合肥大學
HEFEI UNIVERSITY

Programming with Python

35. List Comprehension

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

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

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-10-30

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. List Comprehension
3. Beispiele
4. Performanz
5. Zusammenfassung

Einleitung

Einleitung

- Wir können Listen erstellen, in dem wir diese als Literale hinschreiben.

Einleitung

- Wir können Listen erstellen, in dem wir diese als Literale hinschreiben.
- Zum Beispiel erstellt `[1, 2, 3, 4, 5]` eine Liste mit den ersten fünf natürlichen Zahlen.

Einleitung

- Wir können Listen erstellen, in dem wir diese als Literale hinschreiben.
- Zum Beispiel erstellt `[1, 2, 3, 4, 5]` eine Liste mit den ersten fünf natürlichen Zahlen.
- Das ist sehr nützlich, kann aber umständlich werden wenn wir entweder viele Elemente haben oder diese irgendwie transformieren wollen.

Einleitung

- Wir können Listen erstellen, in dem wir diese als Literale hinschreiben.
- Zum Beispiel erstellt `[1, 2, 3, 4, 5]` eine Liste mit den ersten fünf natürlichen Zahlen.
- Das ist sehr nützlich, kann aber umständlich werden wenn wir entweder viele Elemente haben oder diese irgendwie transformieren wollen.
- Ein Literal für eine Liste mit den ersten hundert natürlichen Zahlen zu schreiben ist umständlich.

Einleitung

- Wir können Listen erstellen, in dem wir diese als Literale hinschreiben.
- Zum Beispiel erstellt `[1, 2, 3, 4, 5]` eine Liste mit den ersten fünf natürlichen Zahlen.
- Das ist sehr nützlich, kann aber umständlich werden wenn wir entweder viele Elemente haben oder diese irgendwie transformieren wollen.
- Ein Literal für eine Liste mit den ersten hundert natürlichen Zahlen zu schreiben ist umständlich.
- So etwas wie `[log(1), log(2), log(3), log(4), log(5)]` zu schreiben sieht auch nicht sehr gut aus.

Einleitung

- Wir können Listen erstellen, in dem wir diese als Literale hinschreiben.
- Zum Beispiel erstellt `[1, 2, 3, 4, 5]` eine Liste mit den ersten fünf natürlichen Zahlen.
- Das ist sehr nützlich, kann aber umständlich werden wenn wir entweder viele Elemente haben oder diese irgendwie transformieren wollen.
- Ein Literal für eine Liste mit den ersten hundert natürlichen Zahlen zu schreiben ist umständlich.
- So etwas wie `[log(1), log(2), log(3), log(4), log(5)]` zu schreiben sieht auch nicht sehr gut aus.
- Zum Glück bietet Python die viel bequemere Syntax für *list comprehension*[39].

List Comprehension

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Diese Syntax erzeugt eine neue Liste deren Inhalt das Ergebnis einem Ausdruck `expression` auf die Elemente `item` einer `sequence` sind[39].

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Diese Syntax erzeugt eine neue Liste deren Inhalt das Ergebnis einem Ausdruck `expression` auf die Elemente `item` einer `sequence` sind[39].
- Sie können sich das wie eine `for`-Schleife vorstellen, bei der jede Iteration einen Wert erzeugt, der dann in einer Liste gespeichert wird.

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Diese Syntax erzeugt eine neue Liste deren Inhalt das Ergebnis einem Ausdruck `expression` auf die Elemente `item` einer `sequence` sind[39].
- Sie können sich das wie eine `for`-Schleife vorstellen, bei der jede Iteration einen Wert erzeugt, der dann in einer Liste gespeichert wird.
- Z. B. `[i for i in range(10)]` erzeugt eine Liste mit den Ganzzahlen von `0` bis `9`.

