合肥大學
HEFEI UNIVERSITY

Programming with Python

40. Operationen für Iteratoren

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-10-29

Programming with Python

Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline

Einleitung

Einleitung

- Sequenzen spielen eine sehr wichtige Rolle in der Python-Programmierung.

Einleitung

- Sequenzen spielen eine sehr wichtige Rolle in der Python-Programmierung.
- Der Fakt das Generator-Funktionen und das Schlüsselwort `yield` zur Sprache hinzugefügt wurden, nur um eine natürliche Möglichkeit zum konstruieren komplizierte Sequenzen bereitzustellen, spricht für sich selbst[20].

Einleitung

- Sequenzen spielen eine sehr wichtige Rolle in der Python-Programmierung.
- Der Fakt das Generator-Funktionen und das Schlüsselwort `yield` zur Sprache hinzugefügt wurden, nur um eine natürliche Möglichkeit zum konstruieren komplizierte Sequenzen bereitzustellen, spricht für sich selbst[20].
- Natürlich gibt es auch viele weitere Werkzeuge, um Sequenzen zu bearbeiten und zu transformieren.

Einleitung

- Sequenzen spielen eine sehr wichtige Rolle in der Python-Programmierung.
- Der Fakt das Generator-Funktionen und das Schlüsselwort `yield` zur Sprache hinzugefügt wurden, nur um eine natürliche Möglichkeit zum konstruieren komplizierte Sequenzen bereitzustellen, spricht für sich selbst[20].
- Natürlich gibt es auch viele weitere Werkzeuge, um Sequenzen zu bearbeiten und zu transformieren.
- Einige davon sind direkt in die Sprache eingebaute Funktionen, andere kommen in dem Modul `itertools`[10].

Einleitung

- Sequenzen spielen eine sehr wichtige Rolle in der Python-Programmierung.
- Der Fakt das Generator-Funktionen und das Schlüsselwort `yield` zur Sprache hinzugefügt wurden, nur um eine natürliche Möglichkeit zum konstruieren komplizierte Sequenzen bereitzustellen, spricht für sich selbst[20].
- Natürlich gibt es auch viele weitere Werkzeuge, um Sequenzen zu bearbeiten und zu transformieren.
- Einige davon sind direkt in die Sprache eingebaute Funktionen, andere kommen in dem Modul `itertools`[10].
- Hier wollen wir ein paar von ihnen diskutieren.

filter und takewhile

`filter` und `takewhile`

- Die ersten beiden Funktionen, die wir uns anschauen, sind die built-in Funktion `filter` und die Funktion `takewhile` aus dem Modul `itertools`.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die ersten beiden Funktionen, die wir uns anschauen, sind die built-in Funktion `filter` und die Funktion `takewhile` aus dem Modul `itertools`.
- In der letzten Einheit haben wir in Datei eine Generator-Funktion implementiert, die eine endlose Sequenz von Primzahlen zurückliefert.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die ersten beiden Funktionen, die wir uns anschauen, sind die built-in Funktion `filter` und die Funktion `takewhile` aus dem Modul `itertools`.
- In der letzten Einheit haben wir in Datei eine Generator-Funktion implementiert, die eine endlose Sequenz von Primzahlen zurückliefert.
- Gibt es eine bequeme Möglichkeit, eine Liste aller Primzahlen die kleiner als 50 sind zurückzuliefern?

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y^2 + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y^2+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x^2+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x^2+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die ersten beiden Funktionen, die wir uns anschauen, sind die built-in Funktion `filter` und die Funktion `takewhile` aus dem Modul `itertools`.
- In der letzten Einheit haben wir in Datei eine Generator-Funktion implementiert, die eine endlose Sequenz von Primzahlen zurückliefert.
- Gibt es eine bequeme Möglichkeit, eine Liste aller Primzahlen die kleiner als 50 sind zurückzuliefern?
- Die Antwort ist *Ja*.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die ersten beiden Funktionen, die wir uns anschauen, sind die built-in Funktion `filter` und die Funktion `takewhile` aus dem Modul `itertools`.
- In der letzten Einheit haben wir in Datei eine Generator-Funktion implementiert, die eine endlose Sequenz von Primzahlen zurückliefert.
- Gibt es eine bequeme Möglichkeit, eine Liste aller Primzahlen die kleiner als 50 sind zurückzuliefern?
- Die Antwort ist *Ja*.
- `takewhile` ist eine Function mit zwei Parametern[10].

