Informatik I: Einführung in die Programmierung

6. Sequenzen und Programme entwickeln

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Peter Thiemann

13. November 2019

UNI FREIBURG

Sequenzen

Sequenzen

UNI FREIBURG

Pythons Sequenztypen

- Strings: `str`
- Tupel: `tuple`
- Listen: `list`

Programmieren mit Sequenzen

- Gemeinsame Operationen
- Iteration (`for`-Schleifen)

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>> print(primes[1], sum(primes))
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>> print(primes[1], sum(primes))
3 17
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>> print(primes[1], sum(primes))
3 17
>>> squares = (1, 4, 9, 16, 25)
>>>
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>> print(primes[1], sum(primes))
3 17
>>> squares = (1, 4, 9, 16, 25)
>>> print(squares[1:4])
```

Beispiel zu Sequenzen

UNI FREIBURG

Python-Interpreter

```
>>> first_name = "Johann"
>>> last_name = 'Gambolputty'
>>> name = first_name + " " + last_name
>>> print(name)
Johann Gambolputty
>>> print(name.split())
['Johann', 'Gambolputty']
>>> primes = [2, 3, 5, 7]
>>> print(primes[1], sum(primes))
3 17
>>> squares = (1, 4, 9, 16, 25)
>>> print(squares[1:4])
(4, 9, 16)
```

Sequenzoperationen

- + Verkettung von Sequenzen
- `s[0]` Indizierung ab 0 (Zugriff aufs erste Element)
- `s[1:3]` Teilsequenz (vom 2. bis 4. Element)
- Weitere Varianten siehe Dokumentation

Strings

- Strings in Python enthalten Unicode-Zeichen.
- Strings werden meistens `"auf diese Weise"` angegeben, es gibt aber viele alternative Schreibweisen.

Tupel und Listen

- Ein Tupel bzw. eine Liste ist eine Sequenz von Objekten.
- Tupel werden mit runden, Listen mit eckigen Klammern geschrieben:
  `(2, 1, "Risiko")` vs. `["red", "green", "blue"]`.
- Tupel und Listen können beliebige Objekte enthalten, natürlich auch andere Tupel und Listen:
  `([18, 20, 22, "Null"], [("spam", [])])`
- Der Hauptunterschied zwischen Tupeln und Listen:
  - Listen sind *veränderlich* (mutable).
    Elemente anhängen, einfügen oder entfernen.
  - Tupel sind *unveränderlich* (immutable).
    Ein Tupel ändert sich nie, es enthält immer dieselben Objekte in derselben Reihenfolge. (Allerdings können sich die *enthaltenen* Objekte verändern, z.B. bei Tupeln von Listen.)

Tuple Unpacking

UNI FREIBURG

- Schreibe `(a, b) = (2, 3)` für die komponentenweise Zuweisung von Tupeln (Tuple Unpacking aka Pattern Matching).
- Effekt wie `a = 2` und `b = 3`, nur dass die Zuweisungen gleichzeitig erfolgen.
- Nach `(b, a) = (a, b)` sind die Inhalte von `a` und `b` vertauscht!
- Tuple Unpacking funktioniert auch mit Listen und Strings und lässt sich beliebig schachteln:

Python-Interpreter

```
>>> [a, (b, c), (d, e), f] = (42, (6, 9), "do", [1, 2, 3])
>>>
```

Tuple Unpacking

UNI FREIBURG

- Schreibe `(a, b) = (2, 3)` für die komponentenweise Zuweisung von Tupeln (Tuple Unpacking aka Pattern Matching).
- Effekt wie `a = 2` und `b = 3`, nur dass die Zuweisungen gleichzeitig erfolgen.
- Nach `(b, a) = (a, b)` sind die Inhalte von `a` und `b` vertauscht!
- Tuple Unpacking funktioniert auch mit Listen und Strings und lässt sich beliebig schachteln:

Python-Interpreter

```
>>> [a, (b, c), (d, e), f] = (42, (6, 9), "do", [1, 2, 3])
>>> print(a, "*", b, "*", c, "*", d, "*", e, "*", f)
```

Tuple Unpacking

UNI FREIBURG

- Schreibe `(a, b) = (2, 3)` für die komponentenweise Zuweisung von Tupeln (Tuple Unpacking aka Pattern Matching).
- Effekt wie `a = 2` und `b = 3`, nur dass die Zuweisungen gleichzeitig erfolgen.
- Nach `(b, a) = (a, b)` sind die Inhalte von `a` und `b` vertauscht!
- Tuple Unpacking funktioniert auch mit Listen und Strings und lässt sich beliebig schachteln:

Python-Interpreter

