

Informatik I: Einführung in die Programmierung

15. Funktionale Programmierung

Albert-Ludwigs-Universität Freiburg



UNI
FREIBURG

Peter Thiemann

29.01.2020



Funktionale Programmierung

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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++.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Deklarative Programmierung

- **Logische** Programmierung beschreibt die Aufgabe durch logische Formeln, wie z.B. in Prolog.
- **Funktionale** Programmierung beschreibt die Aufgabe 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.
- Keine explizite Bearbeitung eines Berechnungszustands.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind **Bürger erster Klasse** (*first-class citizens*). D.h. Funktionen sind auch Daten!

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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**.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.
 - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.
 - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
 - ⇒ **referentielle Transparenz**: Eine Funktion gibt immer das gleiche Ergebnis bei gleichen Argumenten.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.
 - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
 - ⇒ **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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



FP in Python

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „Bürger erster Klasse“.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „Bürger erster Klasse“.
- Funktionen höherer Ordnung werden voll unterstützt.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „Bürger erster Klasse“.
- Funktionen höherer Ordnung werden voll unterstützt.
- Viele andere Konzepte aus funktionalen Programmiersprachen werden unterstützt, wie die *Listen-Comprehension*.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „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:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „**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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Funktionen sind Objekte also „**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: unendliche Sequenzen mit Generatoren.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- **Referentielle Transparenz** kann in Python verletzt werden.
Abhilfe: lokale Variablen nur einmal zuweisen, keine globalen Variablen nutzen, keine Mutables ändern, die als Parameter übergeben werden.
Die meisten Beispiele sind “mostly functional” in diesem Sinn.
Vereinfacht Überlegungen zum aktuellen Zustand der Berechnung.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- **Referentielle Transparenz** kann in Python verletzt werden.
Abhilfe: lokale Variablen nur einmal zuweisen, keine globalen Variablen nutzen, keine Mutables ändern, die als Parameter übergeben werden.
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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachtelung und
Scope

Closures



- **Referentielle Transparenz** kann in Python verletzt werden.
Abhilfe: lokale Variablen nur einmal zuweisen, keine globalen Variablen nutzen, keine Mutables ändern, die als Parameter übergeben werden.
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, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Funktionen definieren und verwenden

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> def simple():  
...     print('invoked')  
...
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> def simple():  
...     print('invoked')  
...
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> def simple():  
...     print('invoked')  
...  
>>> simple # beachte: keine Klammern!
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> def simple():  
...     print('invoked')  
...  
>>> simple # beachte: keine Klammern!  
<function simple at 0x10ccbdcb0>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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!
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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).

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachtelung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 **aufzurufbar** ...

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachtelung
und
Scope

Closures



Python-Interpreter

```
>>> spam = simple; print(spam)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Python-Interpreter

```
>>> spam = simple; print(spam)
<function simple at 0x10ccbdcb0>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Python-Interpreter

```
>>> spam = simple; print(spam)
<function simple at 0x10ccbdcb0>
>>> def call_twice(fun):
...     fun(); fun()
... 
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Python-Interpreter

```
>>> spam = simple; print(spam)
<function simple at 0x10ccbdcb0>
>>> def call_twice(fun):
...     fun(); fun()
...
>>> call_twice(spam) # Keine Klammern hinter spam
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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
...
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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()
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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()()
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Lambda-Notation

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

**Lambda-
Notation**

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at 0x107cf4950>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at 0x107cf4950>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Python-Interpreter

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at 0x107cf4950>
>>> (lambda x, y: x * y)(3, 8)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Der `lambda`-Operator definiert eine **kurze, namenlose** Funktion, deren Rumpf durch einen einzigen 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Der `lambda`-Operator definiert eine **kurze, namenlose** Funktion, deren Rumpf durch einen einzigen 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`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehension

Schachtelung
und
Scope

Closures

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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehension

Schachtelung und
Scope

Closures

Verwendung von Lambda-Funktionen (2)



UNI
FREIBURG

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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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



UNI
FREIBURG

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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



UNI
FREIBURG

- 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:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

map: Anwendung einer Funktion auf Iteratierbares



UNI
FREIBURG

- `map` hat zwei Argumente: eine Funktion und ein iterierbares Objekt.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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.

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]
```



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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Als Generator: effizientere Ausgabe, weniger Einschränkungen

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Als Generator: effizientere Ausgabe, weniger Einschränkungen

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

- Mit map: Vorteile wie Generator, noch knapper

```
f_mapped = map(ctof, c_list)  
  
f_lambda = map(lambda c: 1.8 * c + 32, c_list)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

map mit mehreren Eingaben



UNI
FREIBURG

- `map` kann auch mit einer k -stelligen Funktion und k weiteren Eingaben aufgerufen werden ($k > 0$).