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- In der letzten Einheit haben wir in Datei eine Generator-Funktion implementiert, die eine endlose Sequenz von Primzahlen zurückliefert.
- Gibt es eine bequeme Möglichkeit, eine Liste aller Primzahlen die kleiner als 50 sind zurückzuliefern?
- Die Antwort ist *Ja*.
- `takewhile` ist eine Function mit zwei Parametern[10].
- Der zweite Parameter ist ein `Iterable`.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Gibt es eine bequeme Möglichkeit, eine Liste aller Primzahlen die kleiner als 50 sind zurückzuliefern?
- Die Antwort ist *Ja*.
- `takewhile` ist eine Function mit zwei Parametern[10].
- Der zweite Parameter ist ein `Iterable`.
- Sagen wir, dass dieses `Iterable` eine Sequenz von Elementen von einem Type `T` bereitstellt.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y^2 + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y^2+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x^2+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x^2+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die Antwort ist *Ja*.
- `takewhile` ist eine Function mit zwei Parametern[10].
- Der zweite Parameter ist ein `Iterable`.
- Sagen wir, dass dieses `Iterable` eine Sequenz von Elementen von einem Type `T` bereitstellt.
- Der erste Parameter ist dann ein Prädikat, also eine Funktion die ein Element vom Typ `T` akzeptiert und einen `bool`-Wert zurückgibt.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- `takewhile` ist eine Function mit zwei Parametern[10].
- Der zweite Parameter ist ein `Iterable`.
- Sagen wir, dass dieses `Iterable` eine Sequenz von Elementen von einem Type `T` bereitstellt.
- Der erste Parameter ist dann ein Prädikat, also eine Funktion die ein Element vom Typ `T` akzeptiert und einen `bool`-Wert zurückgibt.
- Dann konstruiert `takewhile` einen *neuen* `Iterator`, der die Elemente des ursprünglichen `Iterable` zurückliefert *so lange das Prädikat wahr für diese ist*.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y^2 + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y^2+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x^2+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x^2+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Sagen wir, dass dieses `Iterable` eine Sequenz von Elementen von einem Type `T` bereitstellt.
- Der erste Parameter ist dann ein Prädikat, also eine Funktion die ein Element vom Typ `T` akzeptiert und einen `bool`-Wert zurückgibt.
- Dann konstruiert `takewhile` einen *neuen* `Iterator`, der die Elemente des ursprünglichen `Iterable` zurückliefert *so lange das Prädikat wahr für diese ist*.
- Sobald es bei einem Element aus dem originalen `Iterable` ankommt, für das das Prädikat `False` ergibt, wird die Iteration abgebrochen.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Dann konstruiert `takewhile` einen *neuen* `Iterator`, der die Elemente des ursprünglichen `Iterable` zurückliefert *so lange das Prädikat wahr für diese ist*.
- Sobald es bei einem Element aus dem originalen `Iterable` ankommt, für das das Prädikat `False` ergibt, wird die Iteration abgebrochen.
- In Einheit 30 haben wir gelernt, dass wir auch Funktionen oder `lambda`s als Argumente an andere Funktionen übergeben können.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Dann konstruiert `takewhile` einen *neuen* `Iterator`, der die Elemente des ursprünglichen `Iterable` zurückliefert *so lange das Prädikat wahr für diese ist*.
- Sobald es bei einem Element aus dem originalen `Iterable` ankommt, für das das Prädikat `False` ergibt, wird die Iteration abgebrochen.
- In Einheit 30 haben wir gelernt, dass wir auch Funktionen oder `lambda`s als Argumente an andere Funktionen übergeben können.
- Das ist ein praktisches Beispiel, wo `lambda`s besonders nützlich sind.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Sobald es bei einem Element aus dem originalen `Iterable` ankommt, für das das Prädikat `False` ergibt, wird die Iteration abgebrochen.
- In Einheit 30 haben wir gelernt, dass wir auch Funktionen oder `lambda`s als Argumente an andere Funktionen übergeben können.
- Das ist ein praktisches Beispiel, wo `lambda`s besonders nützlich sind.
- Die Antwort auf „Wie bekomme ich alle Primzahlen weniger als 50 aus unsere Primzahlsequenz?“ ist, einfach `takewhile(lambda z: z < 50, primes())` aufzurufen.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- In Einheit 30 haben wir gelernt, dass wir auch Funktionen oder `lambda`s als Argumente an andere Funktionen übergeben können.
- Das ist ein praktisches Beispiel, wo `lambda`s besonders nützlich sind.
- Die Antwort auf „Wie bekomme ich alle Primzahlen weniger als 50 aus unsere Primzahlsequenz?“ ist, einfach `takewhile(lambda z: z < 50, primes())` aufzurufen.
- Diese Sequenz ist nicht mehr endlos land und kann bequem in eine `list` umgewandelt werden.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- In Einheit 30 haben wir gelernt, dass wir auch Funktionen oder `lambda`s als Argumente an andere Funktionen übergeben können.
- Das ist ein praktisches Beispiel, wo `lambda`s besonders nützlich sind.
- Die Antwort auf „Wie bekomme ich alle Primzahlen weniger als 50 aus unsere Primzahlsequenz?“ ist, einfach `takewhile(lambda z: z < 50, primes())` aufzurufen.
- Diese Sequenz ist nicht mehr endlos land und kann bequem in eine `list` umgewandelt werden.
- Die built-in Funktion `filter` funktioniert sehr ähnlich[3].

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Das ist ein praktisches Beispiel, wo `lambda`s besonders nützlich sind.
- Die Antwort auf „Wie bekomme ich alle Primzahlen weniger als 50 aus unsere Primzahlsequenz?“ ist, einfach `takewhile(lambda z: z < 50, primes())` aufzurufen.
- Diese Sequenz ist nicht mehr endlos land und kann bequem in eine `list` umgewandelt werden.
- Die built-in Funktion `filter` funktioniert sehr ähnlich[3].
- Auch sie akzeptiert ein Prädikat und einen `Iterable` als Parameter.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die Antwort auf „Wie bekomme ich alle Primzahlen weniger als 50 aus unsere Primzahlsequenz?“ ist, einfach `takewhile(lambda z: z < 50, primes())` aufzurufen.
- Diese Sequenz ist nicht mehr endlos land und kann bequem in eine `list` umgewandelt werden.
- Die built-in Funktion `filter` funktioniert sehr ähnlich[3].
- Auch sie akzeptiert ein Prädikat und einen `Iterable` als Parameter.
- Anders als `takewhile` bricht der von `filter` erzeugte neue `Iterator` nicht ab, wenn das Prädikat `False` liefert.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Diese Sequenz ist nicht mehr endlos land und kann bequem in eine `list` umgewandelt werden.
- Die built-in Funktion `filter` funktioniert sehr ähnlich[3].
- Auch sie akzeptiert ein Prädikat und einen `Iterable` als Parameter.
- Anders als `takewhile` bricht der von `filter` erzeugte neue `Iterator` nicht ab, wenn das Prädikat `False` liefert.
- Stattdessen liefert er nur die Elemente zurück, für welche das Prädikat `True` liefert.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Die built-in Funktion `filter` funktioniert sehr ähnlich[3].
- Auch sie akzeptiert ein Prädikat und einen `Iterable` als Parameter.
- Anders als `takewhile` bricht der von `filter` erzeugte neue `Iterator` nicht ab, wenn das Prädikat `False` liefert.
- Stattdessen liefert er nur die Elemente zurück, für welche das Prädikat `True` liefert.
- Wir benutzen das, um Primzahlen x zu generieren, für die es eine Ganzzahl y gibt so das $x = y^2 + 1$.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Auch sie akzeptiert ein Prädikat und einen `Iterable` als Parameter.
- Anders als `takewhile` bricht der von `filter` erzeugte neue `Iterator` nicht ab, wenn das Prädikat `False` liefert.
- Stattdessen liefert er nur die Elemente zurück, für welche das Prädikat `True` liefert.
- Wir benutzen das, um Primzahlen x zu generieren, für die es eine Ganzzahl y gibt so das $x = y^2 + 1$.
- Wir implementieren dieses Prädikat ebenfalls als `lambda`.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Stattdessen liefert er nur die Elemente zurück, für welche das Prädikat `True` liefert.
- Wir benutzen das, um Primzahlen x zu generieren, für die es eine Ganzzahl y gibt so das $x = y^2 + 1$.
- Wir implementieren dieses Prädikat ebenfalls als `lambda`.
- Weil der von `primes()` gelieferte `Iterator` eine unendliche Sequenz liefert, benutzen wir wieder `takewhile` und limitieren die Ausgabe auf Primzahlen die kleiner als 1000 sind.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y^2 + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y^2+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x^2+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x^2+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Wir benutzen das, um Primzahlen x zu generieren, für die es eine Ganzzahl y gibt so das $x = y^2 + 1$.
- Wir implementieren dieses Prädikat ebenfalls als `lambda`.
- Weil der von `primes()` gelieferte `Iterator` eine unendliche Sequenz liefert, benutzen wir wieder `takewhile` und limitieren die Ausgabe auf Primzahlen die kleiner als 1000 sind.
- Wir übergeben das Ergebnis unseres Konstrukt an die Funktion `tuple`, die eine unveränderliche und indizierbare Sequenz erstellt.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Wir implementieren dieses Prädikat ebenfalls als `lambda`.
- Weil der von `primes()` gelieferte `Iterator` eine unendliche Sequenz liefert, benutzen wir wieder `takewhile` und limitieren die Ausgabe auf Primzahlen die kleiner als 1000 sind.
- Wir übergeben das Ergebnis unseres Konstrukt an die Funktion `tuple`, die eine unveränderliche und indizierbare Sequenz erstellt.
- Wir finden zehn Primzahlen, die unseren Bedingungen entsprechen.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

