Informatik I: Einführung in die Programmierung

16. Funktionale Programmierung

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Prof. Dr. Peter Thiemann

18.01.2022

UNI FREIBURG

Funktionale Programmierung

Programmierparadigmen

- Es gibt verschiedene Programmierparadigmen oder Programmierstile.
- Imperative Programmierung beschreibt, wie etwas erreicht werden soll.
- Deklarative Programmierung beschreibt, was erreicht werden soll.

Imperative Programmierparadigmen

UNI FREIBURG

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.

Imperative Programmierparadigmen

UNI FREIBURG

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.

Deklarative Programmierparadigmen

UNI FREIBURG

Deklarative Programmierung

- Keine explizite Bearbeitung eines Berechnungszustands.
- Logische Programmierung beschreibt das Ziel durch logische Formeln: Prolog, constraint programming, ASP.
- Funktionale Programmierung 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.

Eigenschaften funktionaler Programmierung

UNI FREIBURG

- Funktionen sind Bürger erster Klasse (*first-class citizens*).

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- Es gibt Funktionen höherer Ordnung, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- Es gibt Funktionen höherer Ordnung, d.h. Funktionen, deren Argumente und/oder Ergebnisse selbst wieder Funktionen sind.
- Keine Schleifen, sondern nur Rekursion.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- 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.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- 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 als Ausdrücke definierbar.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- 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 als Ausdrücke definierbar.
- In rein funktionalen Sprachen: keine Zuweisungen und keine Seiteneffekte.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- 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 als Ausdrücke definierbar.
- In rein funktionalen Sprachen: keine Zuweisungen und keine Seiteneffekte.

  $\Rightarrow$ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- 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 als Ausdrücke definierbar.
- In rein funktionalen Sprachen: keine Zuweisungen und keine Seiteneffekte.
  - $\Rightarrow$ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
  - $\Rightarrow$ Referentielle Transparenz: Eine Funktion liefert immer das gleiche Ergebnis bei gleichen Argumenten.

Eigenschaften funktionaler Programmierung

- Funktionen sind Bürger erster Klasse (*first-class citizens*).
- 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 als Ausdrücke definierbar.
- In rein funktionalen Sprachen: keine Zuweisungen und keine Seiteneffekte.
  - $\Rightarrow$ Eine Variable erhält zu Beginn ihren Wert, der sich nicht mehr ändert.
  - $\Rightarrow$ Referentielle Transparenz: Eine Funktion liefert immer das gleiche Ergebnis bei gleichen Argumenten.
- Die meisten funktionalen Sprachen besitzen ein starkes statisches Typsystem, sodass TypeError zur Laufzeit ausgeschlossen ist.

Attribute von Typsystemen

UNI FREIBURG

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 statisch, aber meist ineffizienter!

UNI FREIBURG

FP in Python

FP in Python

UNI FREIBURG

- Funktionen werden durch Objekte repräsentiert: „Bürger erster Klasse“.

FP in Python

UNI FREIBURG

- Funktionen werden durch Objekte repräsentiert: „Bürger erster Klasse“.
- Funktionen höherer Ordnung werden voll unterstützt.

FP in Python

UNI FREIBURG

- Funktionen werden durch Objekte repräsentiert: „Bürger erster Klasse“.
- Funktionen höherer Ordnung werden voll unterstützt.
- Python besitzt ein starkes dynamisches Typsystem.

FP in Python

UNI FREIBURG

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

FP in Python

- Funktionen werden durch Objekte repräsentiert: „Bürger erster Klasse“.
- 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.

FP in Python

UNI FREIBURG

- Funktionen werden durch Objekte repräsentiert: „Bürger erster Klasse“.
- 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.

FP in Python

- Funktionen werden durch Objekte repräsentiert: „Bürger erster Klasse“.
- 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.
- Einige Anwendungen von lazy evaluation können mit Generatoren modelliert werden. Beispiel: unendliche Sequenzen

FP in Python: Defizite

UNI FREIBURG

- 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.

FP in Python: Defizite

UNI FREIBURG

- 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.

FP in Python: Defizite

UNI FREIBURG

- 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.

