

# Informatik I: Einführung in die Programmierung

## 16. Funktionale Programmierung

Albert-Ludwigs-Universität Freiburg



UNI  
FREIBURG

Prof. Dr. Peter Thiemann

10.01.2023



# Funktionale Programmierung

Funktionale  
Programmie-  
rung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

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`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Imperative Programmierung

- Eine Programmausführung besitzt einen Zustand (aktuelle Werte der Variablen, Laufzeitkeller, etc).
- Die Anweisungen des Programms modifizieren den Zustand.
- Zentrales Programmelement ist die Zuweisung.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Imperative Programmierung

- Eine Programmausführung besitzt einen Zustand (aktuelle Werte der Variablen, Laufzeitkeller, etc).
- Die Anweisungen des Programms modifizieren den Zustand.
- Zentrales Programmelement ist die Zuweisung.

## Organisation von imperativen Programmen

- **Prozedural**: Die Aufgabe wird in kleinere Teile – Prozeduren – zerlegt, die auf den Daten arbeiten. Beispielsprachen: Pascal, C
- **Objekt-orientiert**: Daten und ihre Methoden bilden eine Einheit, die gemeinsam zerlegt werden. Die Zerlegung wird durch Klassen beschrieben. Beispielsprachen: Smalltalk, Eiffel, Java.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Deklarative Programmierung

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.

Funktionale  
Programmie-  
rung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- 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`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- 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`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- **Keine Schleifen**, sondern nur Rekursion.
- **Keine Anweisungen**, sondern nur Ausdrücke.
  - Auch Funktionen sind als Ausdrücke definierbar.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- **Keine Schleifen**, sondern nur Rekursion.
- **Keine Anweisungen**, sondern nur Ausdrücke.
  - Auch Funktionen sind als Ausdrücke definierbar.
- In **rein** funktionalen Sprachen: **keine Zuweisungen** und keine Seiteneffekte.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- **Keine Schleifen**, sondern nur Rekursion.
- **Keine Anweisungen**, sondern nur Ausdrücke.
  - Auch Funktionen sind als Ausdrücke definierbar.
- In **rein** funktionalen Sprachen: **keine Zuweisungen** und keine Seiteneffekte.
  - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- **Keine Schleifen**, sondern nur Rekursion.
- **Keine Anweisungen**, sondern nur Ausdrücke.
  - Auch Funktionen sind als Ausdrücke definierbar.
- In **rein** funktionalen Sprachen: **keine Zuweisungen** und keine Seiteneffekte.
  - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
  - ⇒ Alle Datenstrukturen sind unveränderlich.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- **Keine Schleifen**, sondern nur Rekursion.
- **Keine Anweisungen**, sondern nur Ausdrücke.
  - Auch Funktionen sind als Ausdrücke definierbar.
- In **rein** funktionalen Sprachen: **keine Zuweisungen** und keine Seiteneffekte.
  - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
  - ⇒ Alle Datenstrukturen sind unveränderlich.
  - ⇒ **Referentielle Transparenz**: Eine Funktion liefert immer das gleiche Ergebnis bei gleichen Argumenten.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Es gibt **Funktionen höherer Ordnung**, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- **Keine Schleifen**, sondern nur Rekursion.
- **Keine Anweisungen**, sondern nur Ausdrücke.
  - Auch Funktionen sind als Ausdrücke definierbar.
- In **rein** funktionalen Sprachen: **keine Zuweisungen** und keine Seiteneffekte.
  - ⇒ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
  - ⇒ Alle Datenstrukturen sind unveränderlich.
  - ⇒ **Referentielle Transparenz**: Eine Funktion liefert immer das gleiche Ergebnis bei gleichen Argumenten.
- Die meisten funktionalen Sprachen besitzen ein **starkes statisches Typsystem**, sodass zur Laufzeit kein `TypeError` auftreten kann.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Stark vs. schwach