```
>>> [a, (b, c), (d, e), f] = (42, (6, 9), "do", [1, 2, 3])
>>> print(a, "*", b, "*", c, "*", d, "*", e, "*", f)
42 * 6 * 9 * d * o * [1, 2, 3]
```

Operationen auf Sequenzen

Sequenzen

UNI FREIBURG

- Strings, Tupel und Listen haben etwas gemeinsam: Sie enthalten andere Objekte in einer bestimmten Reihenfolge und erlauben direkten Zugriff auf die einzelnen Komponenten mittels Indizierung.
- Typen mit dieser Eigenschaft heißen Sequenztypen, ihre Instanzen Sequenzen.

Sequenzen

UNI FREIBURG

- Strings, Tupel und Listen haben etwas gemeinsam: Sie enthalten andere Objekte in einer bestimmten Reihenfolge und erlauben direkten Zugriff auf die einzelnen Komponenten mittels Indizierung.
- Typen mit dieser Eigenschaft heißen Sequenztypen, ihre Instanzen Sequenzen.

Sequenztypen unterstützen die folgenden Operationen:

- Verkettung: `"Gambol" + "putty" == "Gambolputty"`
- Wiederholung: `2 * "spam" == "spamspam"`
- Indizierung: `"Python"[1] == "y"`
- Mitgliedschaftstest: `17 in [11,13,17,19]`
- Slicing: `"Monty Python's Flying Circus"[6:12] == "Python"`
- Iteration: `for x in "egg"`

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>>
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>>
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>>
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>>
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>>
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>> print(mylist + primes)
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>> print(mylist + primes)
Traceback (most recent call last): ...
TypeError: can only concatenate list (not "tuple") to list
>>>
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>> print(mylist + primes)
Traceback (most recent call last): ...
TypeError: can only concatenate list (not "tuple") to list
>>> print(mylist + list(primes))
```

Verkettung

UNI FREIBURG

Python-Interpreter

```
>>> print("Gambol" + "putty")
Gambolputty
>>> mylist = ["spam", "egg"]
>>> print(["spam"] + mylist)
['spam', 'spam', 'egg']
>>> primes = (2, 3, 5, 7)
>>> print(primes + primes)
(2, 3, 5, 7, 2, 3, 5, 7)
>>> print(mylist + primes)
Traceback (most recent call last): ...
TypeError: can only concatenate list (not "tuple") to list
>>> print(mylist + list(primes))
['spam', 'egg', 2, 3, 5, 7]
```

Wiederholung

UNI FREIBURG

Python-Interpreter

```
>>> print("*" * 20)
```

Wiederholung

UNI FREIBURG

Python-Interpreter

```
>>> print("*" * 20)
********************
>>>
```

Wiederholung

Python-Interpreter

```
>>> print("*" * 20)
********************
>>> print([None, 2, 3] * 3)
```

Wiederholung

UNI FREIBURG

Python-Interpreter

```
>>> print("*" * 20)
********************
>>> print([None, 2, 3] * 3)
[None, 2, 3, None, 2, 3, None, 2, 3]
>>>
```

Wiederholung

UNI FREIBURG

Python-Interpreter

```
>>> print("*" * 20)
********************
>>> print([None, 2, 3] * 3)
[None, 2, 3, None, 2, 3, None, 2, 3]
>>> print(2 * ("parrot", ["is", "dead"]))
```

Wiederholung

Python-Interpreter

```
>>> print("*" * 20)
********************
>>> print([None, 2, 3] * 3)
[None, 2, 3, None, 2, 3, None, 2, 3]
>>> print(2 * ("parrot", ["is", "dead"]))
('parrot', ['is', 'dead'], 'parrot', ['is', 'dead'])
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>>
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>>
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>> animal = "parrot"
>>>
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>> animal = "parrot"
>>> animal[-2]
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>> animal = "parrot"
>>> animal[-2]
'o'
>>>
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>> animal = "parrot"
>>> animal[-2]
'o'
>>> animal[10]
```

Indizierung

UNI FREIBURG

- Sequenzen können von vorne und von hinten indiziert werden.
- Bei Indizierung von vorne hat das erste Element Index 0.
- Zur Indizierung von hinten dienen negative Indizes. Dabei hat das letzte Element den Index -1.