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

map mit mehreren Eingaben



UNI
FREIBURG

- `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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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

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]
```

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):

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



UNI
FREIBURG

- `filter` erwartet als Argumente eine Funktion mit einem Parameter und ein iterierbares Objekt.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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.

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]
```



- `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`

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehension

Schachtelung und
Scope

Closures

partial: Beispiel



UNI
FREIBURG

```
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, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Bedeutung von *args und **kwargs



- Wir hatten bereits die Schreibweise **kwargs kennengelernt, mit dem beliebige Schlüsselwortparameter zugelassen werden.
- *args in der Parameterliste erlaubt den Aufruf einer Funktion mit beliebig vielen unbenannten (positionalen) Parametern
- Die Argumente dazu werden in args als **Tupel** übergeben

```
def parameters(*args):  
    print("you called me with", len(args), "parameters")  
    print("I received a", type(args))
```

Python-Interpreter

```
>>> parameters([1,2,3,4])  
you called me with 4 parameters  
I received a <class 'tuple'>
```

Einschub: Bedeutung von *args und **kwargs



- **kwargs als letztes Element der Parameterliste erlaubt den Aufruf einer Funktion mit beliebig vielen Schlüsselwortparametern
- Kann auch gemeinsam mit *args verwendet werden.
- Die Argumente dazu werden in kwargs als **Dictionary** übergeben

```
def kwparm(**kwargs):  
    print("I got", len(kwargs), "keyword parameters")  
    print("with keys", list(kwargs.keys()))  
    print("and values", list(kwargs.values()))
```

Python-Interpreter

```
>>> kwparm(a=1, b=2, d=4)  
I got 3 keyword parameters  
with keys ['a', 'b', 'd'] and values [1, 2, 4]
```

Einschub: Aufruf mit `*args` und `**kwargs`



UNI
FREIBURG

- `*args` und `**kwargs` können auch (in dieser Reihenfolge am Ende) in Argumentlisten beim Aufruf von Funktionen verwendet werden.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Aufruf mit *args und **kwargs



- *args und **kwargs können auch (in dieser Reihenfolge am Ende) in Argumentlisten beim Aufruf von Funktionen verwendet werden.
- Beispiel: Rufe eine Funktion mit beliebig vielen Argumenten auf

```
def apply(f, *args):  
    return f(*args)
```

Einschub: Aufruf mit *args und **kwargs



- *args und **kwargs können auch (in dieser Reihenfolge am Ende) in Argumentlisten beim Aufruf von Funktionen verwendet werden.
- Beispiel: Rufe eine Funktion mit beliebig vielen Argumenten auf

```
def apply(f, *args):  
    return f(*args)
```

- Verwendung

```
apply(lambda x, y: x - 2 * y, 17, 4)  
# returns 9
```

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



UNI
FREIBURG

```
■ from functools import reduce
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `from functools import reduce`
- `reduce` wendet eine Funktion \oplus mit zwei Argumenten auf ein iterierbares Objekt und einen Startwert an.

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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 Akkumulator durch (alter Akkumulator \oplus nächster Iterationswert) ersetzt.

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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 Akkumulator durch (alter Akkumulator \oplus nächster Iterationswert) ersetzt.
 - Am Ende ist der Akkumulator das Ergebnis.

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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 Akkumulator durch (alter Akkumulator \oplus nächster Iterationswert) ersetzt.
 - Am Ende ist der Akkumulator das Ergebnis.
- Falls kein Startwert vorhanden, verwende das erste Element der Iteration.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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 Akkumulator durch (alter Akkumulator \oplus nächster Iterationswert) ersetzt.
 - Am Ende ist der Akkumulator 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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 Akkumulator durch (alter Akkumulator \oplus nächster Iterationswert) ersetzt.
 - Am Ende ist der Akkumulator 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

reduce: Reduktion eines iterierbaren Objekts auf einen Wert



- `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 Akkumulator durch (alter Akkumulator \oplus nächster Iterationswert) ersetzt.
 - Am Ende ist der Akkumulator 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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Anwendung von reduce (1)



