Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

FS24 CAS PML - Python

Niklaus Johner

niklausbernhard.johner@bfh.ch

FS24 CAS PML - Python

11. Mehr über Listen

Daten filtern und transformieren

- Daten bearbeiten:
  - Filtern: nur Daten die eine gewisse Kondition erfüllen werden behalten
  - Transformieren: eine Funktion auf jedes Element anwenden

```
data = [None, 2, 0, 3]
filtered = []
for el in data:
    if el is not None:
        filtered.append(el)

squared = []
for el in filtered:
    squared.append(el * el)
```

Daten filtern und transformieren

- Daten bearbeiten:
  - Filtern: nur Daten die eine gewisse Kondition erfüllen werden behalten
  - Transformieren: eine Funktion auf jedes Element anwenden

```
filtered = filter(lambda x: x is not None, data)
squared = map(lambda x: x * x, filtered)

squared = map(lambda x: x * x,
              filter(lambda x: x is not None, data))
```

map, *filter* und *lambda* Funktion

Daten filtern und transformieren

- Daten bearbeiten:
  - Filtern: nur Daten die eine gewisse Kondition erfüllen werden behalten
  - Transformieren: eine Funktion auf jedes Element anwenden

```
filtered = [el for el in data if el is not None]
squared = [el*el for el in filtered]

squared = [el*el for el in data if el is not None]
```

list comprehension

Die *map* Funktion

- *map(function, iteratable)*
  - wendet die Funktion *function* auf jedes Element von *iteratable* an.
  - In python3 gibt sie ein *generator* zurück

```
In [9]: gen = map(abs, [-1, 2, -3])
   ...: for el in gen:
   ...:     print(el)
   ...:
1
2
3
```

Die *map* Funktion

- *map(function, iteratable)*
  - wendet die Funktion *function* auf jedes Element von *iteratable* an.
  - In python3 gibt sie ein *generator* zurück

```
In [11]: list(map(abs, [-1, 2, -3]))
Out[11]: [1, 2, 3]

In [12]: list(map(min, [(1, 2), (8, 5)]))
Out[12]: [1, 5]

In [13]: list(map(sorted, [(1, 5, 3), (8, 5, 2)]))
Out[13]: [[1, 3, 5], [2, 5, 8]]
```

Die *filter* Funktion

- *filter(function, iteratable)*
  - behält alle Elemente *el* für welche *bool(function(el)) == True*.
  - In python3 gibt sie ein *generator* zurück

```
In [20]: list(filter(min, [(1, 0), (8, -1)]))
Out[20]: [(8, -1)]
```

```
In [21]: def is_positive(x):
    ...:     return x > 0
    ...:

In [22]: list(filter(is_positive, [-1, 0, 1, 2]))
Out[22]: [1, 2]
```

lambda Funktionen

- Für jedes Filtering eine Funktion definieren macht wenig Sinn
- Die elegante Lösung ist eine *lambda* Funktion
  - Anonyme Funktion
  - Kann direkt in einer *map* oder *filter* Anweisung definiert werden

```
lambda x, y: x * y
```

lambda Funktionen

- Man kann eine lambda Funktion einer Variablen zuweisen

```
prod = lambda x, y: x * y
```

```
def prod(x,y):
    return x*y
```

- Oder anonym in einem Funktionsaufruf definieren

```
In [24]: data = [None, 2, 0, 3]

In [25]: not_none = lambda el: el is not None

In [26]: list(filter(not_none, data))
Out[26]: [2, 0, 3]

In [27]: list(filter(lambda el: el is not None, data))
Out[27]: [2, 0, 3]
```

list-comprehension

- Eine *comprehension* ist eine elegante Syntax um Listen, Dictionaries und Sets zu generieren

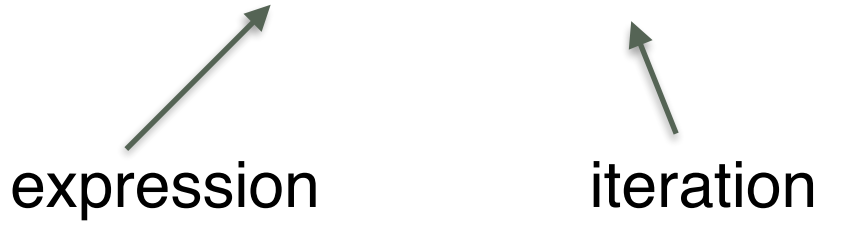
```
l1 = [0, 1, 2, 3]
```

Loop

```
l = []
for el in l1:
    l.append(el * el)
```

Comprehension

```
l = [el * el for el in l1]
```



list-comprehension

- comprehension mit Konditionen

```
l1 = [0, 1, 2, 3]
```

Loop

```
l = []
for el in l1:
    if el != 1:
        l.append(el * el)
```

Comprehension

```
l = [el * el for el in l1
     if el != 1]
```

Zusätzliche Folien

list-comprehension

- Ternary operator

```
a = 0
1 / a if a != 0 else "nan"
```

- comprehension mit ternary operator

```
l1 = [0, 1, 2, 3]
```

Loop

```
l = []
for el in l1:
    if el != 0:
        l.append(1 / el)
    else:
        l.append(None)
```

Comprehension

```
l = [1 / el if el != 0
     else None for el in l1]
```

list-comprehension

- Nested comprehensions

```
l1 = [0, 1, 2, 3] ; l2 = [0, 1]
```

Loop

```
l = []
for el in l1:
    l.append([])
    for i in range(el):
        l[-1].append(i)
```

Comprehension

```
l = [[i for i in range(el)]
     for el in l1]
```

```
l = []
for el1 in l1:
    for el2 in l2:
        l.append((el1, el2))
```

```
l = [(el1, el2) for el1 in
     l1 for el2 in l2]
```

dict-comprehension

- comprehensions gibt es auch für dictionnaries

```
pairs = [("k1", 1), ("k2", 2)]
```

Loop

```
d = {}
for k, el in pairs:
    d[k] = el
```

Comprehension

```
d = {k: el for k, el in pairs}
```

```
d = {}
for k, el in pairs:
    if el < 2:
        d[k] = el
```

```
d = {k: el for k, el in
     pairs if el < 2}
```

set-comprehension

- und für sets

```
l = [1, 1, 2, 3, 2]
```

Loop

```
s = set()
for el in l:
    s.add(el)
```

```
s = set()
for el in l:
    if el != 3:
        s.add(el)
```

Comprehension

```
s = {el for el in l}
```

```
s = {el for el in l
     if el != 3}
```