合肥大學
HEFEI UNIVERSITY

Programming with Python

38. Generator-Ausdrücke

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn · https://thomasweise.github.io

School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-11-09

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



1. Einleitung
2. Generator-Ausdrücke
3. Beispiele zur Funktionsweise
4. Use Cases
5. Zusammenfassung

Einleitung

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.
- Eine List kann vergrößert werden, in dem wir Elemente hinzufügen, aber sie wird immer als kontinuierliches Stück Speicher gemanaged.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.
- Eine List kann vergrößert werden, in dem wir Elemente hinzufügen, aber sie wird immer als kontinuierliches Stück Speicher gemanaged.
- Sie kann nicht nur als „Teil“ im Speicher existieren ... sie existiert immer als „Ganzes“.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.
- Eine List kann vergrößert werden, in dem wir Elemente hinzufügen, aber sie wird immer als kontinuierliches Stück Speicher gemanaged.
- Sie kann nicht nur als „Teil“ im Speicher existieren ... sie existiert immer als „Ganzes“.
- Und das ist, was wir normalerweise auch wollen.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.
- Eine List kann vergrößert werden, in dem wir Elemente hinzufügen, aber sie wird immer als kontinuierliches Stück Speicher gemanaged.
- Sie kann nicht nur als „Teil“ im Speicher existieren ... sie existiert immer als „Ganzes“.
- Und das ist, was wir normalerweise auch wollen.
- Wir wollen eine Datenstruktur, die Elemente speichert und uns effizient auf diese zugreifen lässt.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.
- Eine List kann vergrößert werden, in dem wir Elemente hinzufügen, aber sie wird immer als kontinuierliches Stück Speicher gemanaged.
- Sie kann nicht nur als „Teil“ im Speicher existieren ... sie existiert immer als „Ganzes“.
- Und das ist, was wir normalerweise auch wollen.
- Wir wollen eine Datenstruktur, die Elemente speichert und uns effizient auf diese zugreifen lässt.
- Mengen, Tupel, und Dictionaries sind andere Manifestationen dieses Konzepts.

Von Sequenzen zu Datenstrukturen im Speicher



- List Comprehension gibt uns die Möglichkeit, eine Sequenz von Daten in eine Liste zu gießen.
- Eine Liste ist eine Datenstruktur die im Prinzip in einem Stück Speicher alle Elemente der Liste speichert.
- Eine List kann vergrößert werden, in dem wir Elemente hinzufügen, aber sie wird immer als kontinuierliches Stück Speicher gemanaged.
- Sie kann nicht nur als „Teil“ im Speicher existieren ... sie existiert immer als „Ganzes“.
- Und das ist, was wir normalerweise auch wollen.
- Wir wollen eine Datenstruktur, die Elemente speichert und uns effizient auf diese zugreifen lässt.
- Mengen, Tupel, und Dictionaries sind andere Manifestationen dieses Konzepts.
- Sie alle halten alle ihre Elemente komplett im Speicher.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.
- `sum([i ** 2 for i in range(100)])` z. B. addiert die Quadrate der Zahlen `i` für `i` $\in 0..99$.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.
- `sum([i ** 2 for i in range(100)])` z. B. addiert die Quadrate der Zahlen `i` für $i \in 0..99$.
- Die List Comprehension erzeugt dafür zuerst eine Liste mit allen diesen Quadratzahlen.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.
- `sum([i ** 2 for i in range(100)])` z. B. addiert die Quadrate der Zahlen `i` für $\texttt{i} \in 0..99$.
- Die List Comprehension erzeugt dafür zuerst eine Liste mit allen diesen Quadratzahlen.
- Diese Liste wird dann an `sum` übergeben.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.
- `sum([i ** 2 for i in range(100)])` z. B. addiert die Quadrate der Zahlen `i` für $i \in 0..99$.
- Die List Comprehension erzeugt dafür zuerst eine Liste mit allen diesen Quadratzahlen.
- Diese Liste wird dann an `sum` übergeben.
- Diese Funktion iteriert dann über die Liste und addiert die Werte zusammen.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.
- `sum([i ** 2 for i in range(100)])` z. B. addiert die Quadrate der Zahlen `i` für $i \in 0..99$.
- Die List Comprehension erzeugt dafür zuerst eine Liste mit allen diesen Quadratzahlen.
- Diese Liste wird dann an `sum` übergeben.
- Diese Funktion iteriert dann über die Liste und addiert die Werte zusammen.
- Das sieht erstmal gut aus.

Comprehension und Iteration



- Das ist aber nicht für *jede* Aufgabe die richtige Lösung.
- Sagen wir, dass Sie eine Sequenz von Elementen haben, die Sie alle aufsummieren wollen.
- Die Funktion `sum` macht genau das[5].
- Sie akzeptiert einen `Iterable` als Parameter[5].
- Dann ruft sie wieder und wieder `next` auf, um seine Elemente zu bekommen, und addiert diese zu einer laufenden Summe zusammen.
- `sum([i ** 2 for i in range(100)])` z. B. addiert die Quadrate der Zahlen `i` für $\texttt{i} \in 0..99$.
- Die List Comprehension erzeugt dafür zuerst eine Liste mit allen diesen Quadratzahlen.
- Diese Liste wird dann an `sum` übergeben.
- Diese Funktion iteriert dann über die Liste und addiert die Werte zusammen.
- Das sieht erstmal gut aus.
- Es hat aber einen Nachteil.

Immer alles im Speicher?

- Es hat aber einen Nachteil.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.
- Alles was wir wirklich wollen, ist wiederholt `next` auf einen `Iterator` über die Daten aufzurufen, wie wir das in Einheit 34 diskutiert haben.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.
- Alles was wir wirklich wollen, ist wiederholt `next` auf einen `Iterator` über die Daten aufzurufen, wie wir das in Einheit 34 diskutiert haben.
- So ist `sum` auch implementiert.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.
- Alles was wir wirklich wollen, ist wiederholt `next` auf einen `Iterator` über die Daten aufzurufen, wie wir das in Einheit 34 diskutiert haben.
- So ist `sum` auch implementiert.
- Es erwartet gar nicht, dass eine komplette Liste mit Daten im Speicher existiert.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.
- Alles was wir wirklich wollen, ist wiederholt `next` auf einen `Iterator` über die Daten aufzurufen, wie wir das in Einheit 34 diskutiert haben.
- So ist `sum` auch implementiert.
- Es erwartet gar nicht, dass eine komplette Liste mit Daten im Speicher existiert.
- Alles, was es braucht ist ein `Iterable` als Argument, auf das es dann `iter` anwendet um einen `Iterator` zu erhalten.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.
- Alles was wir wirklich wollen, ist wiederholt `next` auf einen `Iterator` über die Daten aufzurufen, wie wir das in Einheit 34 diskutiert haben.
- So ist `sum` auch implementiert.
- Es erwartet gar nicht, dass eine komplette Liste mit Daten im Speicher existiert.
- Alles, was es braucht ist ein `Iterable` als Argument, auf das es dann `iter` anwendet um einen `Iterator` zu erhalten.
- Dann wendet es wieder und wieder `next` auf den `Iterable` an, um die Elemente zu erhalten, die es dann aufsummiert.

Immer alles im Speicher?



- Es hat aber einen Nachteil.
- Zuerst wird die komplette Liste im Speicher erstellt.
- Genau genommen ist alles, was wir (oder die Funktion `sum`) brauchen, eine Möglichkeit, die Elemente eines nach dem anderen auszulesen.
- Wir müssen das auch nur einmal machen.
- Es gibt überhaupt keinen Grund, alle Elemente im Speicher zu halten.
- Alles was wir wirklich wollen, ist wiederholt `next` auf einen `Iterator` über die Daten aufzurufen, wie wir das in Einheit 34 diskutiert haben.
- So ist `sum` auch implementiert.
- Es erwartet gar nicht, dass eine komplette Liste mit Daten im Speicher existiert.
- Alles, was es braucht ist ein `Iterable` als Argument, auf das es dann `iter` anwendet um einen `Iterator` zu erhalten.
- Dann wendet es wieder und wieder `next` auf den `Iterable` an, um die Elemente zu erhalten, die es dann aufsummiert.
- Nirgendwo erwartet es, dass alle Elemente gleichzeitig im Speicher existieren müssen.

Generator-Ausdrücke



Generator-Ausdrücke



- Generator-Ausdrücke[14] existieren, damit wir „bequem“ Sequenzen erstellen können, die ihre Elemente bei Bedarf erstellen und zurückgeben, anstalle alle im Speicher zu halten.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Generator-Ausdrücke

- Generator-Ausdrücke[14] existieren, damit wir „bequem“ Sequenzen erstellen können, die ihre Elemente bei Bedarf erstellen und zurückgeben, anstalle alle im Speicher zu halten.
- Die Syntax von Generator-Ausdrücken ist praktisch die gleiche wie für List Comprehension, mit dem Unterschied, dass wir runde Klammern statt eckiger Klammern verwenden.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Generator-Ausdrücke

- Generator-Ausdrücke[14] existieren, damit wir „bequem“ Sequenzen erstellen können, die ihre Elemente bei Bedarf erstellen und zurückgeben, anstalle alle im Speicher zu halten.
- Die Syntax von Generator-Ausdrücken ist praktisch die gleiche wie für List Comprehension, mit dem Unterschied, dass wir runde Klammern statt eckiger Klammern verwenden.
- Es gibt einen Spezialfall.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Generator-Ausdrücke



- Die Syntax von Generator-Ausdrücken ist praktisch die gleiche wie für List Comprehension, mit dem Unterschied, dass wir runde Klammern statt eckiger Klammern verwenden.
- Es gibt einen Spezialfall:
- Wenn wir Generator-Ausdrücke als Parameter an eine Funktion `my_func` übergeben, dann können wir die Klammern ganz weglassen.

```
1  """Generator Expressions in Python."""
2
3  # Create a generator from all the items in a sequence.
4  # 'expression' is usually an expression whose result depends on 'item'.
5  (expression for item in sequence)
6
7  # Create a generator from those items in a sequence for which
8  # 'condition' evaluates to True.
9  # 'expression' and 'condition' are usually expressions whose results
10 # depend on 'item'.
11 (expression for item in sequence if condition)
12
13 # If generator expressions are single function parameters, then the
14 # parentheses are unnecessary.
15 my_func(expression for item in sequence if condition)
```

Generator-Ausdrücke



- Es gibt einen Spezialfall:
- Wenn wir Generator-Ausdrücke als Parameter an eine Funktion `my_func` übergeben, dann können wir die Klammern ganz weglassen.
- Wenn wir uns an die Syntax von List-, Mengen-, und Dictionary-Comprehension erinnern, könnte man denken, dass wir hier Tupel-Comprehension machen.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Generator-Ausdrücke



- Wenn wir Generator-Ausdrücke als Parameter an eine Funktion `my_func` übergeben, dann können wir die Klammern ganz weglassen.
- Wenn wir uns an die Syntax von List-, Mengen-, und Dictionary-Comprehension erinnern, könnte man denken, dass wir hier Tupel-Comprehension machen.
- Machen wir aber nicht. Generator-Ausdrücke sind etwas anderes.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Generator-Ausdrücke



- Wenn wir uns an die Syntax von List-, Mengen-, und Dictionary-Comprehension erinnern, könnte man denken, dass wir hier Tupel-Comprehension machen.
- Machen wir aber nicht. Generator-Ausdrücke sind etwas anderes.
- Tupel Comprehension gibt es nicht in Python.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Beispiele zur Funktionsweise



Beispiel: Wie Generator-Ausdrücke funktionieren (1)



- Probieren wir das mal aus.