`filter` und `takewhile`

- Wir implementieren dieses Prädikat ebenfalls als `lambda`.
- Weil der von `primes()` gelieferte `Iterator` eine unendliche Sequenz liefert, benutzen wir wieder `takewhile` und limitieren die Ausgabe auf Primzahlen die kleiner als 1000 sind.
- Wir übergeben das Ergebnis unseres Konstrukt an die Funktion `tuple`, die eine unveränderliche und indizierbare Sequenz erstellt.
- Wir finden zehn Primzahlen, die unseren Bedingungen entsprechen.
- Die kleinste ist $1^2 + 1 = 2$ und die größte ist $26^2 + 1 = 677$.

```python
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes()))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
    ↪ 43, 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197,
    ↪ 257, 401, 577, 677)
```

map

map

- Eine weitere wichtige Funktion für Sequenzen ist die built-in Funktion `map`[3].

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Eine weitere wichtige Funktion für Sequenzen ist die built-in Funktion `map`[3].
- Wir probieren sie in `map.py` aus.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Eine weitere wichtige Funktion für Sequenzen ist die built-in Funktion `map`[3].
- Wir probieren sie in `map.py` aus.
- In Einheit 38 haben wir einen Generator-Ausdruck verwendet, um Daten aus einem String im comma-separated values (CSV)-Format zu ziehen.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Eine weitere wichtige Funktion für Sequenzen ist die built-in Funktion `map`[3].
- Wir probieren sie in `map.py` aus.
- In Einheit 38 haben wir einen Generator-Ausdruck verwendet, um Daten aus einem String im comma-separated values (CSV)-Format zu ziehen.
- Anstatt `int(s) for s in csv_text.split(",")` können wir einfach `map(int, csv_text.split(","))` schreiben.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir probieren sie in `map.py` aus.
- In Einheit 38 haben wir einen Generator-Ausdruck verwendet, um Daten aus einem String im comma-separated values (CSV)-Format zu ziehen.
- Anstatt `int(s) for s in csv_text.split(",")` können wir einfach `map(int, csv_text.split(","))` schreiben.
- Das erste Argument von `map` ist eine Funktion, die auf die Elemente des `Iterable` angewandt wird, der als zweites Argument bereitgestellt wird.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Anstatt `int(s) for s in csv_text.split(",")` können wir einfach `map(int, csv_text.split(","))` schreiben.
- Das erste Argument von `map` ist eine Funktion, die auf die Elemente des `Iterable` angewandt wird, der als zweites Argument bereitgestellt wird.
- Das Ergebnis von `map` ist dann ein neuer `Iterator` mit den Rückgabewerten der Funktion.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Anstatt `int(s) for s in csv_text.split(",")` können wir einfach `map(int, csv_text.split(","))` schreiben.
- Das erste Argument von `map` ist eine Funktion, die auf die Elemente des `Iterable` angewandt wird, der als zweites Argument bereitgestellt wird.
- Das Ergebnis von `map` ist dann ein neuer `Iterator` mit den Rückgabewerten der Funktion.
- In `map.py`, zerteilen wir den `csv_text` basierend auf dem Separator `","`.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Das erste Argument von `map` ist eine Funktion, die auf die Elemente des `Iterable` angewandt wird, der als zweites Argument bereitgestellt wird.
- Das Ergebnis von `map` ist dann ein neuer `Iterator` mit den Rückgabewerten der Funktion.
- In `map.py`, zerteilen wir den `csv_text` basierend auf dem Separator `","`.
- Dann übersetzen wir die Elemente der resultierenden Liste zu `ints` via `map`.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Das erste Argument von `map` ist eine Funktion, die auf die Elemente des `Iterable` angewandt wird, der als zweites Argument bereitgestellt wird.
- Das Ergebnis von `map` ist dann ein neuer `Iterator` mit den Rückgabewerten der Funktion.
- In `map.py`, zerteilen wir den `csv_text` basierend auf dem Separator `","`.
- Dann übersetzen wir die Elemente der resultierenden Liste zu `ints` via `map`.
- Zuletzt filtern wir die Sequenz und behalten nur die Werte, die größer als 20 sind.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Das Ergebnis von `map` ist dann ein neuer `Iterator` mit den Rückgabewerten der Funktion.
- In `map.py`, zerteilen wir den `csv_text` basierend auf dem Separator `","`.
- Dann übersetzen wir die Elemente der resultierenden Liste zu `ints` via `map`.
- Zuletzt filtern wir die Sequenz und behalten nur die Werte, die größer als 20 sind.
- Über die gefilterte und gemappte Sequenz können wir dann bequem mit einer `for`-Schleife iterieren.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- In `map.py`, zerteilen wir den `csv_text` basierend auf dem Separator `","`.
- Dann übersetzen wir die Elemente der resultierenden Liste zu `ints` via `map`.
- Zuletzt filtern wir die Sequenz und behalten nur die Werte, die größer als 20 sind.
- Über die gefilterte und gemappte Sequenz können wir dann bequem mit einer `for`-Schleife iterieren.
- Wir wäre es nun, wenn wir alle Quadratzahlen der Werte aus den CSV-Daten laden wollen, aber jede Zahl nur einmal?

