Informatik I: Einführung in die Programmierung

19. Komprehensionen für Listen und Generatoren

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Prof. Dr. Peter Thiemann

01.02.2023

Komprehensionen

Listen-Komprehension

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. deklarativ und kompakt beschrieben werden.

Listen-Komprehension

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. deklarativ und kompakt beschrieben werden.
- Entlehnt aus der funktionalen Programmiersprache Haskell (Miranda, KRC).

Listen-Komprehension

- 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:

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

Listen-Komprehension

- 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:

```
>>> [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.

Generelle Syntax von Listen-Komprehensionen


```
[ 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 $cond1, \ldots$ sind optional.

Generelle Syntax von Listen-Komprehensionen


```
[ 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 $cond1, \ldots$ sind optional.
- Ist *expr* ein Tupel, muss es in Klammern stehen!

Generelle Syntax von Listen-Komprehensionen


```
[ 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 $cond1, \ldots$ sind optional.
- Ist *expr* ein Tupel, muss es in Klammern stehen!
- Kurzschreibweise für Kombination aus `map` und `filter`.

```
>>> [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 Komprehensionen vs `map` und `filter`

- Betrachte

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

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

Zusammenhang Komprehensionen 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)))
```

Zusammenhang Komprehensionen 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 `if` *cond* fehlt, kann das Filter weggelassen werden:

```
list (map (lambda pat: 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]]
```

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]]
```

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]
...
```

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]]
```

Listen-Komprehensionen: Kartesisches Produkt

- 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)]
...
```

Listen-Komprehensionen: Kartesisches Produkt

- 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
```

Kartesisches Produkt mit `map` und `filter`

- Erster Versuch

```
>>> map (lambda y: map (lambda x: (x,y), range(3)), "abc")
<map object at 0x10ac2a560>
```

Kartesisches Produkt mit `map` und `filter`

- Erster Versuch

```
>>> map (lambda y: map (lambda x: (x,y), range(3)), "abc")
<map object at 0x10ac2a0e0>
```

- ... etwas später

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

Kartesisches Produkt mit `map` und `filter`

- Erster Versuch

```
>>> map (lambda y: map (lambda x: (x,y), range(3)), "abc")
<map object at 0x10ac2a1a0>
```

- ... 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

```
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
```

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")))
```

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'
```

Allgemein: Elimination von Listen-Komprehensionen

Elimination des innersten `for`

[*expr* **for** *pat* **in** *seq* **if** *cond* **for** ...] =

`flatten`(`map`(**lambda** *pat* : [*expr* **for** ...], `filter`(**lambda** *pat* : *cond*, *seq*)))

Allgemein: Elimination von Listen-Komprehensionen

Elimination des innersten `for`

[*expr* **for** *pat* **in** *seq* **if** *cond* **for** ...] =

`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)))
```

Generator-Komprehension

- Eine Variante der Komprehension baut keine Liste auf, sondern liefert einen Iterator, der alle Objekte nacheinander generiert.

Generator-Komprehension

- Eine Variante der Komprehension baut keine Liste auf, sondern liefert einen Iterator, der alle Objekte nacheinander generiert.
- Syntaktischer Unterschied zur Listen-Komprehension: Runde statt eckige Klammern: Generator-Komprehension.

Generator-Komprehension

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

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

Generator-Komprehension

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

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

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

Komprehension für Dictionaries und Mengen

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

```
>>> 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)
{'o', 'g', 'u', 's', 'm', 'n', 'e', 'a', 't', 'r', 'i'}
>>> 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}
```

Komprehension für Dictionaries, Mengen, etc. (2)


```
>>> sqnums = set(d.values())
>>> print(sqnums)
{64, 1, 4, 36, 9, 16, 49, 81, 25}
>>> 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),
>>> 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}
```

Komprehension für Dictionaries, Mengen, etc. (3)

`all` und `any` quantifizieren über die Elemente eines iterierbaren Objekts:

- `all(iterable)` liefert `True` gdw. alle Elemente äquivalent zu `True` sind (oder das `iterable` leer ist).

Komprehension für Dictionaries, Mengen, etc. (3)

`all` und `any` quantifizieren über die Elemente eines iterierbaren Objekts:

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

```
>>> text = 'Management Training Course'
>>> 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
```