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Sie können sich das wie eine `for`-Schleife vorstellen, bei der jede Iteration einen Wert erzeugt, der dann in einer Liste gespeichert wird.
- Z. B. `[i for i in range(10)]` erzeugt eine Liste mit den Ganzzahlen von `0` bis `9`.
- Die List Comprehension `[i ** 2 for i in range(10)]` erzeugt dagegen eine Liste mit den Quadraten dieser Zahlen, wobei der Quadrieren hier der Ausdruck `expression` ist.

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Z. B. `[i for i in range(10)]` erzeugt eine Liste mit den Ganzzahlen von `0` bis `9`.
- Die List Comprehension `[i ** 2 for i in range(10)]` erzeugt dagegen eine Liste mit den Quadraten dieser Zahlen, wobei der Quadrieren hier der Ausdruck `expression` ist.
- Wir können optional auch die Elemente auswählen, die wir in der Liste haben wollen, in dem wir eine `if`-Klausel anfügen.

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Z. B. `[i for i in range(10)]` erzeugt eine Liste mit den Ganzzahlen von `0` bis `9`.
- Die List Comprehension `[i ** 2 for i in range(10)]` erzeugt dagegen eine Liste mit den Quadraten dieser Zahlen, wobei der Quadrieren hier der Ausdruck `expression` ist.
- Wir können optional auch die Elemente auswählen, die wir in der Liste haben wollen, in dem wir eine `if`-Klausel anfügen.
- Z. B. `[i for i in range(10)if i != 3]` schließt die Zahl `3` von unserer List aus.

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Wir können optional auch die Elemente auswählen, die wir in der Liste haben wollen, in dem wir eine `if`-Klausel anfügen.
- Z. B. `[i for i in range(10)if i != 3]` schließt die Zahl `3` von unserer List aus.
- Interessanter Weise kann die Sequenz über die List Comprehension iteriert selbst auch so ein Comprehension-Ausdruck sein.