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Dann übersetzen wir die Elemente der resultierenden Liste zu `ints` via `map`.
- Zuletzt filtern wir die Sequenz und behalten nur die Werte, die größer als 20 sind.
- Über die gefilterte und gemappte Sequenz können wir dann bequem mit einer `for`-Schleife iterieren.
- Wir wäre es nun, wenn wir alle Quadratzahlen der Werte aus den CSV-Daten laden wollen, aber jede Zahl nur einmal?
- Wir würden also alle Duplikate löschen.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Zuletzt filtern wir die Sequenz und behalten nur die Werte, die größer als 20 sind.
- Über die gefilterte und gemappte Sequenz können wir dann bequem mit einer `for`-Schleife iterieren.
- Wir wäre es nun, wenn wir alle Quadratzahlen der Werte aus den CSV-Daten laden wollen, aber jede Zahl nur einmal?
- Wir würden also alle Duplikate löschen.
- Wir benutzen dafür wieder `split` um den Text in durch `","` getrennte Stücke zu zerteilen.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir wäre es nun, wenn wir alle Quadratzahlen der Werte aus den CSV-Daten laden wollen, aber jede Zahl nur einmal?
- Wir würden also alle Duplikate löschen.
- Wir benutzen dafür wieder `split` um den Text in durch `","` getrennte Stücke zu zerteilen.
- Wir übersetzen die Stücke zu `int` und berechnen ihre Quadrate mit der `map`-Funktion, stellen diesmal aber ein `lambda` zur Verfügung, das die Transformation in einem Schritt durchführt.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir würden also alle Duplikate löschen.
- Wir benutzen dafür wieder `split` um den Text in durch `","` getrennte Stücke zu zerteilen.
- Wir übersetzen die Stücke zu `int` und berechnen ihre Quadrate mit der `map`-Funktion, stellen diesmal aber ein `lambda` zur Verfügung, das die Transformation in einem Schritt durchführt.
- Nun wollen wir eine duplikatfrei Kollektion dieser Daten bekommen.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir würden also alle Duplikate löschen.
- Wir benutzen dafür wieder `split` um den Text in durch `","` getrennte Stücke zu zerteilen.
- Wir übersetzen die Stücke zu `int` und berechnen ihre Quadrate mit der `map`-Funktion, stellen diesmal aber ein `lambda` zur Verfügung, das die Transformation in einem Schritt durchführt.
- Nun wollen wir eine duplikatfrei Kollektion dieser Daten bekommen.
- Das geht, in dem wir den `Iterator`, den `map` liefert, einfach an den `set`-Konstruktor übergeben.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir benutzen dafür wieder `split` um den Text in durch `","` getrennte Stücke zu zerteilen.
- Wir übersetzen die Stücke zu `int` und berechnen ihre Quadrate mit der `map`-Funktion, stellen diesmal aber ein `lambda` zur Verfügung, das die Transformation in einem Schritt durchführt.
- Nun wollen wir eine duplikatfrei Kollektion dieser Daten bekommen.
- Das geht, in dem wir den `Iterator`, den `map` liefert, einfach an den `set`-Konstruktor übergeben.
- Eine Menge ist per definition duplikatfrei.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir übersetzen die Stücke zu `int` und berechnen ihre Quadrate mit der `map`-Funktion, stellen diesmal aber ein `lambda` zur Verfügung, das die Transformation in einem Schritt durchführt.
- Nun wollen wir eine duplikatfrei Kollektion dieser Daten bekommen.
- Das geht, in dem wir den `Iterator`, den `map` liefert, einfach an den `set`-Konstruktor übergeben.
- Eine Menge ist per definition duplikatfrei.
- In der Ausgabe sehen wir, dass die `9` tatsächlich nur einmal auftaucht, genau wie die `144`.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ `python3 map.py` ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Nun wollen wir eine duplikatfrei Kollektion dieser Daten bekommen.
- Das geht, in dem wir den `Iterator`, den `map` liefert, einfach an den `set`-Konstruktor übergeben.
- Eine Menge ist per definition duplikatfrei.
- In der Ausgabe sehen wir, dass die 9 tatsächlich nur einmal auftaucht, genau wie die 144.
- Die Function `map` arbeitet auch gut mit Aggregatfunktionen wir `sum`, `min`, oder `max` zusammen.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Das geht, in dem wir den `Iterator`, den `map` liefert, einfach an den `set`-Konstruktor übergeben.
- Eine Menge ist per definition duplikatfrei.
- In der Ausgabe sehen wir, dass die `9` tatsächlich nur einmal auftaucht, genau wie die `144`.
- Die Function `map` arbeitet auch gut mit Aggregatfunktionen wir `sum`, `min`, oder `max` zusammen.
- Im letzten Beispiel haben wir eine Liste `words` von Worten und wollen die Länge des längsten Wortes bestimmen.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Eine Menge ist per definition duplikatfrei.
- In der Ausgabe sehen wir, dass die 9 tatsächlich nur einmal auftaucht, genau wie die 144.
- Die Function `map` arbeitet auch gut mit Aggregatfunktionen wir `sum`, `min`, oder `max` zusammen.
- Im letzten Beispiel haben wir eine Liste `words` von Worten und wollen die Länge des längsten Wortes bestimmen.
- Dazu mappen wir zuerst jedes Wort zu seiner Länge via `map(len, words)`.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- In der Ausgabe sehen wir, dass die `9` tatsächlich nur einmal auftaucht, genau wie die `144`.
- Die Function `map` arbeitet auch gut mit Aggregatfunktionen wir `sum`, `min`, oder `max` zusammen.
- Im letzten Beispiel haben wir eine Liste `words` von Worten und wollen die Länge des längsten Wortes bestimmen.
- Dazu mappen wir zuerst jedes Wort zu seiner Länge via `map(len, words)`.
- Wir bekommen einen `Iterator` der Wortlängen, den wir direkt an `max` übergeben können.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Die Function `map` arbeitet auch gut mit Aggregatfunktionen wir `sum`, `min`, oder `max` zusammen.
- Im letzten Beispiel haben wir eine Liste `words` von Worten und wollen die Länge des längsten Wortes bestimmen.
- Dazu mappen wir zuerst jedes Wort zu seiner Länge via `map(len, words)`.
- Wir bekommen einen `Iterator` der Wortlängen, den wir direkt an `max` übergeben können.
- `max` iteriert dann über diese Sequenz und liefert deren größten Wert.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Im letzten Beispiel haben wir eine Liste `words` von Worten und wollen die Länge des längsten Wortes bestimmen.
- Dazu mappen wir zuerst jedes Wort zu seiner Länge via `map(len, words)`.
- Wir bekommen einen `Iterator` der Wortlängen, den wir direkt an `max` übergeben können.
- `max` iteriert dann über diese Sequenz und liefert deren größten Wert.
- Beachten Sie, dass `map` keine Datenstruktur mit all den transformierten Werten im Speicher erzeugt.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- Wir bekommen einen `Iterator` der Wortlängen, den wir direkt an `max` übergeben können.
- `max` iteriert dann über diese Sequenz und liefert deren größten Wert.
- Beachten Sie, dass `map` keine Datenstruktur mit all den transformierten Werten im Speicher erzeugt.
- Stattdessen wird jedes Element dann erzeugt, wenn es gebraucht wird (und durch die Garbage Collection freigegeben, wenn es nicht mehr gebraucht wird).