UNI FREIBURG

Funktionen definieren und verwenden

Funktionsdefinition und -verwendung

UNI FREIBURG

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

Funktionsdefinition und -verwendung

UNI FREIBURG

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

Python-Interpreter

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

Funktionsdefinition und -verwendung

UNI FREIBURG

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

Python-Interpreter

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

Funktionsdefinition und -verwendung

UNI FREIBURG

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

Python-Interpreter

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

Funktionsdefinition und -verwendung

UNI FREIBURG

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

Python-Interpreter

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

Funktionsdefinition und -verwendung

UNI FREIBURG

- 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!
```

Funktionsdefinition und -verwendung

UNI FREIBURG

- 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 Python-Objekt.

Funktionsdefinition und -verwendung

UNI FREIBURG

- 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 Python-Objekt.
- Es kann zugewiesen werden, als Argument übergeben werden und als Funktionsresultat zurück gegeben werden.

Funktionsdefinition und -verwendung

UNI FREIBURG

- 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 Python-Objekt.
- Es kann zugewiesen werden, als Argument übergeben werden und als Funktionsresultat zurück gegeben werden.
- Und es ist aufrufbar ...

Funktionsverwendung

UNI FREIBURG

Python-Interpreter

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

Funktionsverwendung

UNI FREIBURG

Python-Interpreter

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

Funktionsverwendung

UNI FREIBURG

Python-Interpreter

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

Funktionsverwendung

UNI FREIBURG

Python-Interpreter

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

Funktionsverwendung

UNI FREIBURG

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
```

Funktionsverwendung

UNI FREIBURG

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
...
```

Funktionsverwendung

UNI FREIBURG

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()
```

Funktionsverwendung

UNI FREIBURG

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>
```

Funktionsverwendung

UNI FREIBURG

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()()
```

Funktionsverwendung

UNI FREIBURG

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
```

UNI FREIBURG

Lambda-Notation

Funktionen mit Lambda-Notation definieren

UNI FREIBURG

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

Funktionen mit Lambda-Notation definieren

UNI FREIBURG

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

Python-Interpreter

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

Funktionen mit Lambda-Notation definieren

UNI FREIBURG

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

Python-Interpreter

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

Funktionen mit Lambda-Notation definieren

UNI FREIBURG

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

Python-Interpreter

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

Funktionen mit Lambda-Notation definieren

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

Funktionen mit Lambda-Notation definieren

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

Der Typ `Callable`

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

Der Typ `Callable`

UNI FREIBURG

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

Python-Interpreter

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

Der Typ `Callable`

UNI FREIBURG

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

Python-Interpreter

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

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

Der Typ `Callable`

UNI FREIBURG

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

Python-Interpreter

```
>>> 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,

Der Typ `Callable`

UNI FREIBURG

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

Python-Interpreter

```
>>> 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.

Der Typ `Callable`

UNI FREIBURG

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

Python-Interpreter

```
>>> 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.
- Solch ein Typ wird für Funktionsparameter verwendet, die selbst Funktionen sind.

Verwendung von Lambda-Funktionen (1)

UNI FREIBURG

Python-Interpreter

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

- `mul2` ist äquivalent zu `mul`!

Verwendung von Lambda-Funktionen (1)

UNI FREIBURG

Python-Interpreter

```
>>> 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.

Verwendung von Lambda-Funktionen (1)

UNI FREIBURG

Python-Interpreter

```
>>> 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.
- Diese Argument-Funktionen werden oft nur einmal verwendet und sind kurz, sodass sich die Vergabe eines Namens nicht lohnt.

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)
```

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

UNI FREIBURG

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

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:

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:

Python-Interpreter

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

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:

Python-Interpreter

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

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:

Python-Interpreter

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

UNI FREIBURG

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

map: Anwendung einer Funktion auf Iteratierbares

UNI FREIBURG

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

`map`: Anwendung einer Funktion auf Iteratierbares

UNI FREIBURG

- `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 Iteratierbares

UNI FREIBURG

- 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]
```

Anwendungsbeispiel für `map`

UNI FREIBURG

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

Anwendungsbeispiel für `map`

UNI FREIBURG

- 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)
```

Anwendungsbeispiel für `map`

UNI FREIBURG

- 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)
```

- Ausgabe ist eingeschränkt auf Listen!