Syntax

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list 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]
```

- Z. B. `[i for i in range(10)if i != 3]` schließt die Zahl `3` von unserer List aus.
- Interessanter Weise kann die Sequenz über die List Comprehension iteriert selbst auch so ein Comprehension-Ausdruck sein.
- Es ist OK zu schreiben `[i * j for i in range(2)for j in range(2)]`, was uns die Liste `[0, 0, 0, 1]` liefert, weil `i` und `j` beide jeweils unabhängig von einander nacheinander die Werte `0` und `1` annehmen.

Beispiele

Beispiel

- Schauen wir uns das mal an.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Schauen wir uns das mal an.
- Zuerst wollen wir eine Liste mit den Quadraten der Ganzzahlen von 0 bis 10 erstellen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Schauen wir uns das mal an.
- Zuerst wollen wir eine Liste mit den Quadraten der Ganzzahlen von 0 bis 10 erstellen.
- Bevor wir über List Comprehension gelernt haben würden wir das ganz einfach mit einer normalen `for`-Schleife machen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Schauen wir uns das mal an.
- Zuerst wollen wir eine Liste mit den Quadraten der Ganzzahlen von 0 bis 10 erstellen.
- Bevor wir über List Comprehension gelernt haben würden wir das ganz einfach mit einer normalen `for`-Schleife machen.
- Wir würden anfangen, in dem wir eine leere Liste `squares_1` erstellen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Schauen wir uns das mal an.
- Zuerst wollen wir eine Liste mit den Quadraten der Ganzzahlen von 0 bis 10 erstellen.
- Bevor wir über List Comprehension gelernt haben würden wir das ganz einfach mit einer normalen `for`-Schleife machen.
- Wir würden anfangen, in dem wir eine leere Liste `squares_1` erstellen.
- In der `for`-Schleife würden wir eine Variable `i` über `range(0, 11)` iterieren lassen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Bevor wir über List Comprehension gelernt haben würden wir das ganz einfach mit einer normalen `for`-Schleife machen.
- Wir würden anfangen, in dem wir eine leere Liste `squares_1` erstellen.
- In der `for`-Schleife würden wir eine Variable `i` über `range(0, 11)` iterieren lassen.
- Im Körper der Schleife würden wir dann jeweils `i ** 2` an die Liste `squares_1` anhängen, in dem wir `squares_1.append(i ** 2)` aufrufen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Bevor wir über List Comprehension gelernt haben würden wir das ganz einfach mit einer normalen `for`-Schleife machen.
- Wir würden anfangen, in dem wir eine leere Liste `squares_1` erstellen.
- In der `for`-Schleife würden wir eine Variable `i` über `range(0, 11)` iterieren lassen.
- Im Körper der Schleife würden wir dann jeweils `i ** 2` an die Liste `squares_1` anhängen, in dem wir `squares_1.append(i ** 2)` aufrufen.
- Das benötigt drei Zeilen von Kode. Aber es geht.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- In der `for`-Schleife würden wir eine Variable `i` über `range(0, 11)` iterieren lassen.
- Im Körper der Schleife würden wir dann jeweils `i ** 2` an die Liste `squares_1` anhängen, in dem wir `squares_1.append(i ** 2)` aufrufen.
- Das benötigt drei Zeilen von Kode. Aber es geht.
- Stattdessen können wir aber auch einfach schreiben `[j ** 2 for j in range(11)]`, was genau das selbe Ergebnis mit nur einer Zeile Kode erreicht.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Im Körper der Schleife würden wir dann jeweils `i ** 2` an die Liste `squares_1` anhängen, in dem wir `squares_1.append(i ** 2)` aufrufen.
- Das benötigt drei Zeilen von Kode. Aber es geht.
- Stattdessen können wir aber auch einfach schreiben `[j ** 2 for j in range(11)]`, was genau das selbe Ergebnis mit nur einer Zeile Kode erreicht.