Python-Interpreter

```
>>> primes = (2, 3, 5, 7, 11, 13)
>>> print(primes[1], primes[-1])
3 13
>>> animal = "parrot"
>>> animal[-2]
'o'
>>> animal[10]
Traceback (most recent call last): ...
IndexError: string index out of range
```

Mitgliedschaftstest: Der `in`-Operator

UNI FREIBURG

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- `substring in string` (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
```

Mitgliedschaftstest: Der `in`-Operator

UNI FREIBURG

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- `substring in string` (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
True
>>>
```

Mitgliedschaftstest: Der `in`-Operator

UNI FREIBURG

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- `substring in string` (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
True
>>> if "spam" in ("ham", "eggs", "sausage"):
...   print("tasty")
...
```

Mitgliedschaftstest: Der `in`-Operator

UNI FREIBURG

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- `substring in string` (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
True
>>> if "spam" in ("ham", "eggs", "sausage"):
...   print("tasty")
...
>>>
```

Mitgliedschaftstest: Der `in`-Operator

UNI FREIBURG

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- `substring in string` (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
True
>>> if "spam" in ("ham", "eggs", "sausage"):
...   print("tasty")
...
>>> print("m" in "spam", "ham" in "spam", "pam" in "spam")
```

Mitgliedschaftstest: Der `in`-Operator

UNI FREIBURG

- `item in seq` (`seq` ist ein Tupel oder eine Liste):
  `True`, wenn `seq` das Element `item` enthält.
- `substring in string` (`string` ist ein String):
  `True`, wenn `string` den Teilstring `substring` enthält.

Python-Interpreter

```
>>> print(2 in [1, 4, 2])
True
>>> if "spam" in ("ham", "eggs", "sausage"):
...   print("tasty")
...
>>> print("m" in "spam", "ham" in "spam", "pam" in "spam")
True False True
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>>
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
[3, 5, 7]
>>>
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
[3, 5, 7]
>>> print(primes[:2])
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
[3, 5, 7]
>>> print(primes[:2])
[2, 3]
>>>
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
[3, 5, 7]
>>> print(primes[:2])
[2, 3]
>>> print("egg, sausage and bacon"[-5:])
```

Slicing

UNI FREIBURG

- *Slicing*: Ausschneiden von ‚Scheiben' aus einer Sequenz:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> print(primes[1:4])
[3, 5, 7]
>>> print(primes[:2])
[2, 3]
>>> print("egg, sausage and bacon"[-5:])
bacon
```

Slicing: Erklärung

- `seq[i:j]` liefert den Bereich $[i,j)$, also die Elemente an den Positionen $i, i+1, \ldots, j-1$:
  `("do", "re", 5, 7)[1:3] == ("re", 5)`
- Ohne i beginnt der Bereich an Position 0:
  `("do", "re", 5, 7)[:3] == ("do", "re", 5)`
- Ohne j endet der Bereich am Ende der Folge:
  `("do", "re", 5, 7)[1:] == ("re", 5, 7)`
- Der slice Operator `[:]` liefert eine Kopie der Folge:
  `("do", "re", 5, 7)[:] == ("do", "re", 5, 7)`

Slicing: Erklärung (2)

UNI FREIBURG

- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "spam"[2:10]
```

Slicing: Erklärung (2)

UNI FREIBURG

- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "spam"[2:10]
'am'
>>>
```

Slicing: Erklärung (2)

UNI FREIBURG

- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "spam"[2:10]
'am'
>>> "spam"[-6:3]
```

Slicing: Erklärung (2)

UNI FREIBURG

- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "spam"[2:10]
'am'
>>> "spam"[-6:3]
'spa'
>>>
```

Slicing: Erklärung (2)

UNI FREIBURG

- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "spam"[2:10]
'am'
>>> "spam"[-6:3]
'spa'
>>> "spam"[7:]
```

Slicing: Erklärung (2)

UNI FREIBURG

- Beim Slicing gibt es keine Indexfehler. Bereiche jenseits des Endes der Folge sind leer.

Python-Interpreter

