betrachte

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

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

■ Entspricht

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

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

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

Geschachtelte Listen-Komprehensionen (1)

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

```
>>> 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-Komprehensionen:

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

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Geschachtelte Listen-Komprehensionen (2)

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

```
>>> 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-Komprehensionen:

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

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Listen-Komprehensionen: Kartesisches Produkt



UNI
FREIBURG

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

```
>>> prod: list[tuple[int, str]] = []
>>> for x in range(3):
...     for y in ['a', 'b', 'c']:
...         prod += [(x, y)]
...
...
```

- Lösung mit Listen-Komprehensionen:

```
>>> [(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,
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehe-
nionen

Kartesisches Produkt mit map und filter



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation
map, filter
und reduce

Komprehen-
sionen

■ Erster Versuch

```
>>> map (lambda y: map (lambda x: (x,y), range(3)), "abc")  
<map object at 0x10df5af80>
```

■ ... etwas später

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

```
print(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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Allgemein: Elimination von Listen-Komprehensionen

Wiederhole die Elimination des innersten for

[expr for tup in seq if cond for ...] =

`flatten(map(lambda tup : [expr for ...], filter(lambda tup : 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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Generator-Komprehension



- Eine **Generator-Komprehension** baut keine Liste auf, sondern liefert einen **Iterator**, der die spezifizierten Objekte nacheinander generiert.
- Syntaktischer Unterschied zur Listen-Komprehension:
Runde statt eckige Klammern.
- Die runden Klammern können weggelassen werden, wenn der Ausdruck als Argument einer Funktion mit nur einem Parameter dient.
Beispiel:

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

385
- Braucht weniger Speichplatz als `sum([x**2 for x in range(11)])!`

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen

Komprehension für Dictionaries und Mengen



UNI
FREIBURG

Auch Mengen und Dictionaries können durch Komprehension-Ausdrücke definiert werden. Nachfolgend ein paar Beispiele:

```
>>> evens = set(range(0, 20, 2))
>>> {x for x in evens if x % 3 == 0}
{0, 18, 12, 6}
```

```
>>> text = 'Management Training Course'
>>> {x for x in text if x >= 'a'}
{'e', 'i', 'o', 'u', 'm', 't', 'g', 'n', 's', 'a', 'r'}
```

```
>>> { x: x**2 for x in range(1, 10)}
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, filter
und reduce

Komprehen-
sionen