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

## Statisch vs. dynamisch

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# FP in Python

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Funktionen werden durch Objekte repräsentiert.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Funktionen werden durch Objekte repräsentiert.
- Funktionen höherer Ordnung werden voll unterstützt.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Funktionen werden durch Objekte repräsentiert.
- Funktionen höherer Ordnung werden voll unterstützt.
- Python besitzt ein starkes dynamisches Typsystem.
- Rein funktionale Programmiersprachen verwenden *Lazy Evaluation*:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Funktionen werden durch Objekte repräsentiert.
- Funktionen höherer Ordnung werden voll unterstützt.
- Python besitzt ein starkes dynamisches Typsystem.
- Rein funktionale Programmiersprachen verwenden *Lazy Evaluation*:
  - Die Auswertung eines Ausdrucks wird nur dann angestoßen, wenn das Ergebnis benötigt wird.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Funktionen werden durch Objekte repräsentiert.
- Funktionen höherer Ordnung werden voll unterstützt.
- Python besitzt ein starkes dynamisches Typsystem.
- Rein funktionale Programmiersprachen verwenden *Lazy Evaluation*:
  - Die Auswertung eines Ausdrucks wird nur dann angestoßen, wenn das Ergebnis benötigt wird.
  - Das gleiche gilt für Datenstrukturen, die sich erst entfalten, wenn ihre Inhalte benötigt werden.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Funktionen werden durch Objekte repräsentiert.
- Funktionen höherer Ordnung werden voll unterstützt.
- Python besitzt ein starkes dynamisches Typsystem.
- Rein funktionale Programmiersprachen verwenden *Lazy Evaluation*:
  - Die Auswertung eines Ausdrucks wird nur dann angestoßen, wenn das Ergebnis benötigt wird.
  - Das gleiche gilt für Datenstrukturen, die sich erst entfalten, wenn ihre Inhalte benötigt werden.
- Python verwendet *Eager Evaluation*, d.h., jeder Ausdruck (insbesondere Argumente von Funktionen) wird ausgewertet; nur der Wert des Ausdrucks wird weiter verwendet.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

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 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`, `filter`  
und `reduce`

Dekoratoren

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 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, filter  
und reduce

Dekoratoren

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 meisten Beispiele sind “mostly functional” in diesem Sinn.**  
Vereinfacht Überlegungen zum aktuellen Zustand der Berechnung.
- **Rekursion.**  
Python limitiert die Rekursionstiefe, während funktionale Sprachen beliebige Rekursion erlauben und Endrekursion intern automatisch als Schleifen ausführen.
- **Ausdrücke.**  
Python verlangt Anweisungen in Funktionen, aber viel Funktionalität kann in Ausdrücke verschoben werden.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# Funktionen definieren und verwenden

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Funktion ist ein Python-Objekt.

```
>>> def simple() -> None:
...     print('invoked')
...
>>> simple    # keine Klammern -> Funktionsobjekt
<function simple at 0x103999000>
>>> simple() # mit Klammern -> Funktionsaufruf
invoked
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Funktion ist ein Python-Objekt.

```
>>> def simple() -> None:
...     print('invoked')
...
>>> simple    # keine Klammern -> Funktionsobjekt
<function simple at 0x103999090>
>>> simple() # mit Klammern -> Funktionsaufruf
invoked
```

- Es kann **zugewiesen** werden, als **Argument** übergeben werden und als **Funktionsresultat** zurückgegeben werden.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Funktion ist ein Python-Objekt.

```
>>> def simple() -> None:
...     print('invoked')
...
>>> simple    # keine Klammern -> Funktionsobjekt
<function simple at 0x103999000>
>>> simple() # mit Klammern -> Funktionsaufruf
invoked
```

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



```
>>> from typing import Callable
>>> spam = simple; print(spam)
<function simple at 0x103999000>
>>> def call_twice(fun : Callable[[],None]) -> None:
...     fun(); fun()
...
>>> call_twice(spam) # keine Klammern hinter spam
invoked
invoked
>>> def gen_fun() -> Callable[[], None]:
...     return spam
...
>>> gen_fun()
<function simple at 0x103999000>
>>> gen_fun()()
invoked
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# Lambda-Notation

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

**Lambda-  
Notation**

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at 0x103999510>
>>> (lambda x, y: x * y)(3, 8)
24
>>> mul = lambda x, y: x * y
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