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Probieren wir das mal aus.
- Wir erstellen den Generator-Ausdruck `(i ** 2 for i in range(1_000_000_000))`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Probieren wir das mal aus.
- Wir erstellen den Generator-Ausdruck `(i ** 2 for i in range(1_000_000_000))`.
- Was das List-Comprehension wäre, dann würde das eine Milliarde Ganzzahlen in den Speicher laden.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Probieren wir das mal aus.
- Wir erstellen den Generator-Ausdruck `(i ** 2 for i in range(1_000_000_000))`.
- Was das List-Comprehension wäre, dann würde das eine Milliarde Ganzzahlen in den Speicher laden.
- Weil es aber ein Generator-Ausdruck ist, existiert es im Speicher nur als das Stück Kode in den Klammern.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Probieren wir das mal aus.
- Wir erstellen den Generator-Ausdruck `(i ** 2 for i in range(1_000_000_000))`.
- Was das List-Comprehension wäre, dann würde das eine Milliarde Ganzzahlen in den Speicher laden.
- Weil es aber ein Generator-Ausdruck ist, existiert es im Speicher nur als das Stück Kode in den Klammern.
- Dieser Kode erstellt den nachsten Wert von `i` und berechnet `i ** 2` nur wenn diese wirklich benötigt werden.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"next(gen) = }")  # Returns first element:   0
print(f"next(gen) = }")  # Returns second element:  1
print(f"next(gen) = }")  # Returns third element:   4
print(f"next(gen) = }")  # Returns fourth element:  9
print(f"next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Was das List-Comprehension wäre, dann würde das eine Milliarde Ganzzahlen in den Speicher laden.
- Weil es aber ein Generator-Ausdruck ist, existiert es im Speicher nur als das Stück Kode in den Klammern.
- Dieser Kode erstellt den nachsten Wert von `i` und berechnet `i ** 2` nur wenn diese wirklich benötigt werden.
- Der richtige Type Hint für so einen Generator-Ausdruck ist `Generator[etype]`, wobei `etype` der Datentyp der Elemente ist, hier also `int`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Weil es aber ein Generator-Ausdruck ist, existiert es im Speicher nur als das Stück Kode in den Klammern.
- Dieser Kode erstellt den nachsten Wert von `i` und berechnet `i ** 2` nur wenn diese wirklich benötigt werden.
- Der richtige Type Hint für so einen Generator-Ausdruck ist `Generator[etype]`, wobei `etype` der Datentyp der Elemente ist, hier also `int`.
- `Generator` wird dafür ebenfalls aus dem `typing`-Modul importiert.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Dieser Kode erstellt den nachsten Wert von `i` und berechnet `i ** 2` nur wenn diese wirklich benötigt werden.
- Der richtige Type Hint für so einen Generator-Ausdruck ist `Generator[etype]`, wobei `etype` der Datentyp der Elemente ist, hier also `int`.
- `Generator` wird dafür ebenfalls aus dem `typing`-Modul importiert.
- Wir speichern den Generator-Ausdruck also in einer Variable `gen`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"next(gen) = }")  # Returns first element:   0
print(f"next(gen) = }")  # Returns second element:  1
print(f"next(gen) = }")  # Returns third element:   4
print(f"next(gen) = }")  # Returns fourth element:  9
print(f"next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Dieser Kode erstellt den nachsten Wert von `i` und berechnet `i ** 2` nur wenn diese wirklich benötigt werden.
- Der richtige Type Hint für so einen Generator-Ausdruck ist `Generator[etype]`, wobei `etype` der Datentyp der Elemente ist, hier also `int`.
- `Generator` wird dafür ebenfalls aus dem `typing`-Modul importiert.
- Wir speichern den Generator-Ausdruck also in einer Variable `gen`.
- Dann drucken wir die Variable `gen` aus.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Der richtige Type Hint für so einen Generator-Ausdruck ist `Generator[etype]`, wobei `etype` der Datentyp der Elemente ist, hier also `int`.
- `Generator` wird dafür ebenfalls aus dem `typing`-Modul importiert.
- Wir speichern den Generator-Ausdruck also in einer Variable `gen`.
- Dann drucken wir die Variable `gen` aus.
- Wenn `gen` ein Tupel wäre, dann würde dies eine riesige Zeichenkette mit einer Milliarde Quadratzahlen ausgeben.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- `Generator` wird dafür ebenfalls aus dem `typing`-Modul importiert.
- Wir speichern den Generator-Ausdruck also in einer Variable `gen`.
- Dann drucken wir die Variable `gen` aus.
- Wenn `gen` ein Tupel wäre, dann würde dies eine riesige Zeichenkette mit einer Milliarde Quadratzahlen ausgeben.
- Stattdessen sagt es uns nur den Objekttyp von `gen` und wo es im Speicher liegt.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Wir speichern den Generator-Ausdruck also in einer Variable `gen`.
- Dann drucken wir die Variable `gen` aus.
- Wenn `gen` ein Tupel wäre, dann würde dies eine riesige Zeichenkette mit einer Milliarde Quadratzahlen ausgeben.
- Stattdessen sagt es uns nur den Objekttyp von `gen` und wo es im Speicher liegt.
- Weil es ein Generator-Ausdruck ist, also nur ein Stück Kode im Speicher, gibt es wirklich nicht viel, was Python hier für uns ausgeben kann.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Wenn `gen` ein Tupel wäre, dann würde dies eine riesige Zeichenkette mit einer Milliarde Quadratzahlen ausgeben.
- Stattdessen sagt es uns nur den Objekttyp von `gen` und wo es im Speicher liegt.
- Weil es ein Generator-Ausdruck ist, also nur ein Stück Kode im Speicher, gibt es wirklich nicht viel, was Python hier für uns ausgeben kann.
- Die nächste Zeile in der Ausgabe zeigt uns, dass der `type` von `gen` gleich `<class 'generator'>` ist – nicht `list` und auch nicht `tuple`.

```
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"next(gen) = }")  # Returns first element:   0
print(f"next(gen) = }")  # Returns second element:  1
print(f"next(gen) = }")  # Returns third element:   4
print(f"next(gen) = }")  # Returns fourth element:  9
print(f"next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Stattdessen sagt es uns nur den Objekttyp von `gen` und wo es im Speicher liegt.
- Weil es ein Generator-Ausdruck ist, also nur ein Stück Kode im Speicher, gibt es wirklich nicht viel, was Python hier für uns ausgeben kann.
- Die nächste Zeile in der Ausgabe zeigt uns, dass der `type` von `gen` gleich `<class 'generator'>` ist – nicht `list` und auch nicht `tuple`.
- Wir bestätigen dass Generator-Ausdrücke ein Spezialfall von `Iterator` sind, in dem wir den `isinstance`-Operator verwenden.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Weil es ein Generator-Ausdruck ist, also nur ein Stück Kode im Speicher, gibt es wirklich nicht viel, was Python hier für uns ausgeben kann.
- Die nächste Zeile in der Ausgabe zeigt uns, dass der `type` von `gen` gleich `<class 'generator'>` ist – nicht `list` und auch nicht `tuple`.
- Wir bestätigen dass Generator-Ausdrücke ein Spezialfall von `Iterator` sind, in dem wir den `isinstance`-Operator verwenden.
- Und so ein Iterator kann genau nur einmal verwendet werde.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Weil es ein Generator-Ausdruck ist, also nur ein Stück Kode im Speicher, gibt es wirklich nicht viel, was Python hier für uns ausgeben kann.
- Die nächste Zeile in der Ausgabe zeigt uns, dass der `type` von `gen` gleich `<class 'generator'>` ist – nicht `list` und auch nicht `tuple`.
- Wir bestätigen dass Generator-Ausdrücke ein Spezialfall von `Iterator` sind, in dem wir den `isinstance`-Operator verwenden.
- Und so ein Iterator kann genau nur einmal verwendet werde.
- Das können wir sehen, wenn wir `iter` auf ihn anwenden.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Die nächste Zeile in der Ausgabe zeigt uns, dass der `type` von `gen` gleich `<class 'generator'>` ist – nicht `list` und auch nicht `tuple`.
- Wir bestätigen dass Generator-Ausdrücke ein Spezialfall von `Iterator` sind, in dem wir den `isinstance`-Operator verwenden.
- Und so ein Iterator kann genau nur einmal verwendet werde.
- Das können wir sehen, wenn wir `iter` auf ihn anwenden.
- Dieser Operator gibt uns nämlich das selbe Objekt wieer zurpck, nämlich `gen`, weshalb `iter(gen)is gen` dann `True` ergibt.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Und so ein Iterator kann genau nur einmal verwendet werde.
- Das können wir sehen, wenn wir `iter` auf ihn anwenden.
- Dieser Operator gibt uns nämlich das selbe Objekt wieer zurpck, nämlich `gen`, weshalb `iter(gen)is gen` dann `True` ergibt.
- Wir können beliebig viele Iteratoren für Listen oder Mengen erstellen, wobei jeder davon ein unabhängiges, anderes, eigenständiges Objekt darstellt, mit seiner eigenen Zustandsinformation.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Und so ein Iterator kann genau nur einmal verwendet werde.
- Das können wir sehen, wenn wir `iter` auf ihn anwenden.
- Dieser Operator gibt uns nämlich das selbe Objekt wieer zurpck, nämlich `gen`, weshalb `iter(gen)is gen` dann `True` ergibt.
- Wir können beliebig viele Iteratoren für Listen oder Mengen erstellen, wobei jeder davon ein unabhängiges, anderes, eigenständiges Objekt darstellt, mit seiner eigenen Zustandsinformation.
- `iter(gen)` ergibt daher wieder `gen` selber.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Das können wir sehen, wenn wir `iter` auf ihn anwenden.
- Dieser Operator gibt uns nämlich das selbe Objekt wieer zurpck, nämlich `gen`, weshalb `iter(gen)is gen` dann `True` ergibt.
- Wir können beliebig viele Iteratoren für Listen oder Mengen erstellen, wobei jeder davon ein unabhängiges, anderes, eigenständiges Objekt darstellt, mit seiner eigenen Zustandsinformation.
- `iter(gen)` ergibt daher wieder `gen` selber.
- Und wir iterieren jetzt mal über `gen`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Dieser Operator gibt uns nämlich das selbe Objekt wieer zurpck, nämlich `gen`, weshalb `iter(gen)is gen` dann `True` ergibt.
- Wir können beliebig viele Iteratoren für Listen oder Mengen erstellen, wobei jeder davon ein unabhängiges, anderes, eigenständiges Objekt darstellt, mit seiner eigenen Zustandsinformation.
- `iter(gen)` ergibt daher wieder `gen` selber.
- Und wir iterieren jetzt mal über `gen`.
- Wir bekommen das erste Element (`0`) der Sequenz, in dem wir `next(gen)` aufrufen.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Wir können beliebig viele Iteratoren für Listen oder Mengen erstellen, wobei jeder davon ein unabhängiges, anderes, eigenständiges Objekt darstellt, mit seiner eigenen Zustandsinformation.
- `iter(gen)` ergibt daher wieder `gen` selber.
- Und wir iterieren jetzt mal über `gen`.
- Wir bekommen das erste Element (`0`) der Sequenz, in dem wir `next(gen)` aufrufen.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- `iter(gen)` ergibt daher wieder `gen` selber.
- Und wir iterieren jetzt mal über `gen`.
- Wir bekommen das erste Element (`0`) der Sequenz, in dem wir `next(gen)` aufrufen.
- Der zweite Aufruf von `next(gen)` ergibt dann `1`.
- Der dritte Aufruf von `next(gen)` gibt uns `4`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Und wir iterieren jetzt mal über `gen`.
- Wir bekommen das erste Element (`0`) der Sequenz, in dem wir `next(gen)` aufrufen.
- Der zweite Aufruf von `next(gen)` ergibt dann `1`.
- Der dritte Aufruf von `next(gen)` gibt uns `4`.
- Danach liefert `next(gen)` dann `9`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Wir bekommen das erste Element (`0`) der Sequenz, in dem wir `next(gen)` aufrufen.
- Der zweite Aufruf von `next(gen)` ergibt dann `1`.
- Der dritte Aufruf von `next(gen)` gibt uns `4`.
- Danach liefert `next(gen)` dann `9`.
- Zu guter Letzt machen wir nochmal `next(gen)` und bekommen `16`.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Der zweite Aufruf von `next(gen)` ergibt dann `1`.
- Der dritte Aufruf von `next(gen)` gibt uns `4`.
- Danach liefert `next(gen)` dann `9`.
- Zu guter Letzt machen wir nochmal `next(gen)` und bekommen `16`.
- Im Beispielprogramm hören wir jetzt auf.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (1)

- Der dritte Aufruf von `next(gen)` gibt uns `4`.
- Danach liefert `next(gen)` dann `9`.
- Zu guter Letzt machen wir nochmal `next(gen)` und bekommen `16`.
- Im Beispielprogramm hören wir jetzt auf.
- Das bedeutet, dass die übrigen 999 999 995 Elemente niemals erzeugt werden.

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ python3 generator_expressions_next_1.py ↓

```
<generator object <genexpr> at 0x7f676d2d4c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

Beispiel: Wie Generator-Ausdrücke funktionieren (2)



- Um diese so genannte *lazy evaluation* besser zu verstehen, bauen wir ein zweites Beispielprogramm, nämlich `generator_expressions_next_2.py`.