Anwendungsbeispiel für `map` (2)

UNI FREIBURG

- Als Generator: effizientere Ausgabe, weniger Einschränkungen

```
def gen_ctof (cl: Iterable[float]) -> Iterator[float]:
    for c in cl:
        yield ctof(c)
f_list = list (gen_ctof (c_list))
```

- Mit `map`: Vorteile wie Generator, noch 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` mit mehreren Eingaben

UNI FREIBURG

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

`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.

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 (vgl. convolute0)

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

Zip aus map

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

Zip aus map

UNI FREIBURG

- 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)]
```

Zip aus map

UNI FREIBURG

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

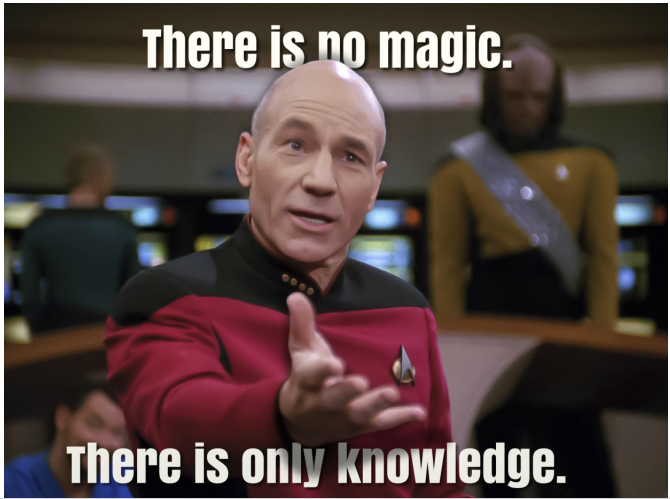
Python-Interpreter

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

- Volle zip-Funktionalität selbst gemacht:

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

*arg?



Einschub: Variable Parameterlisten

UNI FREIBURG

- 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.

`filter`: Filtert unpassende Objekte aus

UNI FREIBURG

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

`filter`: Filtert unpassende Objekte aus

UNI FREIBURG

- `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 unpassende Objekte aus

UNI FREIBURG

- `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]
```

`partial`: Partielle Anwendung von Funktionen

UNI FREIBURG

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

Beispiel

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

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')
```

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `from functools import reduce`

`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.

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `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:

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `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.

`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.

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `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.

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `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.

Python-Interpreter

```
>>> reduce(lambda x, y: x * y, range(1, 5))
24 # ((1 * 2) * 3) * 4
```

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `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.

Python-Interpreter

```
>>> reduce(lambda x, y: x * y, range(1, 5))
24 # ((1 * 2) * 3) * 4
```

`reduce`: Reduktion eines iterierbaren Objekts auf ein Element

UNI FREIBURG

- `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.

Python-Interpreter

```
>>> reduce(lambda x, y: x * y, range(1, 5))
24 # ((1 * 2) * 3) * 4
>>> def product(it: Iterable[float]) -> float:
...   return reduce (lambda x,y: x*y, it, 1)
```

Anwendung von `reduce` (1)

UNI FREIBURG

Python-Interpreter

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

Anwendung von `reduce` (1)

UNI FREIBURG

Python-Interpreter