- Wir können uns auch aussuchen, welche Elemente einer Sequenz wir in unsere Liste einfügen wollen, in dem wir ein `if`-Statement in der List Comprehension verwenden.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Das benötigt drei Zeilen von Kode. Aber es geht.
- Stattdessen können wir aber auch einfach schreiben `[j ** 2 for j in range(11)]`, was genau das selbe Ergebnis mit nur einer Zeile Kode erreicht.
- Wir können uns auch aussuchen, welche Elemente einer Sequenz wir in unsere Liste einfügen wollen, in dem wir ein `if`-Statement in der List Comprehension verwenden.
- Im Beispiel probieren wir das aus, in dem wir eine Liste der geraden Zahlen aus der Range 0 bis 9 erstellen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Stattdessen können wir aber auch einfach schreiben `[j ** 2 for j in range(11)]`, was genau das selbe Ergebnis mit nur einer Zeile Kode erreicht.
- Wir können uns auch aussuchen, welche Elemente einer Sequenz wir in unsere Liste einfügen wollen, in dem wir ein `if`-Statement in der List Comprehension verwenden.
- Im Beispiel probieren wir das aus, in dem wir eine Liste der geraden Zahlen aus der Range 0 bis 9 erstellen.
- Wir lassen eine Variable `k` über `range(10)` iterieren.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Wir können uns auch aussuchen, welche Elemente einer Sequenz wir in unsere Liste einfügen wollen, in dem wir ein `if`-Statement in der List Comprehension verwenden.
- Im Beispiel probieren wir das aus, in dem wir eine Liste der geraden Zahlen aus der Range 0 bis 9 erstellen.
- Wir lassen eine Variable `k` über `range(10)` iterieren.
- `k` nimmt also die Werte `0`, `1`, `2`, ..., `8`, und `9` an.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Wir können uns auch aussuchen, welche Elemente einer Sequenz wir in unsere Liste einfügen wollen, in dem wir ein `if`-Statement in der List Comprehension verwenden.
- Im Beispiel probieren wir das aus, in dem wir eine Liste der geraden Zahlen aus der Range 0 bis 9 erstellen.
- Wir lassen eine Variable `k` über `range(10)` iterieren.
- `k` nimmt also die Werte `0`, `1`, `2`, ..., `8`, und `9` an.
- Von diesem Werten nehmen wir nur die, für die `k % 2 == 0` zutrifft.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Im Beispiel probieren wir das aus, in dem wir eine Liste der geraden Zahlen aus der Range 0 bis 9 erstellen.
- Wir lassen eine Variable `k` über `range(10)` iterieren.
- `k` nimmt also die Werte `0`, `1`, `2`, ..., `8`, und `9` an.
- Von diesem Werten nehmen wir nur die, für die `k % 2 == 0` zutrifft.
- Wir berechnen also den Rest der Division von `k` und `2`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Wir lassen eine Variable `k` über `range(10)` iterieren.
- `k` nimmt also die Werte `0`, `1`, `2`, ..., `8`, und `9` an.
- Von diesem Werten nehmen wir nur die, für die `k % 2 == 0` zutrifft.
- Wir berechnen also den Rest der Division von `k` und `2`.
- Wenn der 0 ist, dann ist `k` durch `2` teilbar und daher gerade.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- `k` nimmt also die Werte `0`, `1`, `2`, ..., `8`, und `9` an.
- Von diesem Werten nehmen wir nur die, für die `k % 2 == 0` zutrifft.
- Wir berechnen also den Rest der Division von `k` und `2`.
- Wenn der 0 ist, dann ist `k` durch `2` teilbar und daher gerade.
- Ja, ja, ich weiß ... wir hätten das auch ohne `if` machen können, wenn wir die Range `range(0, 10, 2)` genommen hätten ... oder wenn wir gleich `list(range(0, 10, 2))` gemacht hätten... ...es ist ja nur ein Beispiel.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Von diesem Werten nehmen wir nur die, für die `k % 2 == 0` zutrifft.
- Wir berechnen also den Rest der Division von `k` und `2`.
- Wenn der 0 ist, dann ist `k` durch `2` teilbar und daher gerade.
- Ja, ja, ich weiß ... wir hätten das auch ohne `if` machen können, wenn wir die Range `range(0, 10, 2)` genommen hätten ... oder wenn wir gleich `list(range(0, 10, 2))` gemacht hätten... ...es ist ja nur ein Beispiel.