Beispiel: Wie Generator-Ausdrücke fun

- Um diese so genannte *lazy evaluation* besser zu verstehen, bauen wir ein zweites Beispielprogramm, nämlich `generator_expressions_next_2.py`.
- Wir wollen nun einen Generator-Ausdruck herstellen, bei dem wir genau sehen können, wann seine Elemente erzeugt werden.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Um diese so genannte *lazy evaluation* besser zu verstehen, bauen wir ein zweites Beispielprogramm, nämlich `generator_expressions_next_2.py`.
- Wir wollen nun einen Generator-Ausdruck herstellen, bei dem wir genau sehen können, wann seine Elemente erzeugt werden.
- Dafür definieren wir zuerst die Funktion `as_str`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Um diese so genannte *lazy evaluation* besser zu verstehen, bauen wir ein zweites Beispielprogramm, nämlich `generator_expressions_next_2.py`.
- Wir wollen nun einen Generator-Ausdruck herstellen, bei dem wir genau sehen können, wann seine Elemente erzeugt werden.
- Dafür definieren wir zuerst die Funktion `as_str`.
- `as_str` akzeptiert einen ganzzahligen Parameter `a` als Input.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Um diese so genannte *lazy evaluation* besser zu verstehen, bauen wir ein zweites Beispielprogramm, nämlich `generator_expressions_next_2.py`.
- Wir wollen nun einen Generator-Ausdruck herstellen, bei dem wir genau sehen können, wann seine Elemente erzeugt werden.
- Dafür definieren wir zuerst die Funktion `as_str`.
- `as_str` akzeptiert einen ganzzahligen Parameter `a` als Input.
- Wir wollen genau wissen, wann diese Funktion aufgerufen wird, also fügen wie einen so genannten Seiteneffekt hinzu.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wir wollen nun einen Generator-Ausdruck herstellen, bei dem wir genau sehen können, wann seine Elemente erzeugt werden.
- Dafür definieren wir zuerst die Funktion `as_str`.
- `as_str` akzeptiert einen ganzzahligen Parameter `a` als Input.
- Wir wollen genau wissen, wann diese Funktion aufgerufen wird, also fügen wie einen so genannten Seiteneffekt hinzu.
- Die Funktion schreibt sofort `f"as_str({a})"` zum standard output stream (`stdout`), wenn sie aufgerufen wird.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wir wollen genau wissen, wann diese Funktion aufgerufen wird, also fügen wie einen so genannten Seiteneffekt hinzu.
- Die Funktion schreibt sofort `f"as_str({a})"` zum standard output stream (`stdout`), wenn sie aufgerufen wird.
- Sie macht übergibt auch das Argument `flush=True` an `print`, was erzwingt dass aller zwischengespeicherter Output sofort geschrieben wird, also dass wir *wirklich* sofort sehen, dass unsere Funktion aufgerufen wurde.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"{next(iter(lst))} = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Die Funktion schreibt sofort `f"as_str({a})"` zum standard output stream (`stdout`), wenn sie aufgerufen wird.
- Sie macht übergibt auch das Argument `flush=True` an `print`, was erzwingt dass aller zwischengespeicherter Output sofort geschrieben wird, also dass wir *wirklich* sofort sehen, dass unsere Funktion aufgerufen wurde.
- Mit anderen Worten, wenn `as_str` aufgerufen wird, dann sehen wir es sofort.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Sie macht übergibt auch das Argument `flush=True` an `print`, was erzwingt dass aller zwischengespeicherter Output sofort geschrieben wird, also dass wir *wirklich* sofort sehen, dass unsere Funktion aufgerufen wurde.
- Mit anderen Worten, wenn `as_str` aufgerufen wird, dann sehen wir es sofort.
- Danach liefert die Funktion einfach die Stringrepräsentation von `a` zurück, also `str(a)`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fu

- Mit anderen Worten, wenn `as_str` aufgerufen wird, dann sehen wir es sofort.
- Danach liefert die Funktion einfach die Stringrepräsentation von `a` zurück, also `str(a)`.
- Benutzen wir diese Funktion zuerst in der List-Comprehension `lst = [as_str(j) for j in range(3)]`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
                ^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Mit anderen Worten, wenn `as_str` aufgerufen wird, dann sehen wir es sofort.
- Danach liefert die Funktion einfach die Stringrepräsentation von `a` zurück, also `str(a)`.
- Benutzen wir diese Funktion zuerst in der List-Comprehension `lst = [as_str(j) for j in range(3)]`.
- List-Comprehension erstellt die gesamte Liste sofort.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Danach liefert die Funktion einfach die Stringrepräsentation von `a` zurück, also `str(a)`.
- Benutzen wir diese Funktion zuerst in der List-Comprehension `lst = [as_str(j) for j in range(3)]`.
- List-Comprehension erstellt die gesamte Liste sofort.
- Das bedeutet, dass in dem Moment, in dem diese Zeile Kode ausgeführt wird, unsere Funktion dreimal aufgerufen wird. nämlich mit `a = 0`, `a = 1`, und `a = 2`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Benutzen wir diese Funktion zuerst in der List-Comprehension `lst = [as_str(j) for j in range(3)]`.
- List-Comprehension erstellt die gesamte Liste sofort.
- Das bedeutet, dass in dem Moment, in dem diese Zeile Kode ausgeführt wird, unsere Funktion dreimal aufgerufen wird. nämlich mit `a = 0`, `a = 1`, und `a = 2`.
- Wir sehen, dass das wirklich passiert, weil alle `as_str`-Ausgaben vor dem `print("list created")` in der nächsten Zeile kommen.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"{next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- List-Comprehension erstellt die gesamte Liste sofort.
- Das bedeutet, dass in dem Moment, in dem diese Zeile Kode ausgeführt wird, unsere Funktion dreimal aufgerufen wird. nämlich mit `a = 0`, `a = 1`, und `a = 2`.
- Wir sehen, dass das wirklich passiert, weil alle `as_str`-Ausgaben vor dem `print("list created")` in der nächsten Zeile kommen.
- In der folgenden Zeile drucken wir as Ergebnis von `next(iter(lst))` aus.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- List-Comprehension erstellt die gesamte Liste sofort.
- Das bedeutet, dass in dem Moment, in dem diese Zeile Kode ausgeführt wird, unsere Funktion dreimal aufgerufen wird. nämlich mit `a = 0`, `a = 1`, und `a = 2`.
- Wir sehen, dass das wirklich passiert, weil alle `as_str`-Ausgaben vor dem `print("list created")` in der nächsten Zeile kommen.
- In der folgenden Zeile drucken wir as Ergebnis von `next(iter(lst))` aus.
- `iter(lst))` erzeugt einen `Iterator` über `lst`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Das bedeutet, dass in dem Moment, in dem diese Zeile Kode ausgeführt wird, unsere Funktion dreimal aufgerufen wird. nämlich mit `a = 0`, `a = 1`, und `a = 2`.
- Wir sehen, dass das wirklich passiert, weil alle `as_str`-Ausgaben vor dem `print("list created")` in der nächsten Zeile kommen.
- In der folgenden Zeile drucken wir as Ergebnis von `next(iter(lst))` aus.
- `iter(lst))` erzeugt einen `Iterator` über `lst`.
- Der `next`-Operator liefert dann das erste Element von diesem Iterator.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wir sehen, dass das wirklich passiert, weil alle `as_str`-Ausgaben vor dem `print("list created")` in der nächsten Zeile kommen.
- In der folgenden Zeile drucken wir as Ergebnis von `next(iter(lst))` aus.
- `iter(lst))` erzeugt einen `Iterator` über `lst`.
- Der `next`-Operator liefert dann das erste Element von diesem Iterator.
- Der entsprechende Text taucht in der `stdout` auf.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- In der folgenden Zeile drucken wir as Ergebnis von `next(iter(lst))` aus.
- `iter(lst))` erzeugt einen `Iterator` über `lst`.
- Der `next`-Operator liefert dann das erste Element von diesem Iterator.
- Der entsprechende Text taucht in der `stdout` auf.
- Die Funktion `as_str` wird hierbei nicht noch einmal aufgerufen.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fu

- `iter(lst))` erzeugt einen `Iterator` über `lst`.
- Der `next`-Operator liefert dann das erste Element von diesem Iterator.
- Der entsprechende Text taucht in der `stdout` auf.
- Die Funktion `as_str` wird hierbei nicht noch einmal aufgerufen.
- Natürlich nicht, sie wurde ja schon benutzt, wenn die Liste erstellt wurde.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Der `next`-Operator liefert dann das erste Element von diesem Iterator.
- Der entsprechende Text taucht in der `stdout` auf.
- Die Funktion `as_str` wird hierbei nicht noch einmal aufgerufen.
- Natürlich nicht, sie wurde ja schon benutzt, wenn die Liste erstellt wurde.
- Die Liste wurde also tatsächlich vollständig erstellt, bevor wir angefangen haben, sie zu verarbeiten.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Der entsprechende Text taucht in der `stdout` auf.
- Die Funktion `as_str` wird hierbei nicht noch einmal aufgerufen.
- Natürlich nicht, sie wurde ja schon benutzt, wenn die Liste erstellt wurde.
- Die Liste wurde also tatsächlich vollständig erstellt, bevor wir angefangen haben, sie zu verarbeiten.
- Was passiert, wenn wir `as_str` stattdessen einem Generator-Ausdruck verwenden?

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
                ^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Die Funktion `as_str` wird hierbei nicht noch einmal aufgerufen.
- Natürlich nicht, sie wurde ja schon benutzt, wenn die Liste erstellt wurde.
- Die Liste wurde also tatsächlich vollständig erstellt, bevor wir angefangen haben, sie zu verarbeiten.
- Was passiert, wenn wir `as_str` stattdessen einem Generator-Ausdruck verwenden?
- Wir machen das, in dem wir
  `gen = (as_str(j)`
  `for j in range(3))`
  setzen (natürlich mit dem richtigen Type Hint).