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

map

- `max` iteriert dann über diese Sequenz und liefert deren größten Wert.
- Beachten Sie, dass `map` keine Datenstruktur mit all den transformierten Werten im Speicher erzeugt.
- Stattdessen wird jedes Element dann erzeugt, wenn es gebraucht wird (und durch die Garbage Collection freigegeben, wenn es nicht mehr gebraucht wird).
- `map` ist daher eine elegante und effiziente Methode, um Daten zu transformieren.

```python
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ python3 map.py ↓

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

zip

zip

- Als letztes Beispiel für Sequenz-Verarbeitung spielen wir ein wenig mit der Funktion `zip`[3].

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Als letztes Beispiel für Sequenz-Verarbeitung spielen wir ein wenig mit der Funktion `zip`[3].
- Diese Funktion akzeptiert mehrere `Iterables` als Argumente und liefert einen `Iterator` als Ergebnis, der alle Iterables synchron durchiteriert und Tupel mit jeweils einem Wert von allen zurückliefert.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Als letztes Beispiel für Sequenz-Verarbeitung spielen wir ein wenig mit der Funktion `zip`[3].
- Diese Funktion akzeptiert mehrere `Iterables` als Argumente und liefert einen `Iterator` als Ergebnis, der alle Iterables synchron durchiteriert und Tupel mit jeweils einem Wert von allen zurückliefert.
- Zum Bespiel liefert `zip([1, 2, 3], ["a", "b", "c"])` einen `Iterator` der die Sequenz `(1, "a")`, `(2, "b")`, und `(3, "c")` produziert.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Als letztes Beispiel für Sequenz-Verarbeitung spielen wir ein wenig mit der Funktion `zip`[3].
- Diese Funktion akzeptiert mehrere `Iterables` als Argumente und liefert einen `Iterator` als Ergebnis, der alle Iterables synchron durchiteriert und Tupel mit jeweils einem Wert von allen zurückliefert.
- Zum Bespiel liefert `zip([1, 2, 3], ["a", "b", "c"])` einen `Iterator` der die Sequenz `(1, "a")`, `(2, "b")`, und `(3, "c")` produziert.
- Manchmal haben die `Iterables` verschiedene Längen.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Diese Funktion akzeptiert mehrere `Iterables` als Argumente und liefert einen `Iterator` als Ergebnis, der alle Iterables synchron durchiteriert und Tupel mit jeweils einem Wert von allen zurückliefert.
- Zum Bespiel liefert `zip([1, 2, 3], ["a", "b", "c"])` einen `Iterator` der die Sequenz `(1, "a")`, `(2, "b")`, und `(3, "c")` produziert.
- Manchmal haben die `Iterables` verschiedene Längen.
- Um sicherzustellen, dass so ein Fehler einen `ValueError` auslöst, müssen wir immer den Parameter `strict=True` angeben[2].

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Zum Bespiel liefert `zip([1, 2, 3], ["a", "b", "c"])` einen `Iterator` der die Sequenz `(1, "a")`, `(2, "b")`, und `(3, "c")` produziert.
- Manchmal haben die `Iterables` verschiedene Längen.
- Um sicherzustellen, dass so ein Fehler einen `ValueError` auslöst, müssen wir immer den Parameter `strict=True` angeben[2].
- In `zip.py` benutzen wir `zip` um eine Funktion `distance` zu implementieren, die den Euklidischen Abstand von zwei n-dimensionalen Vektoren oder Punkten `p1` und `p2` berechnet.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Manchmal haben die `Iterables` verschiedene Längen.
- Um sicherzustellen, dass so ein Fehler einen `ValueError` auslöst, müssen wir immer den Parameter `strict=True` angeben[2].
- In `zip.py` benutzen wir `zip` um eine Funktion `distance` zu implementieren, die den Euklidischen Abstand von zwei n-dimensionalen Vektoren oder Punkten `p1` und `p2` berechnet.
- Die beiden Punkte werden als `Iterables` von entweder `float` oder `int` als Parameter bereitgestellt.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Um sicherzustellen, dass so ein Fehler einen `ValueError` auslöst, müssen wir immer den Parameter `strict=True` angeben[2].
- In `zip.py` benutzen wir `zip` um eine Funktion `distance` zu implementieren, die den Euklidischen Abstand von zwei n-dimensionalen Vektoren oder Punkten `p1` und `p2` berechnet.
- Die beiden Punkte werden als `Iterables` von entweder `float` oder `int` als Parameter bereitgestellt.
- Wir könnten sie also z. B. als `listss` angeben.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- In `zip.py` benutzen wir `zip` um eine Funktion `distance` zu implementieren, die den Euklidischen Abstand von zwei n-dimensionalen Vektoren oder Punkten `p1` und `p2` berechnet.
- Die beiden Punkte werden als `Iterables` von entweder `float` oder `int` als Parameter bereitgestellt.
- Wir könnten sie also z. B. als `listss` angeben.
- Der Euklidische Abstand ist definiert als

$$\texttt{distance(p1, p2)} = \sqrt{\sum_{i=1}^{n}(\texttt{p1}_i - \texttt{p2}_i)^2}$$

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Die beiden Punkte werden als `Iterables` von entweder `float` oder `int` als Parameter bereitgestellt.
- Wir könnten sie also z. B. als `listss` angeben.
- Der Euklidische Abstand ist definiert als

$$\texttt{distance(p1, p2)} = \sqrt{\sum_{i=1}^{n} (\texttt{p1}_i - \texttt{p2}_i)^2}$$

- Wir müssen also über die Elemente beider Vektoren synchron iterieren.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Die beiden Punkte werden als `Iterables` von entweder `float` oder `int` als Parameter bereitgestellt.
- Wir könnten sie also z. B. als `listss` angeben.
- Der Euklidische Abstand ist definiert als

$$\texttt{distance(p1, p2)} = \sqrt{\sum_{i=1}^{n} (\texttt{p1}_i - \texttt{p2}_i)^2}$$

- Wir müssen also über die Elemente beider Vektoren synchron iterieren.
- Das ist genau das, was `zip` macht.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Wir müssen also über die Elemente beider Vektoren synchron iterieren.
- Das ist genau das, was `zip` macht.
- Wenn beide Punkte z. B. als `lists` angegeben werden, dann wird `zip(p1, p2, strict=True)` uns, Schritt für Schritt, die Tupel `(p1[0], p2[0])`, `(p1[1], p2[1])`, ..., angeben, bis es an Ende der Listen kommt.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Wir müssen also über die Elemente beider Vektoren synchron iterieren.
- Das ist genau das, was `zip` macht.
- Wenn beide Punkte z. B. als `lists` angegeben werden, dann wird `zip(p1, p2, strict=True)` uns, Schritt für Schritt, die Tupel `(p1[0], p2[0])`, `(p1[1], p2[1])`, ..., angeben, bis es an Ende der Listen kommt.
- Wir können nun den Generator-Ausdruck `(a - b)** 2 for a, b in zip(p1, p2, strict=True)` schreiben.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Das ist genau das, was `zip` macht.
- Wenn beide Punkte z. B. als `lists` angegeben werden, dann wird `zip(p1, p2, strict=True)` uns, Schritt für Schritt, die Tupel `(p1[0], p2[0])`, `(p1[1], p2[1])`, ..., angeben, bis es an Ende der Listen kommt.
- Wir können nun den Generator-Ausdruck `(a - b)** 2 for a, b in zip(p1, p2, strict=True)` schreiben.
- Er benutzt das Auspacken der Tupel um die beiden Elemente `a` und `b` aus den Tupeln zu holen, die `zip` erstellt.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Wir können nun den Generator-Ausdruck `(a - b)** 2 for a, b in zip(p1, p2, strict=True)` schreiben.
- Er benutzt das Auspacken der Tupel um die beiden Elemente `a` und `b` aus den Tupeln zu holen, die `zip` erstellt.
- Er berechnet dann das Quadrat der Differenz dieser zwei Elemente.