- So oder so, wir bekommen die Liste `[0, 2, 4, 6, 8]`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Wir berechnen also den Rest der Division von `k` und `2`.
- Wenn der 0 ist, dann ist `k` durch `2` teilbar und daher gerade.
- Ja, ja, ich weiß ... wir hätten das auch ohne `if` machen können, wenn wir die Range `range(0, 10, 2)` genommen hätten ... oder wenn wir gleich `list(range(0, 10, 2))` gemacht hätten... ...es ist ja nur ein Beispiel.
- So oder so, wir bekommen die Liste `[0, 2, 4, 6, 8]`.
- Nun spielen wir noch mit verschachtelter Comprehension.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Ja, ja, ich weiß ... wir hätten das auch ohne `if` machen können, wenn wir die Range `range(0, 10, 2)` genommen hätten ... oder wenn wir gleich `list(range(0, 10, 2))` gemacht hätten... ...es ist ja nur ein Beispiel.
- So oder so, wir bekommen die Liste `[0, 2, 4, 6, 8]`.
- Nun spielen wir noch mit verschachtelter Comprehension.
- Sagen wir, dass wir zwei `Iterables` habe und alle möglichen Kombinationen ihres Output produzieren wollen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- So oder so, wir bekommen die Liste `[0, 2, 4, 6, 8]`.
- Nun spielen wir noch mit verschachtelter Comprehension.
- Sagen wir, dass wir zwei `Iterables` habe und alle möglichen Kombinationen ihres Output produzieren wollen.
- Nebenbei: Strings sind auch `Iterables` ... wir können über ihre Zeichen iterieren.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- So oder so, wir bekommen die Liste `[0, 2, 4, 6, 8]`.
- Nun spielen wir noch mit verschachtelter Comprehension.
- Sagen wir, dass wir zwei `Iterables` habe und alle möglichen Kombinationen ihres Output produzieren wollen.
- Nebenbei: Strings sind auch `Iterables` ... wir können über ihre Zeichen iterieren.
- Nehmen wir an, wir haben eine erste Sequenz `"abc"` und die zweite ist `"xy"`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Nun spielen wir noch mit verschachtelter Comprehension.
- Sagen wir, dass wir zwei `Iterables` habe und alle möglichen Kombinationen ihres Output produzieren wollen.
- Nebenbei: Strings sind auch `Iterables` ... wir können über ihre Zeichen iterieren.
- Nehmen wir an, wir haben eine erste Sequenz `"abc"` und die zweite ist `"xy"`.
- Wir können wir eine Liste mit allen möglichen Paaren bauen, die jeweils ein Zeichen aus jedem der beiden Strings beinhalten?

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Sagen wir, dass wir zwei `Iterables` habe und alle möglichen Kombinationen ihres Output produzieren wollen.
- Nebenbei: Strings sind auch `Iterables` ... wir können über ihre Zeichen iterieren.
- Nehmen wir an, wir haben eine erste Sequenz `"abc"` und die zweite ist `"xy"`.
- Wir können wir eine Liste mit allen möglichen Paaren bauen, die jeweils ein Zeichen aus jedem der beiden Strings beinhalten?
- Einfach: in dem wir zwei `for` Statements schreiben!

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Nebenbei: Strings sind auch `Iterables` ... wir können über ihre Zeichen iterieren.
- Nehmen wir an, wir haben eine erste Sequenz `"abc"` und die zweite ist `"xy"`.
- Wir können wir eine Liste mit allen möglichen Paaren bauen, die jeweils ein Zeichen aus jedem der beiden Strings beinhalten?
- Einfach: in dem wir zwei `for` Statements schreiben!
- Wir schreiben `[f"{m}{n}"for m in "abc"for n in "xy"]`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Nehmen wir an, wir haben eine erste Sequenz `"abc"` und die zweite ist `"xy"`.
- Wir können wir eine Liste mit allen möglichen Paaren bauen, die jeweils ein Zeichen aus jedem der beiden Strings beinhalten?
- Einfach: in dem wir zwei `for` Statements schreiben!
- Wir schreiben `[f"{m}{n}"for m in "abc"for n in "xy"]`.
- Die Variable `m` nimmt als Wert alle Zeichen aus dem String `"abc"` nacheinander an.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Wir können wir eine Liste mit allen möglichen Paaren bauen, die jeweils ein Zeichen aus jedem der beiden Strings beinhalten?