```
>>> "spam"[2:10]
'am'
>>> "spam"[-6:3]
'spa'
>>> "spam"[7:]
''
```

- Auch Slicing kann ‚von hinten zählen'.
  Z.B. liefert `seq[-3:]` die drei letzten Elemente.

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>>
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
[1, 3, 5, 7]
>>>
```

Slicing: Schrittweite

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
[1, 3, 5, 7]
>>> zahlen[7:2:-1]
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
[1, 3, 5, 7]
>>> zahlen[7:2:-1]
[7, 6, 5, 4, 3]
>>>
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
[1, 3, 5, 7]
>>> zahlen[7:2:-1]
[7, 6, 5, 4, 3]
>>> zahlen[::-1]
```

Slicing: Schrittweite

UNI FREIBURG

- Beim *erweiterten Slicing* gibt es zusätzlich noch eine Schrittweite:

Python-Interpreter

```
>>> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> zahlen[1:7:2]
[1, 3, 5]
>>> zahlen[1:8:2]
[1, 3, 5, 7]
>>> zahlen[7:2:-1]
[7, 6, 5, 4, 3]
>>> zahlen[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>>
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>>
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>>
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
['s', 'p', 'a', 'm']
>>>
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple('spam')
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple('spam')
('s', 'p', 'a', 'm')
>>>
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple('spam')
('s', 'p', 'a', 'm')
>>> str(['a', 'b', 'c'])
```

Typkonversion

UNI FREIBURG

Die Funktionen list, tuple, und str konvertieren zwischen den Sequenztypen (aber manchmal unerwartet).

Python-Interpreter

```
>>> tuple([0, 1, 2])
(0, 1, 2)
>>> list(('spam', 'egg'))
['spam', 'egg']
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple('spam')
('s', 'p', 'a', 'm')
>>> str(['a', 'b', 'c'])
"['a', 'b', 'c']"
```

Weitere Sequenzoperationen 1

UNI FREIBURG

- `sum(seq)`:
  Berechnet die Summe einer Zahlensequenz.
- `min(seq)`, `min(x, y, ...)`:
  Berechnet das Minimum einer Sequenz (erste Form)
  bzw. der Argumente (zweite Form).
  - Sequenzen werden lexikographisch verglichen.
  - Der Versuch, das Minimum konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen) zu bilden, führt zu einem `TypeError`.
- `max(seq)`, `max(x, y, ...)`: $\rightsquigarrow$ analog zu `min`

Python-Interpreter

```
>>> max([1, 23, 42, 5])
```

Weitere Sequenzoperationen 1

UNI FREIBURG

- `sum(seq)`:
  Berechnet die Summe einer Zahlensequenz.
- `min(seq)`, `min(x, y, ...)`:
  Berechnet das Minimum einer Sequenz (erste Form)
  bzw. der Argumente (zweite Form).
  - Sequenzen werden lexikographisch verglichen.
  - Der Versuch, das Minimum konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen) zu bilden, führt zu einem `TypeError`.
- `max(seq)`, `max(x, y, ...)`: $\rightsquigarrow$ analog zu `min`

Python-Interpreter

```
>>> max([1, 23, 42, 5])
42
>>>
```

Weitere Sequenzoperationen 1

UNI FREIBURG

- `sum(seq)`:
  Berechnet die Summe einer Zahlensequenz.
- `min(seq)`, `min(x, y, ...)`:
  Berechnet das Minimum einer Sequenz (erste Form)
  bzw. der Argumente (zweite Form).
  - Sequenzen werden lexikographisch verglichen.
  - Der Versuch, das Minimum konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen) zu bilden, führt zu einem `TypeError`.
- `max(seq)`, `max(x, y, ...)`: $\rightsquigarrow$ analog zu `min`

Python-Interpreter

```
>>> max([1, 23, 42, 5])
42
>>> sum([1, 23, 42, 5])
```

Weitere Sequenzoperationen 1

- `sum(seq)`:
  Berechnet die Summe einer Zahlensequenz.
- `min(seq)`, `min(x, y, ...)`:
  Berechnet das Minimum einer Sequenz (erste Form)
  bzw. der Argumente (zweite Form).
  - Sequenzen werden lexikographisch verglichen.
  - Der Versuch, das Minimum konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen) zu bilden, führt zu einem `TypeError`.
- `max(seq)`, `max(x, y, ...)`: $\rightsquigarrow$ analog zu `min`

Python-Interpreter