```
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Wir können nun den Generator-Ausdruck `(a - b)** 2 for a, b in zip(p1, p2, strict=True)` schreiben.
- Er benutzt das Auspacken der Tupel um die beiden Elemente `a` und `b` aus den Tupeln zu holen, die `zip` erstellt.
- Er berechnet dann das Quadrat der Differenz dieser zwei Elemente.
- In dem wir diesen Generator-Ausdruck als Parameter an die Funktion `sum` übergeben, bekommen wir die Summe deiser Quadrate.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Er benutzt das Auspacken der Tupel um die beiden Elemente `a` und `b` aus den Tupeln zu holen, die `zip` erstellt.
- Er berechnet dann das Quadrat der Differenz dieser zwei Elemente.
- In dem wir diesen Generator-Ausdruck als Parameter an die Funktion `sum` übergeben, bekommen wir die Summe deiser Quadrate.
- Schlussendlich benutzen wir die `sqrt`-Funktion aus dem Modul `math` um die Berechnung des Euklidischen Abstands zu komplettieren.

```
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Er berechnet dann das Quadrat der Differenz dieser zwei Elemente.
- In dem wir diesen Generator-Ausdruck als Parameter an die Funktion `sum` übergeben, bekommen wir die Summe deiser Quadrate.
- Schlussendlich benutzen wir die `sqrt`-Funktion aus dem Modul `math` um die Berechnung des Euklidischen Abstands zu komplettieren.
- Anstatt diese neue Funktion mit ein paar Beispielen manuell zu testen, machen wir das lieber mit einem Doctest.

```
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- In dem wir diesen Generator-Ausdruck als Parameter an die Funktion `sum` übergeben, bekommen wir die Summe deiser Quadrate.
- Schlussendlich benutzen wir die `sqrt`-Funktion aus dem Modul `math` um die Berechnung des Euklidischen Abstands zu komplettieren.
- Anstatt diese neue Funktion mit ein paar Beispielen manuell zu testen, machen wir das lieber mit einem Doctest.
- Der Doctest zeigt dass der Abstand von zwei identischen Vektoren mit den selben Werten `[1, 1]` unbedingt `0.0` sein muss.

```
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Schlussendlich benutzen wir die `sqrt`-Funktion aus dem Modul `math` um die Berechnung des Euklidischen Abstands zu komplettieren.
- Anstatt diese neue Funktion mit ein paar Beispielen manuell zu testen, machen wir das lieber mit einem Doctest.
- Der Doctest zeigt dass der Abstand von zwei identischen Vektoren mit den selben Werten `[1, 1]` unbedingt `0.0` sein muss.
- `distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))`, welcher im Grunde $\sqrt{1+1+1+1}$ ist, soll `2.0` sein.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Anstatt diese neue Funktion mit ein paar Beispielen manuell zu testen, machen wir das lieber mit einem Doctest.
- Der Doctest zeigt dass der Abstand von zwei identischen Vektoren mit den selben Werten `[1, 1]` unbedingt `0.0` sein muss.
- `distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))`, welcher im Grunde $\sqrt{1+1+1+1}$ ist, soll `2.0` sein.
- Der Abstand zweier eindimensionaler Vektoren `[100]` und `[10]` soll `90.0` betragen.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

- Der Doctest zeigt dass der Abstand von zwei identischen Vektoren mit den selben Werten `[1, 1]` unbedingt `0.0` sein muss.
- `distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))`, welcher im Grunde $\sqrt{1+1+1+1}$ ist, soll `2.0` sein.
- Der Abstand zweier eindimensionaler Vektoren `[100]` und `[10]` soll `90.0` betragen.
- Geben wir dagegen zwei Vektoren verschiedener Länge an, dann soll das zu der Ausnahme `ValueError` führen.