- Einfach: in dem wir zwei `for` Statements schreiben!
- Wir schreiben `[f"{m}{n}"for m in "abc"for n in "xy"]`.
- Die Variable `m` nimmt als Wert alle Zeichen aus dem String `"abc"` nacheinander an.
- Für jeden Wert, den `m` annimmt, iteriert die Variable `n` über `"xy"` and und wird daher erst `"x"` und dann `"y"`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Einfach: in dem wir zwei `for` Statements schreiben!
- Wir schreiben `[f"{m}{n}"for m in "abc"for n in "xy"]`.
- Die Variable `m` nimmt als Wert alle Zeichen aus dem String `"abc"` nacheinander an.
- Für jeden Wert, den `m` annimmt, iteriert die Variable `n` über `"xy"` and und wird daher erst `"x"` und dann `"y"`.
- Der f-String `f"{m}{n}"` wird für jede Kombination von `m` und `n` interpoliert.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Wir schreiben `[f"{m}{n}"for m in "abc"for n in "xy"]`.
- Die Variable `m` nimmt als Wert alle Zeichen aus dem String `"abc"` nacheinander an.
- Für jeden Wert, den `m` annimmt, iteriert die Variable `n` über `"xy"` and und wird daher erst `"x"` und dann `"y"`.
- Der f-String `f"{m}{n}"` wird für jede Kombination von `m` und `n` interpoliert.
- Als Ergebnis bekommen wir die Liste `["ax", "ay", "bx", "by", "cx", "cy"]`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Die Variable `m` nimmt als Wert alle Zeichen aus dem String `"abc"` nacheinander an.
- Für jeden Wert, den `m` annimmt, iteriert die Variable `n` über `"xy"` and und wird daher erst `"x"` und dann `"y"`.
- Der f-String `f"{m}{n}"` wird für jede Kombination von `m` und `n` interpoliert.
- Als Ergebnis bekommen wir die Liste `["ax", "ay", "bx", "by", "cx", "cy"]`.
- Natürlich können wir Listen beliebiger Datentypen mit List Comprehension erstellen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Für jeden Wert, den `m` annimmt, iteriert die Variable `n` über `"xy"` and und wird daher erst `"x"` und dann `"y"`.
- Der f-String `f"{m}{n}"` wird für jede Kombination von `m` und `n` interpoliert.
- Als Ergebnis bekommen wir die Liste `["ax", "ay", "bx", "by", "cx", "cy"]`.
- Natürlich können wir Listen beliebiger Datentypen mit List Comprehension erstellen.
- Dies beinhaltet Listen anderer Listen, Listen von Tupeln, Mengen, oder Dictionaries ... was immer wir wollen.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Der f-String `f"{m}{n}"` wird für jede Kombination von `m` und `n` interpoliert.
- Als Ergebnis bekommen wir die Liste `["ax", "ay", "bx", "by", "cx", "cy"]`.
- Natürlich können wir Listen beliebiger Datentypen mit List Comprehension erstellen.
- Dies beinhaltet Listen anderer Listen, Listen von Tupeln, Mengen, oder Dictionaries . . . was immer wir wollen.
- Wir wiederholen den selben Ansatz wie oben und speichern die Buchstabenkombinationen diesmal als Tupel.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Als Ergebnis bekommen wir die Liste `["ax", "ay", "bx", "by", "cx", "cy"]`.
- Natürlich können wir Listen beliebiger Datentypen mit List Comprehension erstellen.
- Dies beinhaltet Listen anderer Listen, Listen von Tupeln, Mengen, oder Dictionaries . . . was immer wir wollen.
- Wir wiederholen den selben Ansatz wie oben und speichern die Buchstabenkombinationen diesmal als Tupel.
- `[(o, p)for o in "abc"for p in "xy"]` macht genau das.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  # ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Beispiel

- Dies beinhaltet Listen anderer Listen, Listen von Tupeln, Mengen, oder Dictionaries . . . was immer wir wollen.
- Wir wiederholen den selben Ansatz wie oben und speichern die Buchstabenkombinationen diesmal als Tupel.
- `[(o, p)for o in "abc"for p in "xy"]` macht genau das.
- Dieser Ausdruck produziert `[("a", "x"), ("a", "y"), ("b", "x"), ("b", "y"), ("c", "x"), ("c", "y")]`.