```
>>> max([1, 23, 42, 5])
42
>>> sum([1, 23, 42, 5])
71
```

Weitere Sequenzoperationen 2

- `any(seq)`:
  Äquivalent zu `elem1 or elem2 or elem3 or ...`, wobei `elem`*i* die Elemente von `seq` sind und nur `True` oder `False` zurück geliefert wird.
- `all(seq)`: $\rightsquigarrow$ analog zu `any`, aber mit
  `elem1 and elem2 and elem3 and ...`

Weitere Sequenzoperationen 3

- `len(seq)`:
  Berechnet die Länge einer Sequenz.
- `sorted(seq)`:
  Liefert eine Liste, die dieselben Elemente hat wie `seq`, aber (stabil) sortiert ist.

Iteration

Iteration

- Durchlaufen von Sequenzen mit `for`-Schleifen:

Python-Interpreter

```
>>> primes = [2, 3, 5, 7]
>>> product = 1
>>> for number in primes:
...     product = product * number
...
>>> print(product)
210
```

Visualisierung

Iteration (1)

UNI FREIBURG

- Bestandteile der `for`-Schleife:

```
for number in primes:                    # Schleifenkopf
    product = product * number  # Schleifenrumpf
```

- Schlüsselworte `for` und `in`
- `number`: Schleifenvariable, Laufvariable, Iterationsvariable
- `primes`: allgemein ein Ausdruck, dessen Wert durchlaufen werden kann. Z.B. eine Sequenz
- Der Schleifenrumpf ist wieder eingerückt und muss aus mindestens einer (oder mehreren) Anweisungen bestehen.
- Beliebige Schachtelung ist möglich.

Iteration (2)

- `for` funktioniert mit allen Sequenztypen:

Python-Interpreter

```
>>> for character in "spam":
...   print(character * 2)
...
ss
pp
aa
mm
>>> for ingredient in ("spam", "spam", "egg"):
...   if ingredient == "spam":
...     print("tasty!")
...
tasty!
tasty!
```

break, continue, else

Im Zusammenhang mit Schleifen sind die folgenden drei Anweisungen interessant:

- `break` beendet eine Schleife vorzeitig.
- `continue` beendet die aktuelle Schleifeniteration vorzeitig, d.h. springt zum Schleifenkopf und setzt die Schleifenvariable auf den nächsten Wert.
- Schleifen können einen `else`-Zweig haben. Dieser wird nach Beendigung der Schleife ausgeführt, und zwar genau dann, wenn die Schleife *nicht* mit `break` verlassen wurde.

break, continue und else: Beispiel

UNI FREIBURG


```
foods_and_amounts = [("sausage", 2), ("eggs", 0),
                     ("spam", 2), ("ham", 1)]

for fa in foods_and_amounts:
    (food, amount) = fa
    if amount == 0:
        continue
    if food == "spam":
        print(amount, "tasty piece(s) of spam.")
        break
else:
    print("No spam!")

# Ausgabe:
# 2 tasty piece(s) of spam.
```

Nützliche Funktionen im Zusammenhang mit `for`-Schleifen

Einige Funktionen tauchen häufig im Zusammenhang mit `for`-Schleifen auf:

- `range`
- `zip`
- `reversed`

range

- Konzeptuell erzeugt `range` eine Liste von Indexen für Schleifendurchläufe:
  - **`range`**`(stop)` ergibt
    `0, 1, ..., stop-1`
  - **`range`**`(start, stop)` ergibt
    `start, start+1, ..., stop-1`
  - **`range`**`(start, stop, step)` ergibt
    `start, start + step, start + 2 * step, ..., start + n * step`
    `n` ist bestimmt durch
    `start + n * step < stop <= start + (n + 1) * step`
- `range` vermeidet aber das Anlegen einer Liste!

range: Beispiele

UNI FREIBURG

Python-Interpreter