UNI
FREIBURG

Python-Interpreter

```
>>> def to_dict(d, key):  
...     d[key] = key**2  
...     return d  
...  
>>> reduce(to_dict, list(range(5)), {})
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehension

Schachtelung und
Scope

Closures

Anwendung von reduce (2)



UNI
FREIBURG

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

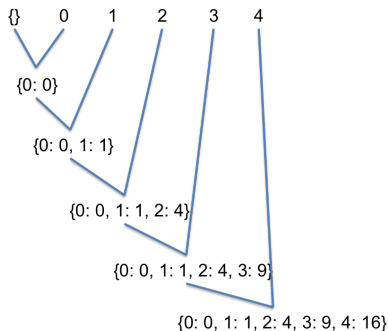
Schachte-
lung und
Scope

Closures

Anwendung von reduce (2)



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



Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Pythons `reduce` ist in Wirklichkeit ein sogenannter **Fold Operator**.
[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



UNI
FREIBURG

- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

**map, reduce
und filter**

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



UNI
FREIBURG

- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} mit $m = 2^n$ Elementen.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



UNI
FREIBURG

- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} mit $m = 2^n$ Elementen.
 - Parameter ist **assoziative Funktion** \oplus .

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} mit $m = 2^n$ Elementen.
 - Parameter ist **assoziative Funktion** \oplus .
 - Berechnet $r = ((x_0 \oplus x_1) \oplus x_2) \cdots \oplus x_{m-1}$.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} 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 sequentiell $m - 1 \oplus$ Operationen in $m - 1$ Schritten zu berechnen,

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} 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 sequentiell $m - 1 \oplus$ Operationen in $m - 1$ Schritten zu berechnen,
- Beginne mit $x_0, x_2, \dots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \dots, (x_{m-2} \oplus x_{m-1})$

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} 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 sequentiell $m - 1 \oplus$ 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. berechne $m/2 \oplus$ Operationen **parallel in einem Schritt!**

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Einschub: Das echte, parallele Reduce



- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} 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 sequentiell $m - 1 \oplus$ 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. berechne $m/2 \oplus$ Operationen **parallel in einem Schritt!**
- Weiter so bis $x_0 \leftarrow (x_0 \oplus x_{m/2-1})$ nach $\lceil \log_2 m \rceil$ Schritten das Ergebnis liefert.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehension

Schachtelung
und
Scope

Closures

Einschub: Das echte, parallele Reduce



- Pythons `reduce` ist in Wirklichkeit 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` ist eine Operation im **parallelen Rechnen**.
 - Arbeitet auf einem Array x_0, \dots, x_{m-1} 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 sequentiell $m - 1 \oplus$ 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. berechne $m/2 \oplus$ Operationen **parallel in einem Schritt!**
- Weiter so bis $x_0 \leftarrow (x_0 \oplus x_{m/2-1})$ nach $\lceil \log_2 m \rceil$ Schritten 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`, `reduce`
und `filter`

Comprehension

Schachtelung
und
Scope

Closures



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

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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']
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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!

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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`.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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']
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Zusammenhang Comprehensions vs map und filter



UNI
FREIBURG

■ Betrachte

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

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Zusammenhang Comprehensions vs map und filter



■ Betrachte

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

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

■ Entspricht

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Zusammenhang Comprehensions vs map und filter



■ Betrachte

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

mit $\text{pat} ::= x_1, x_2, \dots, x_n$ 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)



UNI
FREIBURG

- Wir wollen eine zweidimensionale Matrix der Art
[[0,1,2,3], [0,1,2,3], [0,1,2,3]] konstruieren.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Geschachtelte Listen-*Comprehensions* (1)



UNI
FREIBURG

- Wir wollen eine zweidimensionale Matrix der Art $[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]$ konstruieren.
- Imperative Lösung:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

**Comprehen-
sion**

Schachte-
lung und
Scope

Closures

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]]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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*:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)



UNI
FREIBURG

- Wir wollen `[[1,2,3], [4,5,6], [7,8,9]]` konstruieren.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

**Comprehen-
sion**

Schachte-
lung und
Scope

Closures

Geschachtelte Listen-*Comprehensions* (2)



UNI
FREIBURG

- Wir wollen $[[1,2,3], [4,5,6], [7,8,9]]$ konstruieren.
- Imperativ:

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

**Comprehen-
sion**

Schachte-
lung und
Scope

Closures

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]
... 
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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*:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehension

Schachtelung und
Scope

Closures

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]]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Listen-*Comprehensions*: Kartesisches Produkt



UNI
FREIBURG

- Wir wollen das kartesische Produkt aus $[0, 1, 2]$ und $['a', 'b', 'c']$ erzeugen.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Listen-*Comprehensions*: Kartesisches Produkt



UNI
FREIBURG

- Wir wollen das kartesische Produkt aus $[0, 1, 2]$ und $['a', 'b', 'c']$ erzeugen.
- Imperativ:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)]
... 
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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*:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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



UNI
FREIBURG

■ Erster Versuch

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Kartesisches Produkt mit map und filter



■ Erster Versuch

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

■ Ergebnis

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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') ]]
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachte-
lung und
Scope

Closures

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



UNI
FREIBURG

■ 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Kartesisches Produkt mit map, filter und flatten



UNI
FREIBURG

- 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")))
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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


Allgemein: Elimination von Listen-Comprehensions

Basisfall

$$ELC([expr]) = [expr]$$



UNI
FREIBURG

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Allgemein: Elimination von Listen-Comprehensions

Basisfall

$$ELC([expr]) = [expr]$$

Elimination von for

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

Allgemein: Elimination von Listen-Comprehensions

Basisfall

$$ELC([expr]) = [expr]$$

Elimination von for

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

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"))
```