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Natürlich nicht, sie wurde ja schon benutzt, wenn die Liste erstellt wurde.
- Die Liste wurde also tatsächlich vollständig erstellt, bevor wir angefangen haben, sie zu verarbeiten.
- Was passiert, wenn wir `as_str` stattdessen einem Generator-Ausdruck verwenden?
- Wir machen das, in dem wir
  `gen = (as_str(j)`
  `for j in range(3))`
  setzen (natürlich mit dem richtigen Type Hint).
- Wenn diese Zeile Kode ausgeführt wird, passiert erstmal garnichts.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Die Liste wurde also tatsächlich vollständig erstellt, bevor wir angefangen haben, sie zu verarbeiten.
- Was passiert, wenn wir `as_str` stattdessen einem Generator-Ausdruck verwenden?
- Wir machen das, in dem wir
  `gen = (as_str(j)`
  `for j in range(3))`
  setzen (natürlich mit dem richtigen Type Hint).
- Wenn diese Zeile Kode ausgeführt wird, passiert erstmal garnichts.
- Der Generator-Ausdruck ist zu dieser Zeit nur ein Stück Kode im Speicher.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"{next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Was passiert, wenn wir `as_str` stattdessen einem Generator-Ausdruck verwenden?
- Wir machen das, in dem wir `gen = (as_str(j) for j in range(3))` setzen (natürlich mit dem richtigen Type Hint).
- Wenn diese Zeile Kode ausgeführt wird, passiert erstmal garnichts.
- Der Generator-Ausdruck ist zu dieser Zeit nur ein Stück Kode im Speicher.
- Die nächste Zeile `print("generator created")` schreibt ihre Ausgabe.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wir machen das, in dem wir
  `gen = (as_str(j) for j in range(3))`
  setzen (natürlich mit dem richtigen Type Hint).
- Wenn diese Zeile Kode ausgeführt wird, passiert erstmal garnichts.
- Der Generator-Ausdruck ist zu dieser Zeit nur ein Stück Kode im Speicher.
- Die nächste Zeile
  `print("generator created")`
  schreibt ihre Ausgabe.
- Offensichtlich wurde `as_str` noch nicht aufgerufen.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
                             ^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wenn diese Zeile Kode ausgeführt wird, passiert erstmal garnichts.
- Der Generator-Ausdruck ist zu dieser Zeit nur ein Stück Kode im Speicher.
- Die nächste Zeile `print("generator created")` schreibt ihre Ausgabe.
- Offensichtlich wurde `as_str` noch nicht aufgerufen.
- Wir haben noch kein Element aus der Sequenz abgefragt, weshalb der Generator-Ausdruck einfach nur so im Speicher herumliegt.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
                ^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Der Generator-Ausdruck ist zu dieser Zeit nur ein Stück Kode im Speicher.
- Die nächste Zeile `print("generator created")` schreibt ihre Ausgabe.
- Offensichtlich wurde `as_str` noch nicht aufgerufen.
- Wir haben noch kein Element aus der Sequenz abgefragt, weshalb der Generator-Ausdruck einfach nur so im Speicher herumliegt.
- Wir fragen nun die Ergebnisse iterativ von `gen` mit `next` ab.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Die nächste Zeile `print("generator created")` schreibt ihre Ausgabe.
- Offensichtlich wurde `as_str` noch nicht aufgerufen.
- Wir haben noch kein Element aus der Sequenz abgefragt, weshalb der Generator-Ausdruck einfach nur so im Speicher herumliegt.
- Wir fragen nun die Ergebnisse iterativ von `gen` mit `next` ab.
- Das erste Mal, als wir das machen, erscheint `as_str(0)` im `stdout` *vor* dem Ergebnis der Interpolation des f-Strings `f"next(gen): {next(gen)}"`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Offensichtlich wurde `as_str` noch nicht aufgerufen.
- Wir haben noch kein Element aus der Sequenz abgefragt, weshalb der Generator-Ausdruck einfach nur so im Speicher herumliegt.
- Wir fragen nun die Ergebnisse iterativ von `gen` mit `next` ab.
- Das erste Mal, als wir das machen, erscheint `as_str(0)` im `stdout` *vor* dem Ergebnis der Interpolation des f-Strings `f"next(gen): {next(gen)}"`.
- Das ist, weil das element, dass `next` zurückliefern soll, erstmal erstellt werden muss.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wir haben noch kein Element aus der Sequenz abgefragt, weshalb der Generator-Ausdruck einfach nur so im Speicher herumliegt.
- Wir fragen nun die Ergebnisse iterativ von `gen` mit `next` ab.
- Das erste Mal, als wir das machen, erscheint `as_str(0)` im `stdout` *vor* dem Ergebnis der Interpolation des f-Strings `f"next(gen): {next(gen)}"`.
- Das ist, weil das element, dass `next` zurückliefern soll, erstmal erstellt werden muss.
- Der Generator-Ausdruck ruft dafür `as_str(0)` auf.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wir fragen nun die Ergebnisse iterativ von `gen` mit `next` ab.
- Das erste Mal, als wir das machen, erscheint `as_str(0)` im `stdout` *vor* dem Ergebnis der Interpolation des f-Strings `f"next(gen): {next(gen)}"`.
- Das ist, weil das element, dass `next` zurückliefern soll, erstmal erstellt werden muss.
- Der Generator-Ausdruck ruft dafür `as_str(0)` auf.
- In dem Moment wird `"as_str(0)"` in den `stdout` geschrieben, und alle Ausgabepuffer werden geflushed.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Das ist, weil das element, dass `next` zurückliefern soll, erstmal erstellt werden muss.
- Der Generator-Ausdruck ruft dafür `as_str(0)` auf.
- In dem Moment wird `"as_str(0)"` in den stdout geschrieben, und alle Ausgabepuffer werden geflushed.
- Dann wird der String `"0"` an den aufrufenden Kode zurückgeliefert, welcher in dem Fall die String-Interpolation ist.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Das ist, weil das element, dass `next` zurückliefern soll, erstmal erstellt werden muss.
- Der Generator-Ausdruck ruft dafür `as_str(0)` auf.
- In dem Moment wird `"as_str(0)"` in den stdout geschrieben, und alle Ausgabepuffer werden geflushed.
- Dann wird der String `"0"` an den aufrufenden Kode zurückgeliefert, welcher in dem Fall die String-Interpolation ist.
- Der f-String wird zu `"next(gen): '0'}"` interpoliert und ausgrdruckt.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Der Generator-Ausdruck ruft dafür `as_str(0)` auf.
- In dem Moment wird `"as_str(0)"` in den stdout geschrieben, und alle Ausgabepuffer werden geflushed.
- Dann wird der String `"0"` an den aufrufenden Kode zurückgeliefert, welcher in dem Fall die String-Interpolation ist.
- Der f-String wird zu `"next(gen): '0'}"` interpoliert und ausgrdruckt.
- Er erscheint in der Ausgabe *nach* `"as_str(0)"`.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
                             ^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- In dem Moment wird `"as_str(0)"` in den `stdout` geschrieben, und alle Ausgabepuffer werden geflushed.
- Dann wird der String `"0"` an den aufrufenden Kode zurückgeliefert, welcher in dem Fall die String-Interpolation ist.
- Der f-String wird zu `"next(gen): '0'}"` interpoliert und ausgrdruckt.
- Er erscheint in der Ausgabe *nach* `"as_str(0)"`.
- Dabei wurde `as_str` genau nur einmal aufgerufen.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Dann wird der String `"0"` an den aufrufenden Kode zurückgeliefert, welcher in dem Fall die String-Interpolation ist.
- Der f-String wird zu `"next(gen): '0'}"` interpoliert und ausgrdruckt.
- Er erscheint in der Ausgabe *nach* `"as_str(0)"`.
- Dabei wurde `as_str` genau nur einmal aufgerufen.
- Zu diesem Zeitpunkt wurden die anderen Elemente der Sequenz noch nicht erstellt.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Der f-String wird zu `"next(gen): '0'}"` interpoliert und ausgrdruckt.
- Er erscheint in der Ausgabe *nach* `"as_str(0)"`.
- Dabei wurde `as_str` genau nur einmal aufgerufen.
- Zu diesem Zeitpunkt wurden die anderen Elemente der Sequenz noch nicht erstellt.
- Wenn wir nochmal `next(gen)` macen, dann erscheint `as_str(1)`, und so weiter.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Er erscheint in der Ausgabe *nach* `"as_str(0)"`.
- Dabei wurde `as_str` genau nur einmal aufgerufen.
- Zu diesem Zeitpunkt wurden die anderen Elemente der Sequenz noch nicht erstellt.
- Wenn wir nochmal `next(gen)` macen, dann erscheint `as_str(1)`, und so weiter.
- Die Elemente des Generator-Ausdruck werden tatsächlich erst dann erstellt, wenn wir sie brauchen.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Dabei wurde `as_str` genau nur einmal aufgerufen.
- Zu diesem Zeitpunkt wurden die anderen Elemente der Sequenz noch nicht erstellt.
- Wenn wir nochmal `next(gen)` macen, dann erscheint `as_str(1)`, und so weiter.
- Die Elemente des Generator-Ausdruck werden tatsächlich erst dann erstellt, wenn wir sie brauchen.
- Das nennt man *lazy evaluation*.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fu

- Zu diesem Zeitpunkt wurden die anderen Elemente der Sequenz noch nicht erstellt.
- Wenn wir nochmal `next(gen)` macen, dann erscheint `as_str(1)`, und so weiter.
- Die Elemente des Generator-Ausdruck werden tatsächlich erst dann erstellt, wenn wir sie brauchen.
- Das nennt man *lazy evaluation*.
- Kein Speicher wird während dieses Prozesses verschwendet.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Wenn wir nochmal `next(gen)` macen, dann erscheint `as_str(1)`, und so weiter.
- Die Elemente des Generator-Ausdruck werden tatsächlich erst dann erstellt, wenn wir sie brauchen.
- Das nennt man *lazy evaluation*.
- Kein Speicher wird während dieses Prozesses verschwendet.
- Nur ein Element existiert im Speicher zu einer Zeit.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Die Elemente des Generator-Ausdruck werden tatsächlich erst dann erstellt, wenn wir sie brauchen.
- Das nennt man *lazy evaluation*.
- Kein Speicher wird während dieses Prozesses verschwendet.
- Nur ein Element existiert im Speicher zu einer Zeit.
- Nach dem dritten Aufruf von `next(gen)` ist `range(3)` verbraucht.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Das nennt man *lazy evaluation*.
- Kein Speicher wird während dieses Prozesses verschwendet.
- Nur ein Element existiert im Speicher zu einer Zeit.
- Nach dem dritten Aufruf von `next(gen)` ist `range(3)` verbraucht.
- Wenn wir `next(gen)` ein viertes Mal aufrufen, dann bekommen wir eine `StopIteration`-Ausnahme.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Kein Speicher wird während dieses Prozesses verschwendet.
- Nur ein Element existiert im Speicher zu einer Zeit.
- Nach dem dritten Aufruf von `next(gen)` ist `range(3)` verbraucht.
- Wenn wir `next(gen)` ein viertes Mal aufrufen, dann bekommen wir eine `StopIteration`-Ausnahme.
- Das ist kein Fehler, sondern signalisiert das Ende der Iteration.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Beispiel: Wie Generator-Ausdrücke fun