```
>>> 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}
```

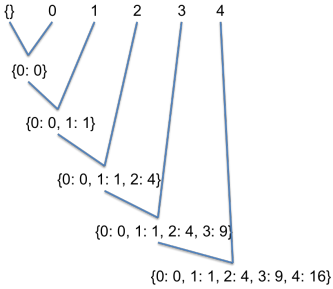
Anwendung von `reduce` (2)

UNI FREIBURG

- Was genau wird da schrittweise reduziert?

Anwendung von `reduce` (2)

- Was genau wird da schrittweise reduziert?



Einschub: Der echte Reduktionsoperator ist parallel!

UNI FREIBURG

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

Einschub: Der echte Reduktionsoperator ist parallel!

UNI FREIBURG

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

Einschub: Der echte Reduktionsoperator ist parallel!

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

Einschub: Der echte Reduktionsoperator ist parallel!

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

Einschub: Der echte Reduktionsoperator ist parallel!

UNI FREIBURG

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

Einschub: Der echte Reduktionsoperator ist parallel!

- Pythons `reduce` ist ein sogenannter **Fold Operator**.
  `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`
- Das echte `reduce`$(\oplus, [x_0, \ldots, x_{m-1}])$ 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 ...

Einschub: Der echte Reduktionsoperator ist parallel!

- Pythons `reduce` ist ein sogenannter **Fold Operator**.
  `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`
- Das echte `reduce`$(\oplus, [x_0, \ldots, x_{m-1}])$ 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 $x_0, x_2, \ldots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \ldots, (x_{m-2} \oplus x_{m-1})$

Einschub: Der echte Reduktionsoperator ist parallel!

- Pythons `reduce` ist ein sogenannter **Fold Operator**.
  `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`
- Das echte `reduce`$(\oplus, [x_0, \ldots, x_{m-1}])$ 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 $x_0, x_2, \ldots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \ldots, (x_{m-2} \oplus x_{m-1})$
- D.h. $m/2$ Operationen parallel in einem Schritt!

Einschub: Der echte Reduktionsoperator ist parallel!

- Pythons `reduce` ist ein sogenannter **Fold Operator**.
  `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`
- Das echte `reduce`$(\oplus, [x_0, \ldots, x_{m-1}])$ 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 $x_0, x_2, \ldots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \ldots, (x_{m-2} \oplus x_{m-1})$
- 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.

Einschub: Der echte Reduktionsoperator ist parallel!

UNI FREIBURG

- Pythons `reduce` ist ein sogenannter **Fold Operator**.
  `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`
- Das echte `reduce`$(\oplus, [x_0, \ldots, x_{m-1}])$ 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 $x_0, x_2, \ldots, x_{m-2} \leftarrow (x_0 \oplus x_1), (x_2 \oplus x_3), \ldots, (x_{m-2} \oplus x_{m-1})$
- 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.

UNI FREIBURG

Listen- und Generator-Comprehensionen

Listen-*Comprehension*

UNI FREIBURG

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

Listen-*Comprehension*

UNI FREIBURG

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

Listen-*Comprehension*

UNI FREIBURG

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. deklarativ und kompakt beschrieben werden.
- Entlehnt aus der funktionalen Programmiersprache Haskell (Miranda, KRC).
- Inspiriert von der mathematischen Mengenschreibweise: $\{x \in U \mid \phi(x)\}$ (alle x aus U, die die Bedingung ϕ erfüllen). Beispiel:

Listen-*Comprehension*

UNI FREIBURG

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. deklarativ und kompakt beschrieben werden.
- Entlehnt aus der funktionalen Programmiersprache Haskell (Miranda, KRC).
- Inspiriert von der mathematischen Mengenschreibweise: $\{x \in U \mid \phi(x)\}$ (alle x aus U, die die Bedingung ϕ erfüllen). Beispiel:

Python-Interpreter

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

Listen-*Comprehension*

UNI FREIBURG

- Mit *Comprehensions* (im Deutschen auch Abstraktionen) können Listen u.a. deklarativ und kompakt beschrieben werden.
- Entlehnt aus der funktionalen Programmiersprache Haskell (Miranda, KRC).
- Inspiriert von der mathematischen Mengenschreibweise: $\{x \in U \mid \phi(x)\}$ (alle x aus U, die die Bedingung ϕ erfüllen). Beispiel:

Python-Interpreter

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

- Bedeutung: Erstelle eine Liste aus allen `str(x)`, wobei `x` über das iterierbare Objekt `range(10)` läuft und nur die geraden Zahlen berücksichtigt werden.

Generelle Syntax von Listen-*Comprehensions*

```
[ expr  for pat1  in seq1  if cond1
        for pat2  in seq2  if cond2
        ...
        for patn  in seqn  if condn ]
```

- Die `if`-Klauseln mit den booleschen Ausdrücken $cond1, \ldots$ sind optional.

Generelle Syntax von Listen-*Comprehensions*

UNI FREIBURG

```
[ expr  for pat1  in seq1  if cond1
        for pat2  in seq2  if cond2
        ...
        for patn  in seqn  if condn ]
```

- Die `if`-Klauseln mit den booleschen Ausdrücken $cond1, \ldots$ sind optional.
- Ist *expr* ein Tupel, muss es in Klammern stehen!

Generelle Syntax von Listen-*Comprehensions*

```
[ expr  for pat1  in seq1  if cond1
        for pat2  in seq2  if cond2
        ...
        for patn  in seqn  if condn ]
```

- Die `if`-Klauseln mit den booleschen Ausdrücken $cond1, \ldots$ sind optional.
- Ist *expr* ein Tupel, muss es in Klammern stehen!
- Kurzschreibweise für Kombination aus `map` und `filter`.

Generelle Syntax von Listen-*Comprehensions*