```python
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

zip

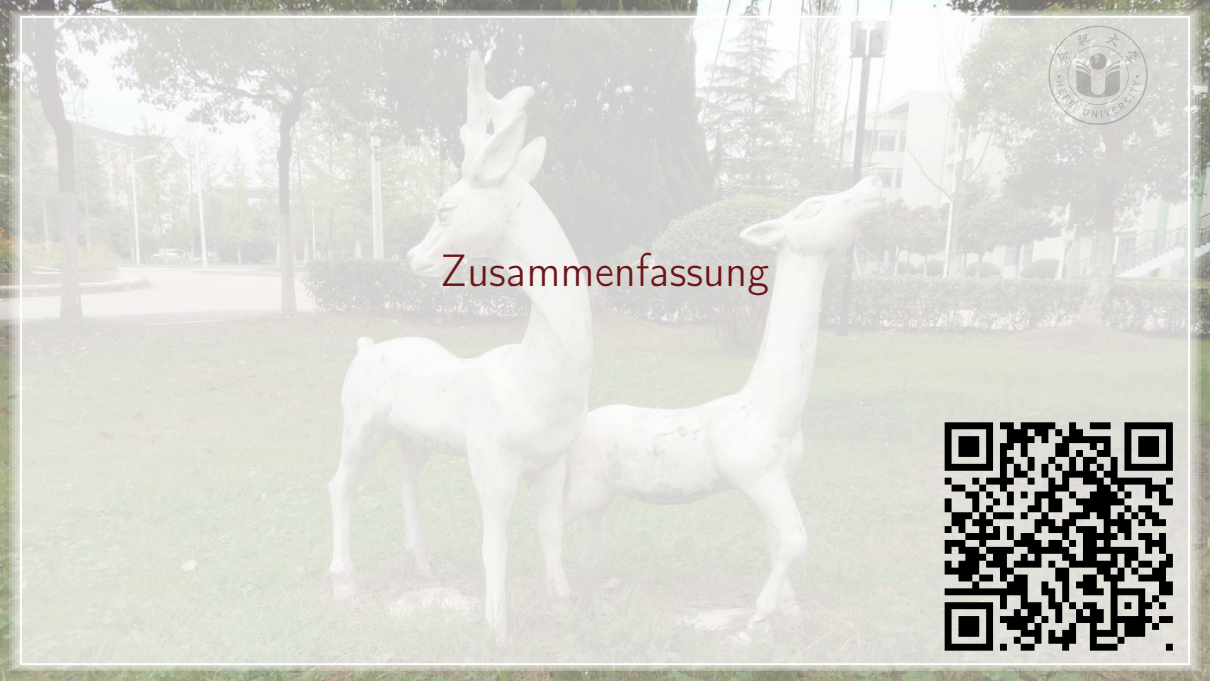
- `distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))`, welcher im Grunde $\sqrt{1+1+1+1}$ ist, soll `2.0` sein.
- Der Abstand zweier eindimensionaler Vektoren `[100]` und `[10]` soll `90.0` betragen.
- Geben wir dagegen zwei Vektoren verschiedener Länge an, dann soll das zu der Ausnahme `ValueError` führen.
- Wir sehen am Output von pytest, dass unsere Funktion diese Tests besteht.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules zip.py
========================= test session starts
   ↪ ==========================
collected 1 item

zip.py .                                                          [100%]

========================== 1 passed in 0.02s
   ↪ ===========================
# pytest 8.4.2 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Zusammenfassung



Zusammenfassung: Operationen auf Iteratoren

- Damit sind wir am Ende unseres kurzen Ausflugs zu den Operationen für `Iterators`.

Zusammenfassung: Operationen auf Iteratoren

- Damit sind wir am Ende unseres kurzen Ausflugs zu den Operationen für `Iterators`.
- Wir können hier wirklich nur ganz wenige Funktionen angucken, und diese auch nur oberflächlich.

Zusammenfassung: Operationen auf Iteratoren

- Damit sind wir am Ende unseres kurzen Ausflugs zu den Operationen für `Iterators`.
- Wir können hier wirklich nur ganz wenige Funktionen angucken, und diese auch nur oberflächlich.
- Das Modul `itertools`[10], welches zu Python gehört, bietet noch viele weitere nützliche Funktionen.

Zusammenfassung: Operationen auf Iteratoren

- Damit sind wir am Ende unseres kurzen Ausflugs zu den Operationen für `Iterators`.
- Wir können hier wirklich nur ganz wenige Funktionen angucken, und diese auch nur oberflächlich.
- Das Modul `itertools`[10], welches zu Python gehört, bietet noch viele weitere nützliche Funktionen.
- Ein Verständnis der Funktionen `map`, `filter`, und `zip` wird Ihnen aber sicherlich nützlich seien, wenn sie weitere Funktionen selbst erforschen.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.
- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.
- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.
- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.
- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.
- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.
- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.
- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.
- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.
- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.
- Mit deren `yield`-Statement erlauben ist uns Funktionen zu definieren, die Berechnungen durchführen und Ergebnisse als Ausgabe liefern, die dann vom aufrufenden Kode verarbeitet werden.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.
- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.
- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.
- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.
- Mit deren `yield`-Statement erlauben ist uns Funktionen zu definieren, die Berechnungen durchführen und Ergebnisse als Ausgabe liefern, die dann vom aufrufenden Kode verarbeitet werden.
- Anders als normale Funktionen wird ihre Ausführung danach fortgesetzt, bis sie weitere Ergebnisse mit `yield` zurückliefern oder das Ende der Sequenz erreicht ist.

Zusammenfassung: Sequenzen

- Arbeiten mit Sequenzen ist ein sehr wichtiger Aspekt vom Python-Programming.
- Die Programmiersprache stellt eine simplifizierte Syntax zum Arbeiten mit Schleifen in Form von Listen-, Mengen-, und Dictionary Comprehension zur Verfügung.
- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.
- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.
- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.
- Mit deren `yield`-Statement erlauben ist uns Funktionen zu definieren, die Berechnungen durchführen und Ergebnisse als Ausgabe liefern, die dann vom aufrufenden Kode verarbeitet werden.
- Anders als normale Funktionen wird ihre Ausführung danach fortgesetzt, bis sie weitere Ergebnisse mit `yield` zurückliefern oder das Ende der Sequenz erreicht ist.
- Sequenzen von Daten können mit vielen Python-Funktionen verarbeitet, transformiert, oder aggregiert werden.

Zusammenfassung: Sequenzen

- Anders als Comprehension erlauben uns Generator-Ausdrücke, Sequenzen von Daten zu definieren, deren Elemente eins nach dem Anderen verarbeitet werden, ohne diese vorher erst alle im Speicher zu manifestieren.
- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.
- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.
- Mit deren `yield`-Statement erlauben ist uns Funktionen zu definieren, die Berechnungen durchführen und Ergebnisse als Ausgabe liefern, die dann vom aufrufenden Kode verarbeitet werden.
- Anders als normale Funktionen wird ihre Ausführung danach fortgesetzt, bis sie weitere Ergebnisse mit `yield` zurückliefern oder das Ende der Sequenz erreicht ist.
- Sequenzen von Daten können mit vielen Python-Funktionen verarbeitet, transformiert, oder aggregiert werden.
- Solche Funktionen können auf Kontainer, Comprehensions, oder Generatoren angewendet werden.