- Nur ein Element existiert im Speicher zu einer Zeit.
- Nach dem dritten Aufruf von `next(gen)` ist `range(3)` verbraucht.
- Wenn wir `next(gen)` ein viertes Mal aufrufen, dann bekommen wir eine `StopIteration`-Ausnahme.
- Das ist kein Fehler, sondern signalisiert das Ende der Iteration.
- Wir können nur einmal über einen Generator-Ausdruck iterieren.

```python
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ python3 generator_expressions_next_2.py ↓

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
          ^^^^^^^^^^^^^^
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

Use Cases



Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.
- Wir können sie auch nicht benutzen, wenn wir in beliebiger Reihenfolge auf Elemente zugreifen wollen, z. B. über einen Index.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.
- Wir können sie auch nicht benutzen, wenn wir in beliebiger Reihenfolge auf Elemente zugreifen wollen, z. B. über einen Index.
- Dann sind die Kollektionsdatentypen besser.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.
- Wir können sie auch nicht benutzen, wenn wir in beliebiger Reihenfolge auf Elemente zugreifen wollen, z. B. über einen Index.
- Dann sind die Kollektionsdatentypen besser.
- Es gibt aber viele Use Cases, in denen Generator-Ausdrücke perfekt passen.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.
- Wir können sie auch nicht benutzen, wenn wir in beliebiger Reihenfolge auf Elemente zugreifen wollen, z. B. über einen Index.
- Dann sind die Kollektionsdatentypen besser.
- Es gibt aber viele Use Cases, in denen Generator-Ausdrücke perfekt passen.

Gute Praxis

Generator-Ausdrücke sind besser als List Comprehension wenn wir die Elemente nur einmal verarbeiten müssen.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.
- Wir können sie auch nicht benutzen, wenn wir in beliebiger Reihenfolge auf Elemente zugreifen wollen, z. B. über einen Index.
- Dann sind die Kollektionsdatentypen besser.
- Es gibt aber viele Use Cases, in denen Generator-Ausdrücke perfekt passen.

Gute Praxis

Generator-Ausdrücke sind besser als List Comprehension wenn wir die Elemente nur einmal verarbeiten müssen. Generator-Ausdrücke brauchen weniger Speicher.

Use Cases



- Wir können über Generator-Ausdrücke nur einmal drüberiterieren.
- Daher können wir sie nicht verwenden, wenn wir mehrmals über eine Sequenz iterieren müssen.
- Wir können sie auch nicht benutzen, wenn wir in beliebiger Reihenfolge auf Elemente zugreifen wollen, z. B. über einen Index.
- Dann sind die Kollektionsdatentypen besser.
- Es gibt aber viele Use Cases, in denen Generator-Ausdrücke perfekt passen.

Gute Praxis

Generator-Ausdrücke sind besser als List Comprehension wenn wir die Elemente nur einmal verarbeiten müssen. Generator-Ausdrücke brauchen weniger Speicher. Wenn die Iteration über die Elemente vorzeitig abgebrochen werden kann, wie das z. B. bei den Funktionen `all` oder `any` passieren kann, dann können Generator-Ausdrücke auch schneller sein.

Beispiel: Generator-Ausdrücke als Argumente



- Zuerst wollen wir die Quadrate aller Zahlen von 0 bis 999 999 aufaddieren.

Beispiel: Generator-Ausdrücke als Argumente



- Zuerst wollen wir die Quadrate aller Zahlen von 0 bis 999 999 aufaddieren.
- Wir können das tun, in dem wir den Generator-Ausdruck `(j ** 2 for j in range(1_000_000))` an die `sum`-Funktion übergeben.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Zuerst wollen wir die Quadrate aller Zahlen von 0 bis 999 999 aufaddieren.
- Wir können das tun, in dem wir den Generator-Ausdruck `(j ** 2 for j in range(1_000_000))` an die `sum`-Funktion übergeben.
- Beachten Sie, dass wir nicht `sum((generator expression...))` schreiben.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Zuerst wollen wir die Quadrate aller Zahlen von 0 bis 999 999 aufaddieren.
- Wir können das tun, in dem wir den Generator-Ausdruck `(j ** 2 for j in range(1_000_000))` an die `sum`-Funktion übergeben.
- Beachten Sie, dass wir nicht `sum((generator expression...))` schreiben.
- Einfache Klammern sind hier ausreichend.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Zuerst wollen wir die Quadrate aller Zahlen von 0 bis 999 999 aufaddieren.
- Wir können das tun, in dem wir den Generator-Ausdruck `(j ** 2 for j in range(1_000_000))` an die `sum`-Funktion übergeben.
- Beachten Sie, dass wir nicht `sum((generator expression...))` schreiben.
- Einfache Klammern sind hier ausreichend.
- Das Ergebnis dieser Summierung ist 333 332 833 333 500 000.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir können das tun, in dem wir den Generator-Ausdruck `(j ** 2 for j in range(1_000_000))` an die `sum`-Funktion übergeben.
- Beachten Sie, dass wir nicht `sum((generator expression...))` schreiben.
- Einfache Klammern sind hier ausreichend.
- Das Ergebnis dieser Summierung ist 333 332 833 333 500 000.
- Jetzt wollen wir den größten und den kleinsten Wert von $\sin k$ für all $k \in -99..99$ berechnen.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Beachten Sie, dass wir nicht `sum((generator expression...))` schreiben.
- Einfache Klammern sind hier ausreichend.
- Das Ergebnis dieser Summierung ist 333 332 833 333 500 000.
- Jetzt wollen wir den größten und den kleinsten Wert von $\sin k$ für all $k \in -99..99$ berechnen.
- Wir können den Generator-Ausdruck `(sin(k) for k in range(-100, 100))` verwenden.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Einfache Klammern sind hier ausreichend.
- Das Ergebnis dieser Summierung ist 333 332 833 333 500 000.
- Jetzt wollen wir den größten und den kleinsten Wert von $\sin k$ für all $k \in -99..99$ berechnen.
- Wir können den Generator-Ausdruck `(sin(k) for k in range(-100, 100))` verwenden.
- Wir schreiben ihn einmal hin, um ihn an `max` zu übergeben, damit wir den größten Wert bekommen können.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Das Ergebnis dieser Summierung ist 333 332 833 333 500 000.
- Jetzt wollen wir den größten und den kleinsten Wert von $\sin k$ für all $k \in -99..99$ berechnen.
- Wir können den Generator-Ausdruck `(sin(k) for k in range(-100, 100))` verwenden.
- Wir schreiben ihn einmal hin, um ihn an `max` zu übergeben, damit wir den größten Wert bekommen können.
- Dann schreiben wir ihn ein zweites Mal hin, um ihn an `min` zu übergeben, damit wir den kleinsten Wert bekommen.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Jetzt wollen wir den größten und den kleinsten Wert von $\sin k$ für all $k \in -99..99$ berechnen.
- Wir können den Generator-Ausdruck `(sin(k)for k in range(-100, 100))` verwenden.
- Wir schreiben ihn einmal hin, um ihn an `max` zu übergeben, damit wir den größten Wert bekommen können.
- Dann schreiben wir ihn ein zweites Mal hin, um ihn an `min` zu übergeben, damit wir den kleinsten Wert bekommen.
- Es ist wieder nicht notwendig, doppelte Klammern zu verwenden.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir können den Generator-Ausdruck `(sin(k)for k in range(-100, 100))` verwenden.
- Wir schreiben ihn einmal hin, um ihn an `max` zu übergeben, damit wir den größten Wert bekommen können.
- Dann schreiben wir ihn ein zweites Mal hin, um ihn an `min` zu übergeben, damit wir den kleinsten Wert bekommen.
- Es ist wieder nicht notwendig, doppelte Klammern zu verwenden.
- Für das nächste Beispiel erstellen wir erst eine Liste `words` mit den Worten `["hello", "how", "are", "you"]`.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir schreiben ihn einmal hin, um ihn an `max` zu übergeben, damit wir den größten Wert bekommen können.
- Dann schreiben wir ihn ein zweites Mal hin, um ihn an `min` zu übergeben, damit wir den kleinsten Wert bekommen.
- Es ist wieder nicht notwendig, doppelte Klammern zu verwenden.
- Für das nächste Beispiel erstellen wir erst eine Liste `words` mit den Worten `["hello", "how", "are", "you"]`.
- Wir wollen wissen, ob *alle* Worte in der Liste den Buchstabe „e“ beinhalten.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ `python3 generator_expressions_in_reduction.py` ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Es ist wieder nicht notwendig, doppelte Klammern zu verwenden.
- Für das nächste Beispiel erstellen wir erst eine Liste `words` mit den Worten `["hello", "how", "are", "you"]`.
- Wir wollen wissen, ob *alle* Worte in der Liste den Buchstabe „e“ beinhalten.
- Der Generator-Ausdruck `"e" in word for word in words` wertet `"e" in word` für jedes `word` in der Liste `words` aus.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Es ist wieder nicht notwendig, doppelte Klammern zu verwenden.
- Für das nächste Beispiel erstellen wir erst eine Liste `words` mit den Worten `["hello", "how", "are", "you"]`.
- Wir wollen wissen, ob *alle* Worte in der Liste den Buchstabe „e“ beinhalten.
- Der Generator-Ausdruck `"e" in word for word in words` wertet `"e" in word` für jedes `word` in der Liste `words` aus.
- Er würde die Sequenz `True`, `False`, `True`, und `False` produzieren.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir wollen wissen, ob *alle* Worte in der Liste den Buchstabe „e“ beinhalten.
- Der Generator-Ausdruck `"e" in word for word in words` wertet `"e" in word` für jedes `word` in der Liste `words` aus.
- Er würde die Sequenz `True`, `False`, `True`, und `False` produzieren.
- Die Funktion `all` liefert `True` wenn und nur wenn alle Elemente aus der Sequenz, die sie als Parameter bekommt, ebenfalls `True` sind.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir wollen wissen, ob *alle* Worte in der Liste den Buchstabe „e“ beinhalten.
- Der Generator-Ausdruck `"e" in word for word in words` wertet `"e" in word` für jedes `word` in der Liste `words` aus.
- Er würde die Sequenz `True`, `False`, `True`, und `False` produzieren.
- Die Funktion `all` liefert `True` wenn und nur wenn alle Elemente aus der Sequenz, die sie als Parameter bekommt, ebenfalls `True` sind.
- Sonst liefert sie `False` zurück.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Der Generator-Ausdruck `"e" in word for word in words` wertet `"e" in word` für jedes `word` in der Liste `words` aus.
- Er würde die Sequenz `True`, `False`, `True`, und `False` produzieren.
- Die Funktion `all` liefert `True` wenn und nur wenn alle Elemente aus der Sequenz, die sie als Parameter bekommt, ebenfalls `True` sind.
- Sonst liefert sie `False` zurück.
- Wir übergeben unseren Generator-Ausdruck als Argument an diese Funktion.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Er würde die Sequenz `True`, `False`, `True`, und `False` produzieren.
- Die Funktion `all` liefert `True` wenn und nur wenn alle Elemente aus der Sequenz, die sie als Parameter bekommt, ebenfalls `True` sind.
- Sonst liefert sie `False` zurück.
- Wir übergeben unseren Generator-Ausdruck als Argument an diese Funktion.
- Sie wird hier natürlich `False` zurückliefern.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Die Funktion `all` liefert `True` wenn und nur wenn alle Elemente aus der Sequenz, die sie als Parameter bekommt, ebenfalls `True` sind.
- Sonst liefert sie `False` zurück.
- Wir übergeben unseren Generator-Ausdruck als Argument an diese Funktion.
- Sie wird hier natürlich `False` zurückliefern.
- Das Interessante ist, dass `all` aufhören kann, `next` aufzurufen, sobald das erste Mal `False` auftaucht.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Sonst liefert sie `False` zurück.
- Wir übergeben unseren Generator-Ausdruck als Argument an diese Funktion.
- Sie wird hier natürlich `False` zurückliefern.
- Das Interessante ist, dass `all` aufhören kann, `next` aufzurufen, sobald das erste Mal `False` auftaucht.
- Mit anderen Worten, unser Generator-Ausdruck wird niemals `"e" in "are"` auswerten, weil nämlich `"e" in "how"` schon `False` ergab.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ `python3 generator_expressions_in_reduction.py` ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Sie wird hier natürlich `False` zurückliefern.
- Das Interessante ist, dass `all` aufhören kann, `next` aufzurufen, sobald das erste Mal `False` auftaucht.
- Mit anderen Worten, unser Generator-Ausdruck wird niemals `"e" in "are"` auswerten, weil nämlich `"e" in "how"` schon `False` ergab.
- Zu diesem Zeitpunkt ist klar, dass `all` auf jeden Fall `False` zurückliefert, gleichgültig was noch in der Sequenz kommt ... und genau das macht es auch sofort.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Mit anderen Worten, unser Generator-Ausdruck wird niemals `"e" in "are"` auswerten, weil nämlich `"e" in "how"` schon `False` ergab.
- Zu diesem Zeitpunkt ist klar, dass `all` auf jeden Fall `False` zurückliefert, gleichgültig was noch in der Sequenz kommt ... und genau das macht es auch sofort.
- Hätten wir List Comprehension verwendet, dann wäre jedes einzelne geprüft worden und der entsprechende `bool`-Wert wäre in einer `list` gespeichert worden.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Zu diesem Zeitpunkt ist klar, dass `all` auf jeden Fall `False` zurückliefert, gleichgültig was noch in der Sequenz kommt ... und genau das macht es auch sofort.
- Hätten wir List Comprehension verwendet, dann wäre jedes einzelne geprüft worden und der entsprechende `bool`-Wert wäre in einer `list` gespeichert worden.
- Durch den Generator-Ausdruck wurde nur ein Teil der Daten generiert, der auch gebraucht wurde, ein Wert nach dem Anderen, und niemals irgendwo gespeichert.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Hätten wir List Comprehension verwendet, dann wäre jedes einzelne geprüft worden und der entsprechende `bool`-Wert wäre in einer `list` gespeichert worden.
- Durch den Generator-Ausdruck wurde nur ein Teil der Daten generiert, der auch gebraucht wurde, ein Wert nach dem Anderen, und niemals irgendwo gespeichert.
- Die `any`-Funktion komplementiert die `all`-Funktion.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Hätten wir List Comprehension verwendet, dann wäre jedes einzelne geprüft worden und der entsprechende `bool`-Wert wäre in einer `list` gespeichert worden.
- Durch den Generator-Ausdruck wurde nur ein Teil der Daten generiert, der auch gebraucht wurde, ein Wert nach dem Anderen, und niemals irgendwo gespeichert.
- Die `any`-Funktion komplementiert die `all`-Funktion.
- `any` liefert `True` wenn wenigstens eines der Elemente der Sequenz, die sie als Argument empfängt, auch `True` ist.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Durch den Generator-Ausdruck wurde nur ein Teil der Daten generiert, der auch gebraucht wurde, ein Wert nach dem Anderen, und niemals irgendwo gespeichert.
- Die `any`-Funktion komplementiert die `all`-Funktion.
- `any` liefert `True` wenn wenigstens eines der Elemente der Sequenz, die sie als Argument empfängt, auch `True` ist.
- Es liefert `False` wenn *alle* Elemente der Argumentsequenz `False` sind.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Durch den Generator-Ausdruck wurde nur ein Teil der Daten generiert, der auch gebraucht wurde, ein Wert nach dem Anderen, und niemals irgendwo gespeichert.
- Die `any`-Funktion komplementiert die `all`-Funktion.
- `any` liefert `True` wenn wenigstens eines der Elemente der Sequenz, die sie als Argument empfängt, auch `True` ist.
- Es liefert `False` wenn *alle* Elemente der Argumentsequenz `False` sind.
- Wir wollen jetzt wissen, ob der Buchstabe „w“ in irgendeinem der Worte auftaucht.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Die `any`-Funktion komplementiert die `all`-Funktion.
- `any` liefert `True` wenn wenigstens eines der Elemente der Sequenz, die sie als Argument empfängt, auch `True` ist.
- Es liefert `False` wenn *alle* Elemente der Argumentsequenz `False` sind.
- Wir wollen jetzt wissen, ob der Buchstabe „w“ in irgendeinem der Worte auftaucht.
- Dafür schreiben wir den Generator-Ausdruck `("w" in word for word in words)`.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- `any` liefert `True` wenn wenigstens eines der Elemente der Sequenz, die sie als Argument empfängt, auch `True` ist.
- Es liefert `False` wenn *alle* Elemente der Argumentsequenz `False` sind.
- Wir wollen jetzt wissen, ob der Buchstabe „w“ in irgendeinem der Worte auftaucht.
- Dafür schreiben wir den Generator-Ausdruck `("w" in word for word in words)`.
- Dieser wertet `"w" in word` für jedes `word` in der Liste `words` aus.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Es liefert `False` wenn *alle* Elemente der Argumentsequenz `False` sind.
- Wir wollen jetzt wissen, ob der Buchstabe „w“ in irgendeinem der Worte auftaucht.
- Dafür schreiben wir den Generator-Ausdruck `("w" in word for word in words)`.
- Dieser wertet `"w" in word` für jedes `word` in der Liste `words` aus.
- Wir übergeben diesen Generator-Ausdruck an `any`.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir wollen jetzt wissen, ob der Buchstabe „w“ in irgendeinem der Worte auftaucht.
- Dafür schreiben wir den Generator-Ausdruck `("w" in word for word in words)`.
- Dieser wertet `"w" in word` für jedes `word` in der Liste `words` aus.
- Wir übergeben diesen Generator-Ausdruck an `any`.
- Wir sehen sofort, dass `"w" in "how"` shown `True` ist.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Dafür schreiben wir den Generator-Ausdruck `("w" in word for word in words)`.
- Dieser wertet `"w" in word` für jedes `word` in der Liste `words` aus.
- Wir übergeben diesen Generator-Ausdruck an `any`.
- Wir sehen sofort, dass `"w" in "how"` shown `True` ist.
- Daher ist das Ergebnis von `any` auch `True`.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Dieser wertet `"w" in word` für jedes `word` in der Liste `words` aus.
- Wir übergeben diesen Generator-Ausdruck an `any`.
- Wir sehen sofort, dass `"w" in "how"` shown `True` ist.
- Daher ist das Ergebnis von `any` auch `True`.
- Daher kann die Auswertung der Sequenz bereits an der Stelle abbrechen, wo der erste `True`-Wert erreicht wird.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Generator-Ausdrücke als Argumente



- Wir übergeben diesen Generator-Ausdruck an `any`.
- Wir sehen sofort, dass `"w" in "how"` shown `True` ist.
- Daher ist das Ergebnis von `any` auch `True`.
- Daher kann die Auswertung der Sequenz bereits an der Stelle abbrechen, wo der erste `True`-Wert erreicht wird.
- Hier spart uns also die *lazy evaluation* viel Laufzeit.