```
[ expr  for pat1  in seq1  if cond1
        for pat2  in seq2  if cond2
        ...
        for patn  in seqn  if condn ]
```

- Die `if`-Klauseln mit den booleschen Ausdrücken $cond1, \ldots$ sind optional.
- Ist *expr* ein Tupel, muss es in Klammern stehen!
- Kurzschreibweise für Kombination aus `map` und `filter`.

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']
```

Zusammenhang Comprehensions vs `map` und `filter`

- Betrachte

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

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

Zusammenhang Comprehensions vs `map` und `filter`

UNI FREIBURG

- Betrachte

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

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

- Entspricht

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

Zusammenhang Comprehensions vs `map` und `filter`

UNI FREIBURG

- Betrachte

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

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

- Entspricht

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

- Falls `if` *cond* fehlt, kann das Filter weggelassen werden:

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

Geschachtelte Listen-*Comprehensions* (1)

UNI FREIBURG

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

Geschachtelte Listen-*Comprehensions* (1)

UNI FREIBURG

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

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> for y in range(3):
...     matrix += [list(range(4))]
...
>>> matrix
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

Geschachtelte Listen-*Comprehensions* (1)

UNI FREIBURG

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

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> for y in range(3):
...     matrix += [list(range(4))]
...
>>> matrix
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

- Lösung mit Listen-*Comprehensions*:

Geschachtelte Listen-*Comprehensions* (1)

UNI FREIBURG

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

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> for y in range(3):
...     matrix += [list(range(4))]
...
>>> matrix
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

- Lösung mit Listen-*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)

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

Geschachtelte Listen-*Comprehensions* (2)

UNI FREIBURG

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

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> for rownum in range(3):
...     row = []
...     for x in range(rownum*3, rownum*3 + 3):
...         row += [x+1]
...     matrix += [row]
...
```

Geschachtelte Listen-*Comprehensions* (2)

UNI FREIBURG

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

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> for rownum in range(3):
...     row = []
...     for x in range(rownum*3, rownum*3 + 3):
...         row += [x+1]
...     matrix += [row]
...
```

- Lösung mit Listen-*Comprehensions*:

Geschachtelte Listen-*Comprehensions* (2)

UNI FREIBURG

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

Python-Interpreter

```
>>> matrix: list[list[int]] = []
>>> for rownum in range(3):
...     row = []
...     for x in range(rownum*3, rownum*3 + 3):
...         row += [x+1]
...     matrix += [row]
...
```

- Lösung mit Listen-*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]]
```

Listen-*Comprehensions*: Kartesisches Produkt

UNI FREIBURG

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

Listen-*Comprehensions*: Kartesisches Produkt

UNI FREIBURG

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

Python-Interpreter

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

Listen-*Comprehensions*: Kartesisches Produkt

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

Python-Interpreter

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

- Lösung mit Listen-*Comprehensions*:

Listen-*Comprehensions*: Kartesisches Produkt

UNI FREIBURG

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

Python-Interpreter

```
>>> prod: list[tuple[int, str]] = []
>>> 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")
```

Kartesisches Produkt mit `map` und `filter`

UNI FREIBURG

- Erster Versuch

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

- Ergebnis

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

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
```

Kartesisches Produkt mit `map` und `filter`

UNI FREIBURG

- 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
```

- 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

```
X = TypeVar('X')
def flatten (iix : Iterable[Iterable[X]]) -> Iterator[X]:
    """flattens a nested iterable to a single iterator"""
    for ix in iix:
        for x in ix:
            yield x
```

Kartesisches Produkt mit `map`, `filter` und `flatten`

UNI FREIBURG

- Lösung: `flatten` entfernt eine Ebene von Iteration

```
X = TypeVar('X')
def flatten (iix : Iterable[Iterable[X]]) -> Iterator[X]:
    """flattens a nested iterable to a single iterator"""
    for ix in iix:
        for x in ix:
            yield x
```

- Damit

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

Kartesisches Produkt mit `map`, `filter` und `flatten`

UNI FREIBURG

- Lösung: `flatten` entfernt eine Ebene von Iteration