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

- Der allgemeine Typ einer Funktion ist `Callable[ArgTypes, RetType]` mit

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

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

- Der allgemeine Typ einer Funktion ist `Callable[ArgTypes, RetType]` mit
  - *ArgTypes* ist eine Liste von Typen für die Parameter,

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

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

- Der allgemeine Typ einer Funktion ist `Callable[ArgTypes, RetType]` mit
  - *ArgTypes* ist eine Liste von Typen für die Parameter,
  - *RetType* ist der Typ des Rückgabewerts.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Verwendung von Lambda-Funktionen (1)



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

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Verwendung von Lambda-Funktionen (1)



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

- mul2 ist äquivalent zu mul!
- Lambda-Funktionen werden hauptsächlich als Argumente für Funktionen (höherer Ordnung) benutzt.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Verwendung von Lambda-Funktionen (1)



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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Verwendung von Lambda-Funktionen (2)



## Beispiel cookie\_lib.py

```
# add cookies in order of most specific  
# (i.e., longest) path first  
cookies.sort(key=lambda arg: len(arg.path), reverse=True)
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

**Lambda-  
Notation**

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- 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`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# Nützliche Funktionen höherer Ordnung: map, filter und reduce

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

**map, filter  
und reduce**

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# map: Anwendung einer Funktion auf Iterierbares



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# map: Anwendung einer Funktion auf Iterierbares



- `map` hat zwei Argumente: eine Funktion und ein iterierbares Objekt.
- `map` wendet die Funktion auf jedes Element der Eingabe an und liefert die Funktionswerte als Iterator ab.

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Mit map wesentlich knapper:

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

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



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- `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`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- `map` kann auch mit einer  $k$ -stelligen Funktion und  $k$  weiteren Eingaben aufgerufen werden ( $k > 0$ ).
- Für jeden Funktionsaufruf wird ein Argument von jeder der  $k$  Eingaben angefordert. Stop, falls eine der Eingaben keinen Wert mehr liefert.
- Ein Beispiel (vgl. `convolute0`)

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Ein einfaches zip (das Original funktioniert auch mit  $> 2$  Argumenten):

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Ein einfaches zip (das Original funktioniert auch mit  $> 2$  Argumenten):

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

- Volle Funktionalität von zip selbst gemacht:

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

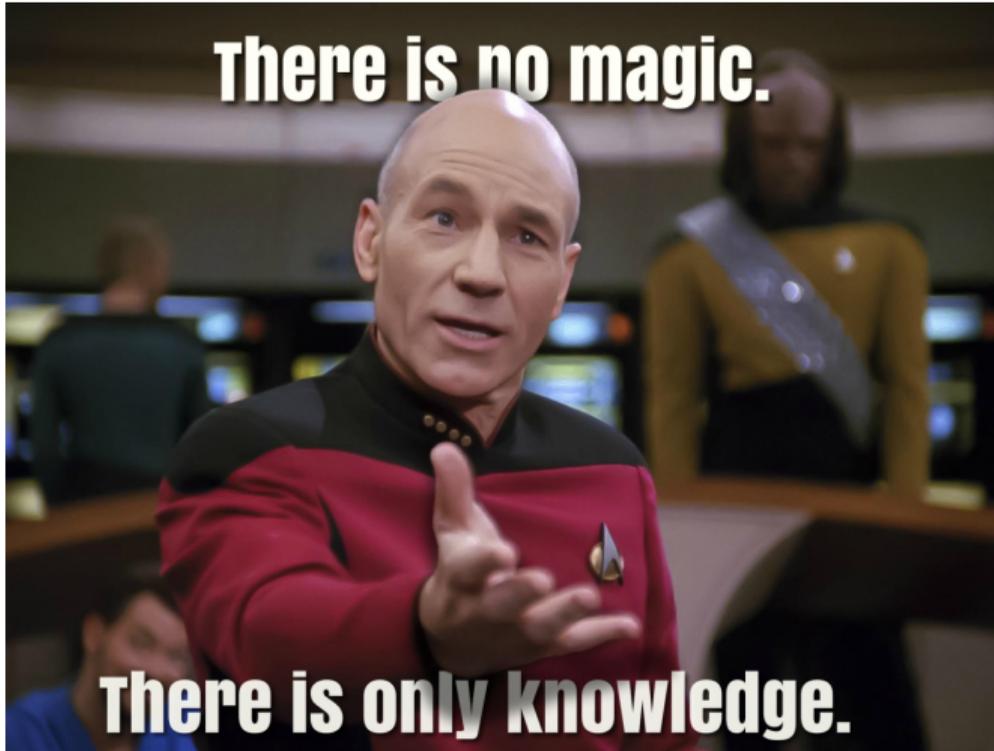
Schachte-  
lung und  
Scope

Closures

\*arg?



UNI  
FREIBURG



Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Funktion kann eine variable Zahl von Argumenten akzeptieren.

- Schreibweise dafür

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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# filter: Filtert unpassende Objekte aus



- `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`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

## filter: Filtert unpassende Objekte aus



- `filter` erwartet als Argumente eine Funktion mit einem Parameter und ein iterierbares Objekt.
- Es liefert einen Iterator zurück, der die Objekte aufzählt, bei denen die Funktion nicht `False` (oder äquivalente Werte) zurück gibt.

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- `from functools import partial`
- `partial (f, *args, **kwargs)` nimmt eine Funktion `f`, Argumente für `f` und Keywordargumente für `f`
- Ergebnis: Funktion, die die verbleibenden Argumente und Keywordargumente für `f` nimmt und dann `f` mit sämtlichen Argumenten aufruft.

## Beispiel

- `int` besitzt einen Keywordparameter `base=`, mit dem die Basis der Zahlendarstellung festgelegt wird.
- `int ("10011", base=2)` liefert 19
- Definiere `int2 = partial (int, base=2)`
- `assert int2 ("10011") == 19`

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