```python
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ python3 simple_list_comprehension.py ↓

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b
    ↪ ', 'y'), ('c', 'x'), ('c', 'y')]
```

Performanz

Was sagt Ruff dazu?

- Wenden wir Ruff auf unser Beispielprogramm `simple_list_comprehension.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,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Wenden wir Ruff auf unser Beispielprogramm `simple_list_comprehension.py` an.
- Ruff sieht die Konstruktion der Liste über die `append`-Funktion in einer Schleife als *Performanz-Problem* an, was durch die Kennung `PERF` ausgedrückt wird.

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Wenden wir Ruff auf unser Beispielprogramm `simple_list_comprehension.py` an.
- Ruff sieht die Konstruktion der Liste über die `append`-Funktion in einer Schleife als *Performanz-Problem* an, was durch die Kennung `PERF` ausgedrückt wird.
- Das ist natürlich nur ein lösbares Problem, wenn wir auch eine andere Möglichkeit haben, die Liste zu erstellen.

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Wenden wir Ruff auf unser Beispielprogramm `simple_list_comprehension.py` an.
- Ruff sieht die Konstruktion der Liste über die `append`-Funktion in einer Schleife als *Performanz-Problem* an, was durch die Kennung `PERF` ausgedrückt wird.
- Das ist natürlich nur ein lösbares Problem, wenn wir auch eine andere Möglichkeit haben, die Liste zu erstellen.
- Nun, wir kennen eine andere Möglichkeit.

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Ruff sieht die Konstruktion der Liste über die `append`-Funktion in einer Schleife als *Performanz-Problem* an, was durch die Kennung `PERF` ausgedrückt wird.
- Das ist natürlich nur ein lösbares Problem, wenn wir auch eine andere Möglichkeit haben, die Liste zu erstellen.
- Nun, wir kennen eine andere Möglichkeit.
- Wir können die Liste via List Comprehension erstellen, was aber nicht immer geht.

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Das ist natürlich nur ein lösbares Problem, wenn wir auch eine andere Möglichkeit haben, die Liste zu erstellen.
- Nun, wir kennen eine andere Möglichkeit.
- Wir können die Liste via List Comprehension erstellen, was aber nicht immer geht.
- Wenn es geht, was in unserem Beispiel ja der Fall ist, dann sehen wir oft, dass List Comprehension kompakter ist.

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Nun, wir kennen eine andere Möglichkeit.
- Wir können die Liste via List Comprehension erstellen, was aber nicht immer geht.
- Wenn es geht, was in unserem Beispiel ja der Fall ist, dann sehen wir oft, dass List Comprehension kompakter ist.
- Kode, der kompakter ist, ist oft lesbarer und daher aus Sicht eines Softwareingenieurs, oftmals zu bevorzugen.

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Was sagt Ruff dazu?

- Nun, wir kennen eine andere Möglichkeit.
- Wir können die Liste via List Comprehension erstellen, was aber nicht immer geht.
- Wenn es geht, was in unserem Beispiel ja der Fall ist, dann sehen wir oft, dass List Comprehension kompakter ist.
- Kode, der kompakter ist, ist oft lesbarer und daher aus Sicht eines Softwareingenieurs, oftmals zu bevorzugen.
- Aber warum wäre die ursprüngliche Schleife ein Performanz-Problem, also irgendwie langsam?