```python
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ python3 generator_expressions_in_reduction.py ↓

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wir haben mal gesagt, dass in Python die `for`-Schleifen eigentlich auf Iteratoren arbeiten.

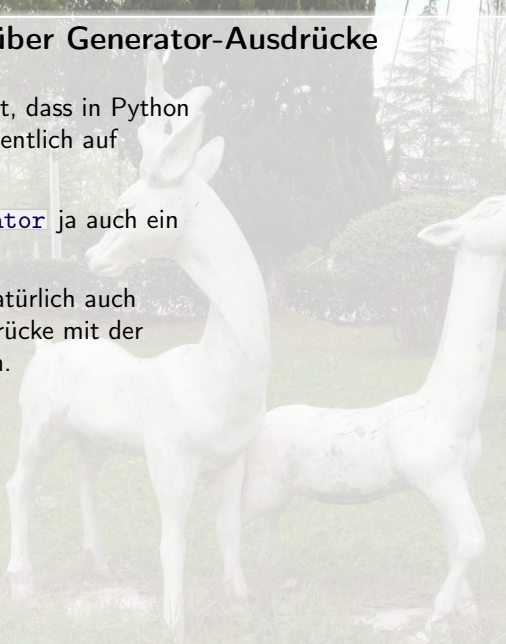


Beispiel: Schleifen über Generator-Ausdrücke



- Wir haben mal gesagt, dass in Python die `for`-Schleifen eigentlich auf Iteratoren arbeiten.
- Nun ist jeder `Generator` ja auch ein `Iterator`.

Beispiel: Schleifen über Generator-Ausdrücke



- Wir haben mal gesagt, dass in Python die `for`-Schleifen eigentlich auf Iteratoren arbeiten.
- Nun ist jeder `Generator` ja auch ein `Iterator`.
- Darum können wir natürlich auch über Generator-Ausdrücke mit der `for`-Schleife iterieren.

Beispiel: Schleifen über Generator-Ausdrücke

- Wir haben mal gesagt, dass in Python die `for`-Schleifen eigentlich auf Iteratoren arbeiten.
- Nun ist jeder `Generator` ja auch ein `Iterator`.
- Darum können wir natürlich auch über Generator-Ausdrücke mit der `for`-Schleife iterieren.
- Das sehen wir nun in Beispiel `generator_expressions_loops.py`.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wir haben mal gesagt, dass in Python die `for`-Schleifen eigentlich auf Iteratoren arbeiten.
- Nun ist jeder `Generator` ja auch ein `Iterator`.
- Darum können wir natürlich auch über Generator-Ausdrücke mit der `for`-Schleife iterieren.
- Das sehen wir nun in Beispiel `generator_expressions_loops.py`.
- Hier erstellen wir den Generator-Ausdruck `gen`, der Tupel der Form $(n, \sin n)$ für alle $n \in 0..1\,000\,000\,000$ erstellt.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke



- Nun ist jeder `Generator` ja auch ein `Iterator`.
- Darum können wir natürlich auch über Generator-Ausdrücke mit der `for`-Schleife iterieren.
- Das sehen wir nun in Beispiel `generator_expressions_loops.py`.
- Hier erstellen wir den Generator-Ausdruck `gen`, der Tupel der Form $(n, \sin n)$ für alle $n \in 0..1\,000\,000\,000$ erstellt.
- Wir schreiben dafür `((n, sin(n))for n in range(1_000_000_000))`.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Darum können wir natürlich auch über Generator-Ausdrücke mit der `for`-Schleife iterieren.
- Das sehen wir nun in Beispiel `generator_expressions_loops.py`.
- Hier erstellen wir den Generator-Ausdruck `gen`, der Tupel der Form $(n, \sin n)$ für alle $n \in 0..1\,000\,000\,000$ erstellt.
- Wir schreiben dafür `((n, sin(n))for n in range(1_000_000_000))`.
- Nehmen wir an, dass wir aus irgendeinem Grund das erste $n \in \mathbb{N}_0$ finden wollen, für das gilt $\sin n < -0.99999$.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Das sehen wir nun in Beispiel `generator_expressions_loops.py`.
- Hier erstellen wir den Generator-Ausdruck `gen`, der Tupel der Form $(n, \sin n)$ für alle $n \in 0..1\,000\,000\,000$ erstellt.
- Wir schreiben dafür `((n, sin(n))for n in range(1_000_000_000))`.
- Nehmen wir an, dass wir aus irgendeinem Grund das erste $n \in \mathbb{N}_0$ finden wollen, für das gilt $\sin n < -0.99999$.
- Wir iterieren dafür über den Generator-Ausdruck `gen`.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke



- Wir schreiben dafür `((n, sin(n))for n in range(1_000_000_000))`.
- Nehmen wir an, dass wir aus irgendeinem Grund das erste $n \in \mathbb{N}_0$ finden wollen, für das gilt $\sin n < -0.99999$.
- Wir iterieren dafür über den Generator-Ausdruck `gen`.
- Wir entpacken die Tupel, die der Ausdruck liefert, genauso wie wir das in Einheit 23 schonmal für Kollektionen gemacht haben.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Nehmen wir an, dass wir aus irgendeinem Grund das erste $n \in \mathbb{N}_0$ finden wollen, für das gilt $\sin n < -0.99999$.
- Wir iterieren dafür über den Generator-Ausdruck `gen`.
- Wir entpacken die Tupel, die der Ausdruck liefert, genauso wie wir das in Einheit 23 schonmal für Kollektionen gemacht haben.
- Die Variable `o` bekommt den Wert von `n`, der als erstes Tupel-Element gespeichert war, und die Variable `p` bekommt den entsprechenden Wert $\sin$ `o`.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wir iterieren dafür über den Generator-Ausdruck `gen`.
- Wir entpacken die Tupel, die der Ausdruck liefert, genauso wie wir das in Einheit 23 schonmal für Kollektionen gemacht haben.
- Die Variable `o` bekommt den Wert von `n`, der als erstes Tupel-Element gespeichert war, und die Variable `p` bekommt den entsprechenden Wert $\sin$ `o`.
- Wir packen dann ein `if p < -0.99999` in die Schleife um unsere Suchbedingung zu prüfen.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke



- Wir entpacken die Tupel, die der Ausdruck liefert, genauso wie wir das in Einheit 23 schonmal für Kollektionen gemacht haben.
- Die Variable `o` bekommt den Wert von `n`, der als erstes Tupel-Element gespeichert war, und die Variable `p` bekommt den entsprechenden Wert $\sin$ `o`.
- Wir packen dann ein `if p < -0.99999` in die Schleife um unsere Suchbedingung zu prüfen.
- Wenn der Ausdruck `p < -0.99999` wahr wird, dann drucken wir den entdeckten Wert und verlassen die Schleife mit `break`.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Die Variable `o` bekommt den Wert von `n`, der als erstes Tupel-Element gespeichert war, und die Variable `p` bekommt den entsprechenden Wert $\sin$ `o`.
- Wir packen dann ein `if p < -0.99999` in die Schleife um unsere Suchbedingung zu prüfen.
- Wenn der Ausdruck `p < -0.99999` wahr wird, dann drucken wir den entdeckten Wert und verlassen die Schleife mit `break`.
- Wenn die `for`-Schleife nicht durch `break` verlassen wird, dann ist die Bedingung für der `else` nach ihrem Körper erfüllt.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ python3 generator_expressions_loops.py ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wir packen dann ein `if p < -0.99999` in die Schleife um unsere Suchbedingung zu prüfen.
- Wenn der Ausdruck `p < -0.99999` wahr wird, dann drucken wir den entdeckten Wert und verlassen die Schleife mit `break`.
- Wenn die `for`-Schleife nicht durch `break` verlassen wird, dann ist die Bedingung für der `else` nach ihrem Körper erfüllt.
- In diesem Fall geben wir eine Nachricht aus, dass wir keinen Wert gefunden haben, der unsere Bedingung erfüllt.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wenn der Ausdruck `p < -0.99999` wahr wird, dann drucken wir den entdeckten Wert und verlassen die Schleife mit `break`.
- Wenn die `for`-Schleife nicht durch `break` verlassen wird, dann ist die Bedingung für der `else` nach ihrem Körper erfüllt.
- In diesem Fall geben wir eine Nachricht aus, dass wir keinen Wert gefunden haben, der unsere Bedingung erfüllt.
- Die Ausgabe `sin(11)=-0.9999902065507035` zeigt uns jedoch, dass es sehr wohl eine passende Zahl gibt.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wenn die `for`-Schleife nicht durch `break` verlassen wird, dann ist die Bedingung für der `else` nach ihrem Körper erfüllt.
- In diesem Fall geben wir eine Nachricht aus, dass wir keinen Wert gefunden haben, der unsere Bedingung erfüllt.
- Die Ausgabe `sin(11)=-0.9999902065507035` zeigt uns jedoch, dass es sehr wohl eine passende Zahl gibt.
- Wir wissen auch, dass diese schon nach 12 Schleifendurchläufen gefunden wurde, weil n ja bei 0 losgeht.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Die Ausgabe `sin(11)=-0.9999902065507035` zeigt uns jedoch, dass es sehr wohl eine passende Zahl gibt.
- Wir wissen auch, dass diese schon nach 12 Schleifendurchläufen gefunden wurde, weil n ja bei 0 losgeht.
- Einen Generator-Ausdruck hier zu verwenden war nicht nur speichereffizienter als das Nutzen einer Liste, es war auch schneller, weil wir nicht alle Werte berechnen mussten.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Die Ausgabe `sin(11)=-0.9999902065507035` zeigt uns jedoch, dass es sehr wohl eine passende Zahl gibt.
- Wir wissen auch, dass diese schon nach 12 Schleifendurchläufen gefunden wurde, weil n ja bei 0 losgeht.
- Einen Generator-Ausdruck hier zu verwenden war nicht nur speichereffizienter als das Nutzen einer Liste, es war auch schneller, weil wir nicht alle Werte berechnen mussten.
- OK, wir hätten auch einfach eine normale `for`-Schleife nehmen können.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wir wissen auch, dass diese schon nach 12 Schleifendurchläufen gefunden wurde, weil n ja bei 0 losgeht.
- Einen Generator-Ausdruck hier zu verwenden war nicht nur speichereffizienter als das Nutzen einer Liste, es war auch schneller, weil wir nicht alle Werte berechnen mussten.
- OK, wir hätten auch einfach eine normale `for`-Schleife nehmen können.
- Das wäre noch besser gewesen.