```
def log(message, subsystem):  
    """Write 'message' to the specified subsystem."""  
    print(subsystem, ': ', message)  
    ...  
  
server_log = partial(log, subsystem='server')  
server_log('Unable to open socket')
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# reduce: Reduktion eines iterierbaren Objekts auf ein Element



```
>>> from functools import reduce
```

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# reduce: Reduktion eines iterierbaren Objekts auf ein Element



```
>>> from functools import reduce
```

- `reduce` wendet eine Funktion  $\oplus$  mit zwei Argumenten auf ein iterierbares Objekt und einen Startwert an.
- Der Startwert fungiert als **akkumulierender Parameter**:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# reduce: Reduktion eines iterierbaren Objekts auf ein Element



```
>>> from functools import reduce
```

- `reduce` wendet eine Funktion  $\oplus$  mit zwei Argumenten auf ein iterierbares Objekt und einen Startwert an.
- Der Startwert fungiert als **akkumulierender Parameter**:
  - Bei jedem Iterationsschritt wird der Startwert durch (alter Startwert  $\oplus$  nächster Iterationswert) ersetzt.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# reduce: Reduktion eines iterierbaren Objekts auf ein Element



```
>>> from functools import reduce
```

- `reduce` wendet eine Funktion  $\oplus$  mit zwei Argumenten auf ein iterierbares Objekt und einen Startwert an.
- Der Startwert fungiert als **akkumulierender Parameter**:
  - Bei jedem Iterationsschritt wird der Startwert durch (alter Startwert  $\oplus$  nächster Iterationswert) ersetzt.
  - Am Ende ist der Startwert das Ergebnis.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# reduce: Reduktion eines iterierbaren Objekts auf ein Element



```
>>> from functools import reduce
```

- reduce wendet eine Funktion  $\oplus$  mit zwei Argumenten auf ein iterierbares Objekt und einen Startwert an.
- Der Startwert fungiert als **akkumulierender Parameter**:
  - Bei jedem Iterationsschritt wird der Startwert durch (alter Startwert  $\oplus$  nächster Iterationswert) ersetzt.
  - Am Ende ist der Startwert das Ergebnis.
- Falls kein Startwert angegeben, verwende das erste Element der Iteration.