```
>>> range(5)
range(0, 5)
>>> range(3, 30, 10)
range(3, 30, 10)
>>> list(range(3, 30, 10))
[3, 13, 23]
>>> for i in range(3, 6):
...   print(i, "** 3 =", i ** 3)
...
3 ** 3 = 27
4 ** 3 = 64
5 ** 3 = 125
```

zip (1)

- Die Funktion `zip` nimmt eine oder mehrere Sequenzen und liefert eine Liste von Tupeln mit korrespondierenden Elementen.
- Auch `zip` erzeugt keine Liste, sondern ein spezielles Objekt; der Listen-Konstruktor erzeugt daraus eine richtige Liste.

Python-Interpreter

```
>>> meat = ["spam", "ham", "beacon"]
>>> sidedish = ["spam", "pasta", "chips"]
>>> print(list(zip(meat,sidedish)))
[('spam', 'spam'), ('ham', 'pasta'), ('beacon', 'chips')]
```

zip (2)

- Besonders nützlich ist `zip`, um mehrere Sequenzen parallel zu durchlaufen:

Python-Interpreter

```
>>> for xyz in zip("ham", "spam", range(5, 10)):
...     (x, y, z) = xyz
...     print(x, y, z)
...
h s 5
a p 6
m a 7
```

- Sind die Eingabesequenzen unterschiedlich lang, ist das Ergebnis so lang wie die kürzeste Eingabe.

reversed

- Die Funktion `reversed` ermöglicht das Durchlaufen einer Sequenz in umgekehrter Richtung.

Python-Interpreter

```
>>> for x in reversed("ham"):
...     print(x)
...
m
a
h
```

Beispiel Iteration (I)

UNI FREIBURG

Fakultätsfunktion

Zu einer positiven ganzen Zahl soll die Fakultät berechnet werden.

$$0! = 1 \qquad (n+1)! = (n+1) \cdot n! \qquad (1)$$

Schritt 1: Bezeichner und Datentypen

Entwickle eine Funktion `fac`, die die Fakultät einer positiven ganzen Zahl berechnet. Eingabe ist

- `n : int` (mit `n >= 0`)

Ausgabe ist ein `int`.

Nächste Schritte

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def fac(
        n : int  # assume n >= 0
        ) -> int:
    # fill in
    return
```

Schritt 3: Beispiele

```
fac(0) == 1
fac(1) == 1
fac(3) == 6
```

Ergebnis


```
def fac(
        n : int
        ) -> int:
    result = 1
    for i in range(1, n+1):
        result = result * i
    return result
```

Beispiel Iteration (II)

UNI FREIBURG

Produkt einer Sequenz

Aus einer Sequenz von Zahlen soll das Produkt der Zahlen berechnet werden.

Schritt 1: Bezeichner und Datentypen

Entwickle eine Funktion `product`, die das Produkt einer Liste von Zahlen berechnet. Eingabe ist

- `xs` *`# sequence`* (dafür gibt es einen Typ, aber umständlich)

Ausgabe ist eine Zahl `numbers.Number`, das Produkt der Elemente der Eingabe.

Zum Typ aller Zahlen

Zur Verwendung muss das Modul `numbers` importiert werden:

```
import numbers
```

Nächste Schritte

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def product(
        xs # sequence of numbers
        ) -> numbers.Number:
    # fill in
    return
```

Schritt 3: Beispiele

```
product([]) == 1
product([42]) == 42
product([3,2,1]) == 3 * product([2,1]) == 6
product([1,-1,1]) == -1
```

Neuer Schritt: Gerüst zur Verarbeitung von Sequenzen

UNI FREIBURG

Falls ein Argument eine Sequenz (Liste, Tupel, String, ...) ist, dann ist es naheliegend, dass diese Sequenz durchlaufen wird.

```
def product(
        xs # sequence of number
        ) -> numbers.Number:
    # fill in
    for x in xs:
        # fill in action for each element
    return
```

Ergebnis


```
def product(
        xs # sequence of number
        ) -> numbers.Number:
    result = 1     # product([])
    for x in xs:
        result = result * x
    return result
```

Zusammenfassung

- Die Datentypen `list`, `tuple` und `string` sind Sequenztypen
- Alle Sequenztypen unterstützen Verkettung, Wiederholung, Indizierung, Test auf Mitgliedschaft, Slicing und Iteration.
- Tupel sind unveränderlich, Listen nicht
- Tuple Unpacking erlaubt das Zerlegen von Sequenzen
- `for-in`-Schleife: Standardmuster zur Verarbeitung von Sequenzen