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ python3 generator_expressions_loops.py ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Schleifen über Generator-Ausdrücke

- Wir wissen auch, dass diese schon nach 12 Schleifendurchläufen gefunden wurde, weil n ja bei 0 losgeht.
- Einen Generator-Ausdruck hier zu verwenden war nicht nur speichereffizienter als das Nutzen einer Liste, es war auch schneller, weil wir nicht alle Werte berechnen mussten.
- OK, wir hätten auch einfach eine normale `for`-Schleife nehmen können.
- Das wäre noch besser gewesen.
- Aber ich wollte ja ein Beispiel für Generator-Ausdrücke machen...

```python
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ python3 generator_expressions_loops.py ↓

```
sin(11)=-0.9999902065507035
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Natürlich können Generator-Ausdrücke auch an die Konstruktoren von Kollektion übergeben werden, oder an andere Funktionen, die solche Datenstrukturen erstellen.

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Natürlich können Generator-Ausdrücke auch an die Konstruktoren von Kollektion übergeben werden, oder an andere Funktionen, die solche Datenstrukturen erstellen.
- Um das zu probieren, enhmen wir an, dass wir daten im comma-separated values (CSV)-Format haben.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Natürlich können Generator-Ausdrücke auch an die Konstruktoren von Kollektion übergeben werden, oder an andere Funktionen, die solche Datenstrukturen erstellen.
- Um das zu probieren, enhmen wir an, dass wir daten im comma-separated values (CSV)-Format haben.
- In diesem Format werden oft Daten gespeichert, die in tabellarischer Form oder als Matrix vorliegen.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Natürlich können Generator-Ausdrücke auch an die Konstruktoren von Kollektion übergeben werden, oder an andere Funktionen, die solche Datenstrukturen erstellen.
- Um das zu probieren, enhmen wir an, dass wir daten im comma-separated values (CSV)-Format haben.
- In diesem Format werden oft Daten gespeichert, die in tabellarischer Form oder als Matrix vorliegen.
- Hier entspricht jede Textzeile einer Zeile von Datenelementen.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Um das zu probieren, enhmen wir an, dass wir daten im comma-separated values (CSV)-Format haben.
- In diesem Format werden oft Daten gespeichert, die in tabellarischer Form oder als Matrix vorliegen.
- Hier entspricht jede Textzeile einer Zeile von Datenelementen.
- Die einzelnen Elemente in solch einer Zeile werden dann durch Kommas („,“) von einander getrennt.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Um das zu probieren, enhmen wir an, dass wir daten im comma-separated values (CSV)-Format haben.
- In diesem Format werden oft Daten gespeichert, die in tabellarischer Form oder als Matrix vorliegen.
- Hier entspricht jede Textzeile einer Zeile von Datenelementen.
- Die einzelnen Elemente in solch einer Zeile werden dann durch Kommas („,“) von einander getrennt.
- In `generator_expressions_to_collection.py` definieren wir zuerst einen String `csv_text` mit dem Wert `"22,56,33,67,43,33,12"`.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- In diesem Format werden oft Daten gespeichert, die in tabellarischer Form oder als Matrix vorliegen.
- Hier entspricht jede Textzeile einer Zeile von Datenelementen.
- Die einzelnen Elemente in solch einer Zeile werden dann durch Kommas („,“) von einander getrennt.
- In `generator_expressions_to_collection.py` definieren wir zuerst einen String `csv_text` mit dem Wert `"22,56,33,67,43,33,12"`.
- Er enthält also eine Zeile von CSV-Daten.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Die einzelnen Elemente in solch einer Zeile werden dann durch Kommas („,“) von einander getrennt.
- In `generator_expressions_to_collection.py` definieren wir zuerst einen String `csv_text` mit dem Wert `"22,56,33,67,43,33,12"`.
- Er enthält also eine Zeile von CSV-Daten.
- Wenn wir `csv_text.split(",")` aufrufen, dann wird der String in einzelne Zeichenketten aufgespalten, und zwar immer am Trennzeichen `","`.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- In `generator_expressions_to_collection.py` definieren wir zuerst einen String `csv_text` mit dem Wert `"22,56,33,67,43,33,12"`.
- Er enthält also eine Zeile von CSV-Daten.
- Wenn wir `csv_text.split(",")` aufrufen, dann wird der String in einzelne Zeichenketten aufgespalten, und zwar immer am Trennzeichen `","`.
- Das ergibt die Liste `["22", "56", "33", "67", "43", "33", "12"]`.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Er enthält also eine Zeile von CSV-Daten.
- Wenn wir `csv_text.split(",")` aufrufen, dann wird der String in einzelne Zeichenketten aufgespalten, und zwar immer am Trennzeichen `","`.
- Das ergibt die Liste `["22", "56", "33", "67", "43", "33", "12"]`.
- Wir können diese Liste in Generator-Ausdrücken verwenden.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Wenn wir `csv_text.split(",")` aufrufen, dann wird der String in einzelne Zeichenketten aufgespalten, und zwar immer am Trennzeichen `","`.
- Das ergibt die Liste `["22", "56", "33", "67", "43", "33", "12"]`.
- Wir können diese Liste in Generator-Ausdrücken verwenden.
- Zum Beispiel ist `(int(i)for i in csv_text.split(","))` ein Generator-Ausdruck der die Strings in aus der Liste in Ganzzahlen umwandelt.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Wir können diese Liste in Generator-Ausdrücken verwenden.
- Zum Beispiel ist `(int(i)for i in csv_text.split(","))` ein Generator-Ausdruck der die Strings in aus der Liste in Ganzzahlen umwandelt.
- Wenn wir diesen Generator-Ausdruck an den Konstruktor `list` übergeben, also `list(int(i)for i in csv_text.split(","))` aufrufen, dann bekommen wir eine Liste mit allen Elementen, die der Ausdruck erstellt.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Zum Beispiel ist `(int(i)for i in csv_text.split(","))` ein Generator-Ausdruck der die Strings in aus der Liste in Ganzzahlen umwandelt.
- Wenn wir diesen Generator-Ausdruck an den Konstruktor `list` übergeben, also `list(int(i)for i in csv_text.split(","))` aufrufen, dann bekommen wir eine Liste mit allen Elementen, die der Ausdruck erstellt.
- Das funktioniert genauso, wenn wir den Ausdruck als Parameter an `tuple` oder `set` übergeben, welche dann ein Tupel bzw. eine Menge erstellen.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Wenn wir diesen Generator-Ausdruck an den Konstruktor `list` übergeben, also `list(int(i)for i in csv_text.split(","))` aufrufen, dann bekommen wir eine Liste mit allen Elementen, die der Ausdruck erstellt.
- Das funktioniert genauso, wenn wir den Ausdruck als Parameter an `tuple` oder `set` übergeben, welche dann ein Tupel bzw. eine Menge erstellen.
- Beachten Sie allerdings, dass die Menge keine Duplikate enthält.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Wenn wir diesen Generator-Ausdruck an den Konstruktor `list` übergeben, also `list(int(i)for i in csv_text.split(","))` aufrufen, dann bekommen wir eine Liste mit allen Elementen, die der Ausdruck erstellt.
- Das funktioniert genauso, wenn wir den Ausdruck als Parameter an `tuple` oder `set` übergeben, welche dann ein Tupel bzw. eine Menge erstellen.
- Beachten Sie allerdings, dass die Menge keine Duplikate enthält.
- Wir können den Generator-Ausdruck auch an die Funktion `sorted` übergeben.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Das funktioniert genauso, wenn wir den Ausdruck als Parameter an `tuple` oder `set` übergeben, welche dann ein Tupel bzw. eine Menge erstellen.
- Beachten Sie allerdings, dass die Menge keine Duplikate enthält.
- Wir können den Generator-Ausdruck auch an die Funktion `sorted` übergeben.
- Diese Funktion erstellt eine sortierte Liste mit allen Elementen aus der Sequenz, die sie als Argument erhält.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Das funktioniert genauso, wenn wir den Ausdruck als Parameter an `tuple` oder `set` übergeben, welche dann ein Tupel bzw. eine Menge erstellen.
- Beachten Sie allerdings, dass die Menge keine Duplikate enthält.
- Wir können den Generator-Ausdruck auch an die Funktion `sorted` übergeben.
- Diese Funktion erstellt eine sortierte Liste mit allen Elementen aus der Sequenz, die sie als Argument erhält.
- Dictionaries können aus Sequenzen von Tupeln erstellt werden.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Beachten Sie allerdings, dass die Menge keine Duplikate enthält.
- Wir können den Generator-Ausdruck auch an die Funktion `sorted` übergeben.
- Diese Funktion erstellt eine sortierte Liste mit allen Elementen aus der Sequenz, die sie als Argument erhält.
- Dictionaries können aus Sequenzen von Tupeln erstellt werden.
- Das erste Element jedes Tupels wird dann als Schlüssel verwendet und das zweite als Wert.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Wir können den Generator-Ausdruck auch an die Funktion `sorted` übergeben.
- Diese Funktion erstellt eine sortierte Liste mit allen Elementen aus der Sequenz, die sie als Argument erhält.
- Dictionaries können aus Sequenzen von Tupeln erstellt werden.
- Das erste Element jedes Tupels wird dann als Schlüssel verwendet und das zweite als Wert.
- Wir übergeben den Generator-Ausdruck, der die Tupel `(i, int(i))` für jedes `i` in `csv_text.split(",")` liefert an `dict`.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ python3 generator_expressions_to_collection.py ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Dictionaries können aus Sequenzen von Tupeln erstellt werden.
- Das erste Element jedes Tupels wird dann als Schlüssel verwendet und das zweite als Wert.
- Wir übergeben den Generator-Ausdruck, der die Tupel `(i, int(i))` für jedes `i` in `csv_text.split(",")` liefert an `dict`.
- Die Schlüssel des neuen Dictionaries sind daher die originalen Strings unserer CSV-Daten und die Werte sind ihre Ganzzahl-Repräsentation.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Das erste Element jedes Tupels wird dann als Schlüssel verwendet und das zweite als Wert.
- Wir übergeben den Generator-Ausdruck, der die Tupel `(i, int(i))` für jedes `i` in `csv_text.split(",")` liefert an `dict`.
- Die Schlüssel des neuen Dictionaries sind daher die originalen Strings unserer CSV-Daten und die Werte sind ihre Ganzzahl-Repräsentation.
- Beachten Sie, dass Dictionary ebenfalls keine doppelten Schlüssel beinhalten.

```python
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33':
    ↪ 33, '67': 67, '43': 43, '12': 12}.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Wir übergeben den Generator-Ausdruck, der die Tupel `(i, int(i))` für jedes `i` in `csv_text.split(",")` liefert an `dict`.
- Die Schlüssel des neuen Dictionaries sind daher die originalen Strings unserer CSV-Daten und die Werte sind ihre Ganzzahl-Repräsentation.
- Beachten Sie, dass Dictionary ebenfalls keine doppelten Schlüssel beinhalten.
- Nun wenden wir unser gutes altes Werkzeug Ruff auf `generator_expressions_to_collection.py` an.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLW1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79
    ↪ generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
   ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
      ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.4 failed with exit code 1.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Die Schlüssel des neuen Dictionaries sind daher die originalen Strings unserer CSV-Daten und die Werte sind ihre Ganzzahl-Repräsentation.
- Beachten Sie, dass Dictionary ebenfalls keine doppelten Schlüssel beinhalten.
- Nun wenden wir unser gutes altes Werkzeug Ruff auf `generator_expressions_to_collection.py` an.
- Es sagt uns, dass es keinen Sinn ergibt, Generator-Ausdrücke zum Erstellen von Listen, Mengen, und Dictionaries zu verwenden.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79
    ↪ generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
   ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
      ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.4 failed with exit code 1.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Beachten Sie, dass Dictionary ebenfalls keine doppelten Schlüssel beinhalten.
- Nun wenden wir unser gutes altes Werkzeug Ruff auf `generator_expressions_to_collection.py` an.
- Es sagt uns, dass es keinen Sinn ergibt, Generator-Ausdrücke zum Erstellen von Listen, Mengen, und Dictionaries zu verwenden.
- Wir könnten genauso gut die entsprechenden Comprehensions direkt verwenden.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,TID,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79
    ↪ generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
   ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
      ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.4 failed with exit code 1.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Beachten Sie, dass Dictionary ebenfalls keine doppelten Schlüssel beinhalten.
- Nun wenden wir unser gutes altes Werkzeug Ruff auf `generator_expressions_to_collection.py` an.
- Es sagt uns, dass es keinen Sinn ergibt, Generator-Ausdrücke zum Erstellen von Listen, Mengen, und Dictionaries zu verwenden.
- Wir könnten genauso gut die entsprechenden Comprehensions direkt verwenden.
- Und Ruff hat recht.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79
    ↪ generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
   ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
      ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.4 failed with exit code 1.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Nun wenden wir unser gutes altes Werkzeug Ruff auf `generator_expressions_to_collection.py` an.
- Es sagt uns, dass es keinen Sinn ergibt, Generator-Ausdrücke zum Erstellen von Listen, Mengen, und Dictionaries zu verwenden.
- Wir könnten genauso gut die entsprechenden Comprehensions direkt verwenden.
- Und Ruff hat recht.
- Es gibt allerdings keine Tuple-Comprehension, also ist das Verwenden eines Generator-Ausdrucks in `tuple(...)` schon OK.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79
    ↪ generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
   ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
      ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.4 failed with exit code 1.
```