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Anwendung von reduce (1)



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

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, filter  
und reduce**

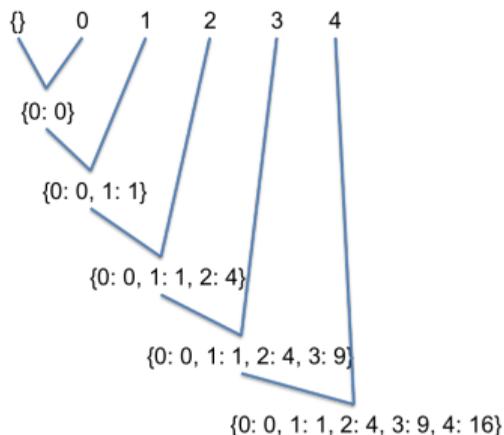
Dekoratoren

Schachte-  
lung und  
Scope

Closures



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



Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

**map, filter  
und reduce**

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



- Python's `reduce` ist ein sogenannter **Fold Operator**.  
[https://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



- Python's `reduce` ist ein sogenannter **Fold Operator**.  
[https://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))
- Das echte `reduce( $\oplus, [x_0, \dots, x_{m-1}]$ )` rechnet **parallel** und zwar so:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



- Python's `reduce` ist ein sogenannter **Fold Operator**.  
[https://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))
- Das echte `reduce( $\oplus, [x_0, \dots, x_{m-1}]$ )` rechnet **parallel** und zwar so:
  - Arbeitet auf einem Array mit  $m = 2^n$  Elementen.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



- Python's `reduce` ist ein sogenannter **Fold Operator**.  
[https://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))
- Das echte `reduce( $\oplus, [x_0, \dots, x_{m-1}]$ )` rechnet **parallel** und zwar so:
  - Arbeitet auf einem Array mit  $m = 2^n$  Elementen.
  - Parameter ist **assoziative Funktion**  $\oplus$ .

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Einschub: Der echte Reduktionsoperator ist parallel!



- Python's `reduce` ist ein sogenannter **Fold Operator**.  
[https://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))
- Das echte `reduce( $\oplus$ ,  $[x_0, \dots, x_{m-1}]$ )` rechnet **parallel** und zwar so:
  - Arbeitet auf einem Array mit  $m = 2^n$  Elementen.
  - Parameter ist **assoziative Funktion**  $\oplus$ .
  - Berechnet  $r = ((x_0 \oplus x_1) \oplus x_2) \cdots \oplus x_{m-1}$ .
- Anstatt  $r$  mit  $\oplus$ -Operationen in  $m - 1$  Schritten zu berechnen ...
- Beginne mit  $x_0, x_2, \dots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \dots, (x_{m-2} \oplus x_{m-1})$
- D.h.  $m/2$  Operationen parallel in einem Schritt!
- Dann weiter so bis  $x_0 \leftarrow (x_0 \oplus x_{m/2-1})$  das Ergebnis liefert.
- Falls  $m$  keine Zweierpotenz, werden fehlende Argumente durch die (Rechts-) Einheit von  $\oplus$  ersetzt.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# Dekoratoren

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures

# Was ist ein Dekorator?



Ein **Dekorator** ist eine Funktion, die eine andere Funktion erweitert, ohne diese selbst zu ändern.

Syntax von Dekoratoren (Funktion `decorator` angewendet auf `fun`):

```
@decorator
def fun():
    ...
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures



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

Dekoratoren, die uns schon früher begegnet sind: `dataclass`, `property`, etc.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures



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

Dekoratoren, die uns schon früher begegnet sind: `dataclass`, `property`, etc.  
Falls der Dekorator `wrapper` definiert wurde, dann hat

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

die gleiche Bedeutung wie

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Dekoratoren: property, staticmethod (1)

decorators.py

```
@dataclass
class C:
    _name : str

    def getname(self):
        return self._name

    # def setname(self, x):
    #     self._name = 2 * x
    name = property(getname)

    def hello():
        print("Hello world")
    hello = staticmethod(hello)
```

lässt sich mittels der @-Syntax schreiben ...



UNI  
FREIBURG

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Dekoratoren: property, staticmethod (2)



```
@dataclass
class C:
    _name : str

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

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

    @staticmethod
    def hello():
        print("Hello world")
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Definition eines Dekorators (1)