```
X = TypeVar('X')
def flatten (iix : Iterable[Iterable[X]]) -> Iterator[X]:
    """flattens a nested iterable to a single iterator"""
    for ix in iix:
        for x in ix:
            yield x
```

- Damit

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

- Ergebnis

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

Allgemein: Elimination von Listen-Comprehensions

Elimination des innersten `for`

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

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

Allgemein: Elimination von Listen-Comprehensions

UNI FREIBURG

Elimination des innersten `for`

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

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

Beispiel schematisch

```
[(x, y) for x in range(3) for y in "abc"]
```

Elimination von “`for x`” ergibt

```
flatten (map (lambda x: [(x, y) for y in "abc"], range(3)))
```

Elimination von “`for y`” ergibt

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

Generator-*Comprehension*

UNI FREIBURG

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

Generator-*Comprehension*

UNI FREIBURG

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

Generator-*Comprehension*

UNI FREIBURG

- Eine Variante der Comprehension baut keine Liste 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 als Argument einer Funktion mit nur einem Parameter dient.

Generator-*Comprehension*

UNI FREIBURG

- Eine Variante der Comprehension baut keine Liste 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 als Argument einer Funktion mit nur einem Parameter dient.

Python-Interpreter

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

Generator-*Comprehension*

UNI FREIBURG

- Eine Variante der Comprehension baut keine Liste 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 als Argument einer Funktion mit nur einem Parameter dient.

Python-Interpreter

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

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

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

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

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

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

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
```

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = {x for x in text if x >= 'a'}
```

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = {x for x in text if x >= 'a'}
>>>  print(res)
```

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = {x for x in text if x >= 'a'}
>>>  print(res)
{'a', 'o', 'm', 'n', 'e', 'i', 'g', 'r', 'u', 't', 's'}
```

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = {x for x in text if x >= 'a'}
>>>  print(res)
{'a', 'o', 'm', 'n', 'e', 'i', 'g', 'r', 'u', 't', 's'}
>>>  d = { x: x**2 for x in range(1, 10)}
```

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = {x for x in text if x >= 'a'}
>>>  print(res)
{'a', 'o', 'm', 'n', 'e', 'i', 'g', 'r', 'u', 't', 's'}
>>>  d = { x: x**2 for x in range(1, 10)}
>>>  print(d)
```

Comprehension für Dictionaries und Mengen

UNI FREIBURG

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

Python-Interpreter

```
>>> evens = set(range(0, 20, 2))
>>> evenmultsofthree = {x for x in evens if x % 3 == 0}
>>> evenmultsofthree
{0, 18, 12, 6}
>>>  text = 'Management Training Course'
>>>  res = {x for x in text if x >= 'a'}
>>>  print(res)
{'a', 'o', 'm', 'n', 'e', 'i', 'g', 'r', 'u', 't', 's'}
>>>  d = { x: x**2 for x in range(1, 10)}
>>>  print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

UNI FREIBURG

Dekoratoren

Dekoratoren

UNI FREIBURG

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

Dekoratoren

UNI FREIBURG

Dekoratoren werden durch Funktionen, die Funktionen als Parameter nehmen und zurückgeben, implementiert.
Dekoratoren, die uns schon früher begegnet sind: `dataclass`, `property`, etc.

Dekoratoren

UNI FREIBURG

Dekoratoren werden durch Funktionen, die Funktionen als Parameter nehmen und zurückgeben, implementiert.
Dekoratoren, die uns schon früher begegnet sind: `dataclass`, `property`, etc.
Es gibt eine spezielle Syntax, um solche Dekoratoren anzuwenden. Falls der Dekorator `wrapper` definiert wurde:

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

können wir auch schreiben:

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

Dekoratoren: `property`, `staticmethod` (1)

UNI FREIBURG

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)
```

Dekoratoren: `property`, `staticmethod` (2)

UNI FREIBURG

decorators.py

```
@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")
```

Definition eines Dekorators (1)

UNI FREIBURG

Aufgabe

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

Definition eines Dekorators (1)

UNI FREIBURG

Aufgabe

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

decorators.py

```
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
```

Definition eines Dekorators (1)

UNI FREIBURG

Aufgabe

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

decorators.py

```
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 ...

Definition eines Dekorators (2)

Wiederverwendbare modulare Lösung

UNI FREIBURG

decorators.py

```
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
```

Definition eines Dekorators (3)

UNI FREIBURG

Aufgabe

Wie lange dauert die Ausführung eines Funktionsaufrufs?