Beispiel: Kollektionen mit Generator-Ausdrücken füllen

- Es sagt uns, dass es keinen Sinn ergibt, Generator-Ausdrücke zum Erstellen von Listen, Mengen, und Dictionaries zu verwenden.
- Wir könnten genauso gut die entsprechenden Comprehensions direkt verwenden.
- Und Ruff hat recht.
- Es gibt allerdings keine Tuple-Comprehension, also ist das Verwenden eines Generator-Ausdrucks in `tuple(...)` schon OK.
- Das selbe gilt auch für die `sorted`-Funktion.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79
    ↪ generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
   ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary
   ↪ comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
      ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.4 failed with exit code 1.
```

Zusammenfassung



Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.

Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.
- Wir haben sie als eine effiziente Quelle von Daten kennengelernt.

Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.
- Wir haben sie als eine effiziente Quelle von Daten kennengelernt.
- Anders als Kollektions-Datenstrukturen halten sie nicht alle ihre Elemente im Speicher.

Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.
- Wir haben sie als eine effiziente Quelle von Daten kennengelernt.
- Anders als Kollektions-Datenstrukturen halten sie nicht alle ihre Elemente im Speicher.
- Stattdessen werden sie *lazily evaluated*, ihre Elemente werden also erst dann berechnet und zurückgeliefert, wenn sie benötigt werden.

Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.
- Wir haben sie als eine effiziente Quelle von Daten kennengelernt.
- Anders als Kollektions-Datenstrukturen halten sie nicht alle ihre Elemente im Speicher.
- Stattdessen werden sie *lazily evaluated*, ihre Elemente werden also erst dann berechnet und zurückgeliefert, wenn sie benötigt werden.
- Das ist viel besser, wenn wir auf Elemente sowieso nur eins nach dem Anderen zugreifen wollen.

Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.
- Wir haben sie als eine effiziente Quelle von Daten kennengelernt.
- Anders als Kollektions-Datenstrukturen halten sie nicht alle ihre Elemente im Speicher.
- Stattdessen werden sie *lazily evaluated*, ihre Elemente werden also erst dann berechnet und zurückgeliefert, wenn sie benötigt werden.
- Das ist viel besser, wenn wir auf Elemente sowieso nur eins nach dem Anderen zugreifen wollen.
- Es ist sogar doppelt effizient, wenn wir vielleicht die Iteration vorzeitig abbrechen wollen.

Zusammenfassung



- Damit haben wir das Ende unserer Diskussion von Generator-Ausdrücken erreicht.
- Wir haben sie als eine effiziente Quelle von Daten kennengelernt.
- Anders als Kollektions-Datenstrukturen halten sie nicht alle ihre Elemente im Speicher.
- Stattdessen werden sie *lazily evaluated*, ihre Elemente werden also erst dann berechnet und zurückgeliefert, wenn sie benötigt werden.
- Das ist viel besser, wenn wir auf Elemente sowieso nur eins nach dem Anderen zugreifen wollen.
- Es ist sogar doppelt effizient, wenn wir vielleicht die Iteration vorzeitig abbrechen wollen.
- Dann sparen wir sowohl Laufzeit als auch Speicher.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 221, 222).

[2] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 221).

[3] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 221).

[4] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 221).

[5] "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. 13–23).

[6] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 222, 223).

[7] "`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. 221).

[8] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 221).

[9] "Formatted String Literals". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 7.1.1. URL: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals (besucht am 2024-07-25) (siehe S. 221).

[10] Bhavesh Gawade. "Mastering F-Strings in Python: Efficient String Handling in Python Using Smart F-Strings". In: *C O D E B*. Mumbai, Maharashtra, India: Code B Solutions Pvt Ltd, 25. Apr.–3. Juni 2025. URL: https://code-b.dev/blog/f-strings-in-python (besucht am 2025-08-04) (siehe S. 221).

References II

[11] Olaf Górski. "Why f-strings are awesome: Performance of different string concatenation methods in Python". In: *DEV Community*. Sacramento, CA, USA: DEV Community Inc., 8. Nov. 2022. URL: https://dev.to/grski/performance-of-different-string-concatenation-methods-in-python-why-f-strings-are-awesome-2e97 (besucht am 2025-08-04) (siehe S. 221).

[12] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 221).

[13] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 223).

[14] Raymond Hettinger. *Generator Expressions*. Python Enhancement Proposal (PEP) 274. Beaverton, OR, USA: Python Software Foundation (PSF), 30. Jan. 2002. URL: https://peps.python.org/pep-0289 (besucht am 2024-11-08) (siehe S. 36–38).

[15] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 222).

[16] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (besucht am 2024-08-23) (siehe S. 221).

[17] ."stderr, stdin, stdout – Standard I/O Streams". In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Hrsg. von Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/stdin.html (besucht am 2024-10-30) (siehe S. 222).

[18] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 223).

[19] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 222).

References III

[20] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 222).

[21] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 222).

[22] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 222).

[23] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: https://docs.astral.sh/ruff (besucht am 2024-08-23) (siehe S. 222).

[24] Aaron Maxwell. *What are f-strings in Python and how can I use them?* Oakville, ON, Canada: Infinite Skills Inc, Juni 2017. ISBN: 978-1-4919-9486-3 (siehe S. 221).

[25] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 221).

[26] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 221).

[27] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (besucht am 2024-06-29) (siehe S. 221).

[28] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: 978-1-4503-9475-8. doi:10.1145/3551349.3559497 (siehe S. 221).

References IV

[29] 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. 221).

[30] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 221).

[31] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 221).

[32] Eric V. „ericvsmith" Smith. *Literal String Interpolation*. Python Enhancement Proposal (PEP) 498. Beaverton, OR, USA: Python Software Foundation (PSF), 6. Nov. 2016–9. Sep. 2023. URL: https://peps.python.org/pep-0498 (besucht am 2024-07-25) (siehe S. 221).

[33] *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).

[34] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 221).

[35] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 221, 223).

[36] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 223).

[37] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 221).

[38] Thomas Weise (汤卫思). *Programming with C*. 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), 2026. URL: https://thomasweise.github.io/programmingWithC (besucht am 2026-05-10) (siehe S. 221).

References V

[39] 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), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 222).

[40] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 221).

Glossary (in English) I

Bash is a shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[3,25,40]. Learn more at https://www.gnu.org/software/bash.

C is a programming language, which is very successful in system programming situations[8,27,38].

CSV *Comma-Separated Values* is a very common and simple text format for exchanging tabular or matrix data[29]. 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[7].

f-string let you include the results of expressions in strings[4,9–11,24,32]. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.

Git is a distributed Version Control System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[31,35]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[26,35]. Learn more at https://github.com.

IT information technology

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles[16,28]. Ruff is an example of a linter used in the Python world.

Linux is the leading open source operating system, i.e., a free alternative to Microsoft Windows[1,12,30,34,37]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

Microsoft Windows is a commercial proprietary operating system[2]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

Glossary (in English) II



Mypy is a static type checking tool for Python[20] that makes use of type hints. Learn more at `https://github.com/python/mypy` and in [39].

Python The Python programming language[15,19,21,39], i.e., what you will learn about in our book[39]. Learn more at `https://python.org`.

Ruff is a linter and code formatting tool for Python[22,23]. Learn more at `https://docs.astral.sh/ruff` or in [39].

stderr The *standard error stream* is one of the three pre-defined streams of a console process (together with the standard input stream (`stdin`) and the `stdout`)[17]. It is the text stream to which the process writes information about errors and exceptions. If an uncaught `Exception` is raised in Python and the program terminates, then this information is written to standard error stream (`stderr`). If you run a program in a terminal, then the text that a process writes to its `stderr` appears in the console.

stdin The *standard input stream* is one of the three pre-defined streams of a console process (together with the `stdout` and the `stderr`)[17]. It is the text stream from which the process reads its input text, if any. The Python instruction `input` reads from this stream. If you run a program in a terminal, then the text that you type into the terminal while the process is running appears in this stream.

stdout The *standard output stream* is one of the three pre-defined streams of a console process (together with the `stdin` and the `stderr`)[17]. It is the text stream to which the process writes its normal output. The `print` instruction of Python writes text to this stream. If you run a program in a terminal, then the text that a process writes to its `stdout` appears in the console.

(string) interpolation In Python, string interpolation is the process where all the expressions in an f-string are evaluated and the final string is constructed. An example of string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`.

terminal A terminal is a text-based window where you can enter commands and execute them[1,6]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

Glossary (in English) III



type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[18,36]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

Ubuntu is a variant of the open source operating system Linux[6,13]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[35]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.

$i..j$ with $i, j \in \mathbb{Z}$ and $i \leq j$ is the set that contains all integer numbers in the inclusive range from i to j. It always holds that $i..j \subset \mathbb{Z}$. For example, 5..9 is equivalent to $\{5, 6, 7, 8, 9\}$

$\mathbb{N_0}$ the set of the natural numbers *including* 0, i.e., 0, 1, 2, 3, and so on. It holds that $\mathbb{N_0} \subset \mathbb{Z}$ and that $\mathbb{N_0} = \mathbb{N_1} \cup \{0\}$.

$\mathbb{N_1}$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N_1} \subset \mathbb{Z}$ and that $\mathbb{N_1} = \mathbb{N_0} \setminus \{0\}$.

$\mathbb{R}$ the set of the real numbers.

$\mathbb{Z}$ the set of the integers numbers including positive and negative numbers and 0, i.e., . . . , -3, -2, -1, 0, 1, 2, 3, . . . , and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.