## Aufgabe

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures



## Aufgabe

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

```
verbose = True
def mult(x:float, y:float) -> float:
    if verbose:
        print("--- a nice header -----")
        print("--> call mult with args: %s, %s" % x, y)
    res = x * y
    if verbose:
        print("--- a nice footer -----")
    return res
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Aufgabe

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

```
verbose = True
def mult(x:float, y:float) -> float:
    if verbose:
        print("--- a nice header -----")
        print("--> call mult with args: %s, %s" % x, y)
    res = x * y
    if verbose:
        print("--- a nice footer -----")
    return res
```

Das ist hässlich! Wir wollen eine generische Lösung ...

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Definition eines Dekorators (2)

Wiederverwendbare modulare Lösung



```
def decorator(f):
    def wrapper(*args, **kwargs):
        print("--- a nice header -----")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer -----")
        return res
    # print("--> wrapper now defined")
    return wrapper
```

```
@decorator
def mult(x:float, y:float) -> float:
    return x * y
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Definition eines Dekorators (3)



## Aufgabe

Wie lange dauert die Ausführung eines Funktionsaufrufs?

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures

# Definition eines Dekorators (3)



## Aufgabe

Wie lange dauert die Ausführung eines Funktionsaufrufs?

```
import time

def timeit(f):
    def wrapper(*args, **kwargs):
        print("--> Start timer")
        t0 = time.time()
        res = f(*args, **kwargs)
        delta = time.time() - t0
        print("--> End timer:  %s sec." % delta)
        return res
    return wrapper
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Definition eines Dekorators (4)

Dekoratoren hintereinander schalten



```
decorators.py
```

```
@decorator
@timeit
def sub(x:float, y:float) -> float:
    return x - y

print(sub(3, 5))
```

liefert z.B.:

```
decorators.py
```

```
--- a nice header -----
--> call wrapper with args: 3,5
--> Start timer
--> End timer:  2.1457672119140625e-06 sec.
--- a nice footer -----
-2
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Dekoratoren: `docstring` und `__name__` (1)



- Beim Dekorieren gehen interne Attribute wie `Name` und `docstring` verloren.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures

# Dekoratoren: docstring und `__name__` (1)



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

```
def decorator(f):  
    def wrapper(*args, **kwargs):  
        print("--- a nice header -----")  
        print("--> call %s with args: %s" %  
              (f.__name__, ",".join(map(str, args))))  
        res = f(*args, **kwargs)  
        print("--- a nice footer -----")  
        return res  
    wrapper.__name__ = f.__name__  
    wrapper.__doc__ = f.__doc__  
    return wrapper
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Dieses Problem kann durch den Dekorator `functools.wraps` gelöst werden:

```
import functools
def decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print("--- a nice header -----")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer -----")
        return res
    return wrapper
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Aufgabe: beschränke alle Stringergebnisse auf 5 Zeichen

```
>>> def trunc(f):  
...     def wrapper(*args, **kwargs):  
...         res = f(*args, **kwargs)  
...         return res[:5]  
...     return wrapper  
...  
>>> @trunc  
... def data():  
...     return 'foobar'  
...  
...
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

**Dekoratoren**

Schachte-  
lung und  
Scope

Closures



- Aufgabe: beschränke alle Stringergebnisse auf 5 Zeichen

```
>>> def trunc(f):  
...     def wrapper(*args, **kwargs):  
...         res = f(*args, **kwargs)  
...         return res[:5]  
...     return wrapper  
...  
>>> @trunc  
... def data():  
...     return 'foobar'  
...  
...
```

- Ein aktueller Aufruf:

```
>>> data()  
'fooba'
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Dekoratoren mit Parametern (2)



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

```
>>> def limit(length:int):  
...     def decorator(f):  
...         def wrapper(*args, **kwargs):  
...             res = f(*args, **kwargs)  
...             return res[:length]  
...         return wrapper  
...     return decorator  
...  
>>> @limit(3)  
... def data_a():  
...     return 'limit to 3'  
...  
>>> @limit(6)  
... def data_b():  
...     return 'limit to 6'
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Was **passiert** hier?
- Der Aufruf von `limit(3)` erzeugt einen Dekorator, der auf `data_a` angewandt wird; `limit(6)` wenden wir auf `data_b` an:

```
>>> data_a()  
'lim'  
>>> data_b()  
'limit '
```

- Aber was passiert genau bei der geschachtelten Definition von Funktionen?

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# Funktionsschachtelung, Namensraum und Umgebung

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Im letzten Abschnitt sind uns **geschachtelte Funktionsdefinitionen** begegnet.
- Nun stellt sich die Frage, auf welche Bindung sich die Verwendung einer Variablen bezieht.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Im letzten Abschnitt sind uns **geschachtelte Funktionsdefinitionen** begegnet.
- Nun stellt sich die Frage, auf welche Bindung sich die Verwendung einer Variablen bezieht.
- Dafür müssen wir die Begriffe **Namensraum (Scope)** und **Umgebung** verstehen.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Im letzten Abschnitt sind uns **geschachtelte Funktionsdefinitionen** begegnet.
- Nun stellt sich die Frage, auf welche Bindung sich die Verwendung einer Variablen bezieht.
- Dafür müssen wir die Begriffe **Namensraum (Scope)** und **Umgebung** verstehen.
- Dabei ergeben sich zum Teil interessante Konsequenzen für die **Lebensdauer** einer Variablen.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Der Namensraum (Scope) ist ein statisches Konzept. Er zeigt an, in welchen Teilen eines Programms ein definierter Name sichtbar und verwendbar ist.
  - Ein Name kommt “in scope” durch
    - Definition einer Variable, Funktion oder Klasse
    - Import eines Moduls
- und ist verfügbar bis zum Ende des Blocks, in dem er definiert wurde.
- D.h. der lokale Namensraum einer Funktionsdefinition enthält die Parameter und lokale Variablen und reicht bis zum Ende des Funktionsrumpfes.
- Namensräume sind wie **Telefonvorwahlbereiche**. Sie sorgen dafür, dass gleiche Namen in verschiedenen Bereichen **nicht verwechselt** werden.
- Auf gleiche Variablennamen in verschiedenen Namensräumen kann oft mit der Punkt-Notation zugegriffen werden (insbesondere bei Modulen).

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - **Built-in**-Umgebung (`__builtins__`) mit allen vordefinierten Variablen

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - **Built-in**-Umgebung (`__builtins__`) mit allen vordefinierten Variablen
  - Umgebung von **Modulen**, die importiert werden

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - **Built-in**-Umgebung (`__builtins__`) mit allen vordefinierten Variablen
  - Umgebung von **Modulen**, die importiert werden
  - **globale** Umgebung (des Moduls `__main__`)

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - **Built-in**-Umgebung (`__builtins__`) mit allen vordefinierten Variablen
  - Umgebung von **Modulen**, die importiert werden
  - **globale** Umgebung (des Moduls `__main__`)
  - **lokale** Umgebung innerhalb einer Funktion (vgl. Kellerrahmen)  
diese können geschachtelt sein.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - **Built-in**-Umgebung (`__builtins__`) mit allen vordefinierten Variablen
  - Umgebung von **Modulen**, die importiert werden
  - **globale** Umgebung (des Moduls `__main__`)
  - **lokale** Umgebung innerhalb einer Funktion (vgl. Kellerrahmen) diese können geschachtelt sein.
- Die Umgebung ist ein dynamisches Konzept (d.h., zur Laufzeit).

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Umgebung ist ein dynamisches Konzept.
- Sie ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - **Built-in**-Umgebung (`__builtins__`) mit allen vordefinierten Variablen
  - Umgebung von **Modulen**, die importiert werden
  - **globale** Umgebung (des Moduls `__main__`)
  - **lokale** Umgebung innerhalb einer Funktion (vgl. Kellerrahmen) diese können geschachtelt sein.
- Die Umgebung ist ein dynamisches Konzept (d.h., zur Laufzeit).
- Die lokale Umgebung einer Funktion existiert normalerweise nur während ihres Aufrufs.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn in der **lokalen** Umgebungen aufzulösen;

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn in der **lokalen** Umgebungen aufzulösen;
  - ihn in den **nicht-lokalen** Umgebungen (die die lokale Umgebung umschließen) aufzulösen;

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn in der **lokalen** Umgebungen aufzulösen;
  - ihn in den **nicht-lokalen** Umgebungen (die die lokale Umgebung umschließen) aufzulösen;
  - ihn in der **globalen** Umgebung aufzulösen;

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn in der **lokalen** Umgebungen aufzulösen;
  - ihn in den **nicht-lokalen** Umgebungen (die die lokale Umgebung umschließen) aufzulösen;
  - ihn in der **globalen** Umgebung aufzulösen;
  - ihn in der **Builtin**-Umgebung aufzulösen.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Eine Variable heißt **sichtbar** in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn in der **lokalen** Umgebungen aufzulösen;
  - ihn in den **nicht-lokalen** Umgebungen (die die lokale Umgebung umschließen) aufzulösen;
  - ihn in der **globalen** Umgebung aufzulösen;
  - ihn in der **Builtin**-Umgebung aufzulösen.
- Dabei heißt “auflösen” das Auffinden des Werts der Variable.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Gibt es eine **Zuweisung** `var = ...` im aktuellen Scope, so wird von einem lokalen Namen ausgegangen und Referenzen auf `var` dürfen erst nach Ausführung der Zuweisung erfolgen.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Gibt es eine **Zuweisung** `var = ...` im aktuellen Scope, so wird von einem lokalen Namen ausgegangen und Referenzen auf `var` dürfen erst nach Ausführung der Zuweisung erfolgen.
- Ausnahmen:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Gibt es eine **Zuweisung** `var = ...` im aktuellen Scope, so wird von einem lokalen Namen ausgegangen und Referenzen auf `var` dürfen erst nach Ausführung der Zuweisung erfolgen.
- Ausnahmen:
  - „`global var`“ bedeutet, dass `var` in der **globalen** Umgebung gesucht werden soll. Auch Zuweisungen an `var` wirken auf die globale Umgebung.

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Ein Beispiel für Namensräume (1)