- Eine Variante der Comprehension baut die Liste nicht explizit auf, sondern liefert einen **Iterator**, der alle Objekte nacheinander generiert.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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**.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (1)



UNI
FREIBURG

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (1)



UNI
FREIBURG

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}
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (1)



UNI
FREIBURG

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'
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (1)



UNI
FREIBURG

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')
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (1)



UNI
FREIBURG

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'}
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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))
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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}
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (2)



UNI
FREIBURG

Python-Interpreter

```
>>> sqnums = set(d.values())  
>>> print(sqnums)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (2)



UNI
FREIBURG

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))
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (2)



UNI
FREIBURG

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Comprehension für Dictionaries, Mengen, etc. (2)



UNI
FREIBURG

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}
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Funktionsschachtelung, Namensräume und Scope

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Im letzten Abschnitt sind uns **geschachtelte** Funktionsdefinitionen begegnet.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Im letzten Abschnitt sind uns **geschachtelte** Funktionsdefinitionen begegnet.
- Es ist dabei nicht immer klar, auf was sich ein bestimmter **Variablenname** bezieht.

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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** verstehen.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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** 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, reduce
und filter

Comprehension

Schachtelung und
Scope

Closures



- Ein Namensraum ist eine **Abbildung** von Namen auf Werte (innerhalb von Python oft durch ein `dict` realisiert).

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Ein Namensraum ist eine **Abbildung** von Namen auf Werte (innerhalb von Python oft durch ein `dict` realisiert).
- Innerhalb von Python gibt es:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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;

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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;

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 Namensraum von **Modulen**, die importiert werden;
 - den **globalen** Namensraum (des Moduls `__main__`);

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 Namensraum von **Modulen**, die importiert werden;
 - den **globalen** Namensraum (des Moduls `__main__`);
 - den **lokalen** Namensraum innerhalb einer Funktion;

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 Namensraum 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 Namensraum 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 Namensraum 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).

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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;

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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;

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachtelung und
Scope

Closures



- 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;

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachtelung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehension

Schachtelung
und
Scope

Closures



- Gibt es eine **Zuweisung** im aktuellen Scope, so wird von einem lokalen Namen ausgegangen (außer es gibt andere Deklarationen):

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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.

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 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.



- 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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)



UNI
FREIBURG

Python-Interpreter

```
>>> scope_test()
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)



UNI
FREIBURG

Python-Interpreter

```
>>> scope_test()
```

```
After local assignment: test spam
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)



UNI
FREIBURG

Python-Interpreter

```
>>> scope_test()
```

```
After local assignment: test spam
```

```
After nonlocal assignment: nonlocal spam
```

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)



UNI
FREIBURG

Python-Interpreter

```
>>> scope_test()
```

```
After local assignment: test spam
```

```
After nonlocal assignment: nonlocal spam
```

```
After global assignment: nonlocal spam
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)



UNI
FREIBURG

Python-Interpreter

```
>>> scope_test()  
After local assignment: test spam  
After nonlocal assignment: nonlocal spam  
After global assignment: nonlocal spam  
>>> print("In global scope:", spam)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)



UNI
FREIBURG

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
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



Closures

Funktionale
Programmie-
rung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

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

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- In Python ist eine Closure einfach eine von einer anderen Funktion zurückgegebene Funktion (die nicht-lokale Referenzen enthält):

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- In Python ist eine Closure einfach eine von einer anderen Funktion zurückgegebene Funktion (die nicht-lokale Referenzen enthält):

Python-Interpreter

```
>>> def add_x(x):
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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):
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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  
...
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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 ...>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- 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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- Dasselbe mit einer `lambda` Abstraktion:

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Closures in Python (2)



UNI
FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x):
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

Closures in Python (2)



UNI
FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x):  
...     return lambda num: x+num
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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  
... 
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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 ...>
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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)
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures

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
```

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

`map`, `reduce`
und `filter`

Comprehen-
sion

Schachte-
lung und
Scope

Closures



- *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!

Funktionale
Programmierung

FP in Python

Funktionen
definieren
und
verwenden

Lambda-
Notation

map, reduce
und filter

Comprehen-
sion

Schachte-
lung und
Scope

Closures