Definition eines Dekorators (3)

UNI FREIBURG

Aufgabe

Wie lange dauert die Ausführung eines Funktionsaufrufs?

decorators.py

```
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
```

Definition eines Dekorators (4)

Dekoratoren hintereinander schalten

UNI FREIBURG

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
```

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

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

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

UNI FREIBURG

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

decorators.py

```
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
```

Dekoratoren: `docstring` und `__name__` (2)

UNI FREIBURG

- Ein Standardproblem, das selbst durch einen Dekorator gelöst werden kann:

decorators.py

```
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
```

Dekoratoren mit Parametern (1)

UNI FREIBURG

- 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'
```

Dekoratoren mit Parametern (1)

UNI FREIBURG

- 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:

Dekoratoren mit Parametern (1)

- 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:

Python-Interpreter

```
>>> data()
```

Dekoratoren mit Parametern (1)

- 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:

Python-Interpreter

```
>>> data()
```

Dekoratoren mit Parametern (1)

UNI FREIBURG

- 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:

Python-Interpreter

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

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'
```

Parametrische Dekoratoren (3)

UNI FREIBURG

- 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:

Python-Interpreter

```
>>> data_a()
```

Parametrische Dekoratoren (3)

UNI FREIBURG

- 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:

Python-Interpreter

```
>>> data_a()
'lim'
```

Parametrische Dekoratoren (3)

- 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:

Python-Interpreter

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

Parametrische Dekoratoren (3)

UNI FREIBURG

- 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:

Python-Interpreter

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

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

UNI FREIBURG

Funktionsschachtelung, Namensräume und Scope

Geschachtelte Funktionsdefinitionen

UNI FREIBURG

- Im letzten Abschnitt sind uns geschachtelte Funktionsdefinitionen begegnet.

Geschachtelte Funktionsdefinitionen

UNI FREIBURG

- Im letzten Abschnitt sind uns geschachtelte Funktionsdefinitionen begegnet.
- Dabei müssen wir klären, auf welche Bindung sich die Verwendung einer Variablen bezieht.

Geschachtelte Funktionsdefinitionen

UNI FREIBURG

- Im letzten Abschnitt sind uns geschachtelte Funktionsdefinitionen begegnet.
- Dabei müssen wir klären, auf welche Bindung sich die Verwendung einer Variablen bezieht.
- Dafür müssen wir die Begriffe Namensraum (*name space*) und Scope oder Gültigkeitsbereich (*scope*) verstehen.

Geschachtelte Funktionsdefinitionen

UNI FREIBURG

- Im letzten Abschnitt sind uns geschachtelte Funktionsdefinitionen begegnet.
- Dabei müssen wir klären, auf welche Bindung sich die Verwendung einer Variablen bezieht.
- Dafür müssen wir die Begriffe Namensraum (*name space*) und Scope oder Gültigkeitsbereich (*scope*) verstehen.
- Dabei ergeben sich zum Teil interessante Konsequenzen für die Lebensdauer einer Variablen.

Namensräume

UNI FREIBURG

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

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
  - Namensraum von Modulen, die importiert werden

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
  - Namensraum von Modulen, die importiert werden
  - globaler Namensraum (des Moduls `__main__`)

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
  - Namensraum von Modulen, die importiert werden
  - globaler Namensraum (des Moduls `__main__`)
  - lokaler Namensraum innerhalb einer Funktion

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
  - Namensraum von Modulen, die importiert werden
  - globaler Namensraum (des Moduls `__main__`)
  - lokaler Namensraum innerhalb einer Funktion
- Namensräume haben verschiedene Lebensdauern; der lokale Namensraum einer Funktion existiert z.B. nur während ihres Aufrufs.

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
  - Namensraum von Modulen, die importiert werden
  - globaler Namensraum (des Moduls `__main__`)
  - lokaler Namensraum innerhalb einer Funktion
- Namensräume haben verschiedene Lebensdauern; der lokale Namensraum einer Funktion existiert z.B. nur während ihres Aufrufs.

→ Namensräume sind wie Telefonvorwahlbereiche. Sie sorgen dafür, dass gleiche Namen in verschiedenen Bereichen nicht verwechselt werden.

Namensräume

UNI FREIBURG

- Ein Namensraum ist eine Abbildung von Namen auf Werte (intern oft durch ein `dict` realisiert).
  - `Built-in`-Namensraum (`__builtins__`) mit allen vordefinierten Variablen
  - Namensraum von Modulen, die importiert werden
  - globaler Namensraum (des Moduls `__main__`)
  - lokaler Namensraum innerhalb einer Funktion
- Namensräume haben verschiedene Lebensdauern; der lokale Namensraum einer Funktion existiert z.B. 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).

Gültigkeitsbereiche

UNI FREIBURG

- 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.

Gültigkeitsbereiche

UNI FREIBURG

- 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 alle äußeren überdeckt!

Gültigkeitsbereiche

UNI FREIBURG

- 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 alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:

Gültigkeitsbereiche

UNI FREIBURG

- 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 alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn im lokalen Scope aufzulösen;

Gültigkeitsbereiche

- 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 alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn im lokalen Scope aufzulösen;
  - ihn im nicht-lokalen Scope aufzulösen;

Gültigkeitsbereiche

UNI FREIBURG

- 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 alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn im lokalen Scope aufzulösen;
  - ihn im nicht-lokalen Scope aufzulösen;
  - ihn im globalen Scope aufzulösen;

Gültigkeitsbereiche

- 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 alle äußeren überdeckt!
- Wird ein Variablenname zum Lesen referenziert, so versucht Python der Reihe nach:
  - ihn im lokalen Scope aufzulösen;
  - ihn im nicht-lokalen Scope aufzulösen;
  - ihn im globalen Scope aufzulösen;
  - ihn im `Builtin`-Namensraum aufzulösen.

Namensraum-Konventionen

UNI FREIBURG

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

Namensraum-Konventionen

UNI FREIBURG

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

Namensraum-Konventionen

UNI FREIBURG

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

Namensraum-Konventionen

UNI FREIBURG

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

Namensraum-Konventionen

UNI FREIBURG

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

Namensraum-Konventionen

UNI FREIBURG

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

UNI FREIBURG