```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Ein Beispiel für Namensräume (2)



## Python-Interpreter

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Python-Interpreter

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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## Python-Interpreter

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

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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Ein Beispiel für Namensräume (2)



## 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, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



## 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, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Ein Beispiel für Namensräume (2)



## Python-Interpreter

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



# Closures

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



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

```
>>> def add_x(x:float) -> Callable[[float], float]:  
...     def adder(num:float) ->float:  
...         return x + num  
...         # adder is a closure  
...         # x is a free variable of adder  
...     return adder  
...  
>>> add_5 = add_x(5); add_5  
<function add_x.<locals>.adder at 0x103a8d750>  
>>> add_5(10)  
15
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures

# Closures in Python (2)



- Dasselbe mit einer `lambda` Abstraktion:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Dasselbe mit einer `lambda` Abstraktion:

```
>>> def add_x(x:float) -> Callable[[float], float]:  
...     return lambda num: x + num  
...     # returns a closure  
...     # num is a bound variable,  
...     # x is a free variable of the lambda  
...  
>>> add_6 = add_x(6); add_6  
<function add_x.<locals>.<lambda> at 0x103a8dbd0>  
>>> add_6(10)  
16
```

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Achtung bei der Interaktion von Closures mit Zuweisungen:

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Achtung bei der Interaktion von Closures mit Zuweisungen:

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

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

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

map, filter  
und reduce

Dekoratoren

Schachte-  
lung und  
Scope

Closures



- Definition: Eine Variable tritt **frei** in einem Funktionsrumpf auf, wenn sie zwar vorkommt, aber weder in der Parameterliste noch in einer lokalen Zuweisung gesetzt wird.
- Jede Funktion mit freien Variablen wird durch eine *Closure* repräsentiert.
- Innerhalb einer Closure kann mit Hilfe der Anweisungen `nonlocal` oder `global` auf freie Variable schreibend zugegriffen werden.
- In den beiden letzteren Fällen verlängert sich die **Lebensdauer** einer Umgebung (nämlich des umschließenden Funktionsaufrufs)! Sie bleibt so lange erhalten wie die Closure zugreifbar ist!

Funktionale  
Programmierung

FP in Python

Funktionen  
definieren  
und  
verwenden

Lambda-  
Notation

`map`, `filter`  
und `reduce`

Dekoratoren

Schachte-  
lung und  
Scope

Closures