Zusammenfassung: Sequenzen

- Stattdessen werden die Elemente erst erzeugt, wenn sie gebraucht werden.
- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.
- Mit deren `yield`-Statement erlauben ist uns Funktionen zu definieren, die Berechnungen durchführen und Ergebnisse als Ausgabe liefern, die dann vom aufrufenden Kode verarbeitet werden.
- Anders als normale Funktionen wird ihre Ausführung danach fortgesetzt, bis sie weitere Ergebnisse mit `yield` zurückliefern oder das Ende der Sequenz erreicht ist.
- Sequenzen von Daten können mit vielen Python-Funktionen verarbeitet, transformiert, oder aggregiert werden.
- Solche Funktionen können auf Kontainer, Comprehensions, oder Generatoren angewendet werden.
- Das ist möglich, weil die gesamte Application Programming Interface (API) zur Verarbeitung von Sequenzen letztendlich auf zwei Grundlegenden Komponenten baisert: `Iterator` und `Iterable`.

Zusammenfassung: Sequenzen

- Wenn das Erstellen der Elemente komplexer ist, als wir mit Generator-Ausdrücken, nunja, ausdrücken können, dann können wir Generator-Funktionen verwenden.
- Mit deren `yield`-Statement erlauben ist uns Funktionen zu definieren, die Berechnungen durchführen und Ergebnisse als Ausgabe liefern, die dann vom aufrufenden Kode verarbeitet werden.
- Anders als normale Funktionen wird ihre Ausführung danach fortgesetzt, bis sie weitere Ergebnisse mit `yield` zurückliefern oder das Ende der Sequenz erreicht ist.
- Sequenzen von Daten können mit vielen Python-Funktionen verarbeitet, transformiert, oder aggregiert werden.
- Solche Funktionen können auf Kontainer, Comprehensions, oder Generatoren angewendet werden.
- Das ist möglich, weil die gesamte Application Programming Interface (API) zur Verarbeitung von Sequenzen letztendlich auf zwei Grundlegenden Komponenten baisert: `Iterator` und `Iterable`.
- An `Iterable` ist ein Interface das von allen Objekten bereitgestellt wird, deren Elemente eins nach dem Anderen ausgelesen werden können.

Zusammenfassung: Sequenzen

- Anders als normale Funktionen wird ihre Ausführung danach fortgesetzt, bis sie weitere Ergebnisse mit `yield` zurückliefern oder das Ende der Sequenz erreicht ist.
- Sequenzen von Daten können mit vielen Python-Funktionen verarbeitet, transformiert, oder aggregiert werden.
- Solche Funktionen können auf Kontainer, Comprehensions, oder Generatoren angewendet werden.
- Das ist möglich, weil die gesamte Application Programming Interface (API) zur Verarbeitung von Sequenzen letztendlich auf zwei Grundlegenden Komponenten baisert: `Iterator` und `Iterable`.
- An `Iterable` ist ein Interface das von allen Objekten bereitgestellt wird, deren Elemente eins nach dem Anderen ausgelesen werden können.
- Ein `Iterator` ist genau eine solche iterative Aufzählung, eine solcher sequentieller Zugriff auf die Elemente.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 107).

[2] Brandt Bucher. *Add Optional Length-Checking To* `zip`. Python Enhancement Proposal (PEP) 618. Beaverton, OR, USA: Python Software Foundation (PSF), 1. Mai 2020. URL: https://peps.python.org/pep-0618 (besucht am 2024-11-09) (siehe S. 63–70).

[3] "Built-in Functions". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/functions.html (besucht am 2024-12-09) (siehe S. 11–29, 36–39, 63–66).

[4] "`csv` – CSV File Reading and Writing". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/csv.html (besucht am 2024-11-14) (siehe S. 107).

[5] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (siehe S. 107).

[6] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/doctest.html (besucht am 2024-11-07) (siehe S. 107).

[7] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: https://peps.python.org/pep-0257 (besucht am 2024-07-27) (siehe S. 107).

[8] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: https://www.ibm.com/topics/api (besucht am 2024-12-12) (siehe S. 107).

[9] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 107).

[10] "`itertools` – Functions Creating `Iterator`S for Efficient Looping". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/itertools.html (besucht am 2024-11-09) (siehe S. 5–9, 11–19, 87–90).

References II

[11] Holger Krekel und pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Kap. 2.8, S. 65–69. URL: https://docs.pytest.org/en/stable/how-to/doctest.html (besucht am 2024-11-07) (siehe S. 107).

[12] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: https://readthedocs.org/projects/pytest/downloads/pdf/latest (besucht am 2024-11-07) (siehe S. 107).

[13] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 107).

[14] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 107).

[15] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (siehe S. 107).

[16] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (siehe S. 107).

[17] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: https://www.researchgate.net/publication/255673967 (besucht am 2025-09-05) (siehe S. 107).

[18] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: 978-1-4842-7854-3 (siehe S. 107).

[19] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (siehe S. 107).

[20] Neil Schemenauer, Tim Peters und Magnus Lie Hetland. *Simple `GeneratorS`*. Python Enhancement Proposal (PEP) 255. Beaverton, OR, USA: Python Software Foundation (PSF), 18. Mai 2001. URL: https://peps.python.org/pep-0255 (besucht am 2024-11-08) (siehe S. 5–9).

References III

[21] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments (RFC) 4180. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Okt. 2005. URL: https://www.ietf.org/rfc/rfc4180.txt (besucht am 2025-02-05) (siehe S. 107).

[22] *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library (besucht am 2025-04-27).

[23] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: https://www.researchgate.net/publication/220094077 (besucht am 2024-10-01) (siehe S. 107).

[24] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: https://peps.python.org/pep-0008 (besucht am 2024-07-27) (siehe S. 107).

[25] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 107).

[26] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 107).

Glossary (in English) I

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another[8].

CSV *Comma-Separated Values* is a very common and simple text format for exchanging tabular or matrix data[21]. Each row in the text file represents one row in the table or matrix. The elements in the row are separated by a fixed delimiter, usually a comma („,“), sometimes a semicolon („;“). Python offers some out-of-the-box CSV support in the `csv` module[4].

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[7]. They must be delimited by `"""..."""`[7,24].

doctest *doctests* are unit tests in the form of as small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by `....` These snippets can be executed by modules like `doctest`[6] or tools such as pytest[11]. Their output is the compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

pytest is a framework for writing and executing unit tests in Python[5,12,16,18,26]. Learn more at `https://pytest.org`.

Python The Python programming language[9,13,14,25], i.e., what you will learn about in our book[25]. Learn more at `https://python.org`.

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification[1,15,17–19,23]. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.