```
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)
```

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)

UNI FREIBURG

Python-Interpreter

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

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)

UNI FREIBURG

Python-Interpreter

```
>>> scope_test()
After local assignment: test spam
```

Ein Beispiel für Namensräume und Gültigkeitsbereiche (2)

UNI FREIBURG

Python-Interpreter

```
>>> scope_test()
After local assignment: test spam
After nonlocal assignment: nonlocal spam
```

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
```

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)
```

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
```

UNI FREIBURG

Closures

Closures in Python

UNI FREIBURG

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

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # adder is a closure
...         # x is a free variable
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # adder is a closure
...         # x is a free variable
...         return x + num
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # adder is a closure
...         # x is a free variable
...         return x + num
...     return adder
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # adder is a closure
...         # x is a free variable
...         return x + num
...     return adder
...
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # adder is a closure
...         # x is a free variable
...         return x + num
...     return adder
...
>>> add_5 = add_x(5); add_5
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

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

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # 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)
```

Closures in Python

UNI FREIBURG

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

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     def adder(num:float) ->float:
...         # 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
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     return lambda num: x+num
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     return lambda num: x+num
...         # returns a closure
...         # x is a free variable of the lambda
```

Closures in Python (2)

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     return lambda num: x+num
...         # returns a closure
...         # x is a free variable of the lambda
...
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     return lambda num: x+num
...         # returns a closure
...         # x is a free variable of the lambda
...
>>> add_5 = add_x(5); add_5
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     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 ...>
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     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)
```

Closures in Python (2)

UNI FREIBURG

- Dasselbe mit einer `lambda` Abstraktion:

Python-Interpreter

```
>>> def add_x(x:float) -> Callable[[float], float]:
...     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
```

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

```
>>> def clo() -> Callable[[], int]:
```

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

```
>>> def clo() -> Callable[[], int]:
...     x = 0
```

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

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

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

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

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

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

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

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

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

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

Closures in Python (3)

UNI FREIBURG

- Achtung bei der Interaktion von Closures mit Zuweisungen:

Python-Interpreter

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

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

Zusammenfassung Closures

UNI FREIBURG

- 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 wird die Lebensdauer eines Namensraums (nämlich der umschliessenden Funktion) nicht notwendig bei Verlassen der Closure beendet!