```
$ 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 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
  |
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to
    ↪ 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
  |
help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.14.2 failed with exit code 1.
```

Prüfen wir das nach

- OK, prüfen wir das nach.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.
- Die Methode, die schneller ist, hat die bessere Performanz.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.
- Die Methode, die schneller ist, hat die bessere Performanz.
- Nun, die Laufzeit, die etwas braucht, zu messen ist immer ein bischen schwierig.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.
- Die Methode, die schneller ist, hat die bessere Performanz.
- Nun, die Laufzeit, die etwas braucht, zu messen ist immer ein bischen schwierig.
- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.
- Die Methode, die schneller ist, hat die bessere Performanz.
- Nun, die Laufzeit, die etwas braucht, zu messen ist immer ein bischen schwierig.
- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.
- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].

Prüfen wir das nach

- OK, prüfen wir das nach.
- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.
- Die Methode, die schneller ist, hat die bessere Performanz.
- Nun, die Laufzeit, die etwas braucht, zu messen ist immer ein bischen schwierig.
- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.
- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].
- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.

Prüfen wir das nach

- Um diese Sache zu untersuchen, könnten wir versuchen, zwei Listem mit dem selben Inhalt zu erstellen.
- Eine erstellen wir, in dem wir die `append`-Methode in einer Schleife aufrufen, die andere mit List Comprehension.
- Also im Grunde das, was wir gerade gemacht haben.
- Die Methode, die schneller ist, hat die bessere Performanz.
- Nun, die Laufzeit, die etwas braucht, zu messen ist immer ein bischen schwierig.
- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.
- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].
- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.

Prüfen wir das nach

- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.
- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].
- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.

Nützliches Werkzeug

`timeit` ist ein Werkzeug um die Laufzeit von kleinen Kodestücken zu messen.

Prüfen wir das nach

- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.
- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].
- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.

Nützliches Werkzeug

`timeit` ist ein Werkzeug um die Laufzeit von kleinen Kodestücken zu messen. Es ist direkt Teil von Python.

Prüfen wir das nach

- Die Laufzeit eines Python-Programms hängt klar von der CPU ab, auf der es läuft.
- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].
- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.

Nützliches Werkzeug

`timeit` ist ein Werkzeug um die Laufzeit von kleinen Kodestücken zu messen. Es ist direkt Teil von Python. Dieses Modul vermeidet eine Menge von Fallen, in die man beim Zeit-Messen oftmals hereintappt[28,36].

Prüfen wir das nach

- Sie wird auch vom Betriebssystem, dem zur Verfügung stehenden RAM, der Festplattengeschwindigkeit, und natürlich von anderen Prozessen, die zur selben Zeit auf dem Computer laufen, beeinflusst[41].
- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.
- `timeit` erlaubt uns also, die Laufzeit eines Statements zu messen.

Nützliches Werkzeug

`timeit` ist ein Werkzeug um die Laufzeit von kleinen Kodestücken zu messen. Es ist direkt Teil von Python. Dieses Modul vermeidet eine Menge von Fallen, in die man beim Zeit-Messen oftmals hereintappt[28,36].

Prüfen wir das nach

- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.
- `timeit` erlaubt uns also, die Laufzeit eines Statements zu messen.
- Wir wollen messen, wie lange es dauert, eine Liste mit den geraden Zahlen aus 0..1 000 000 zu erstellen.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Natürlich hängt sie auch von der Version des Python-Iterpreters ab und das Ergebnis der Messung könnte also nach jedem Systemupdate anders sein.
- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.
- `timeit` erlaubt uns also, die Laufzeit eines Statements zu messen.
- Wir wollen messen, wie lange es dauert, eine Liste mit den geraden Zahlen aus 0..1 000 000 zu erstellen.
- Wir implementieren dafür zuerst eine Funktion `create_by_append`, die die Liste mit der `append`-Methode in einer Schleife erstellt.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Was immer wir messen, wir müssen sehr vorsichtig sein ... aber wir probieren es trotzdem.
- `timeit` erlaubt uns also, die Laufzeit eines Statements zu messen.
- Wir wollen messen, wie lange es dauert, eine Liste mit den geraden Zahlen aus 0..1 000 000 zu erstellen.
- Wir implementieren dafür zuerst eine Funktion `create_by_append`, die die Liste mit der `append`-Methode in einer Schleife erstellt.
- In einer Schleife über `range(1_000_001)` fügen wir die geraden Zahlen an die Liste `numbers` an.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- `timeit` erlaubt uns also, die Laufzeit eines Statements zu messen.
- Wir wollen messen, wie lange es dauert, eine Liste mit den geraden Zahlen aus 0..1 000 000 zu erstellen.
- Wir implementieren dafür zuerst eine Funktion `create_by_append`, die die Liste mit der `append`-Methode in einer Schleife erstellt.
- In einer Schleife über `range(1_000_001)` fügen wir die geraden Zahlen an die Liste `numbers` an.
- Am Ende wird die Liste zurückgeliefert.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Wir implementieren dafür zuerst eine Funktion `create_by_append`, die die Liste mit der `append`-Methode in einer Schleife erstellt.
- In einer Schleife über `range(1_000_001)` fügen wir die geraden Zahlen an die Liste `numbers` an.
- Am Ende wird die Liste zurückgeliefert.
- Um die Laufzeit dieser Funktion zu messen, importieren wir zuerst die Funktion `repeat` aus dem Modul `timeit`.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- In einer Schleife über `range(1_000_001)` fügen wir die geraden Zahlen an die Liste `numbers` an.
- Am Ende wird die Liste zurückgeliefert.
- Um die Laufzeit dieser Funktion zu messen, importieren wir zuerst die Funktion `repeat` aus dem Modul `timeit`.
- Wir sagen `repeat`, dass es unsere Funktion `create_by_append` einmal (`number=1`) aufrufen und die gemessen Zeit zurückliefern soll.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- In einer Schleife über `range(1_000_001)` fügen wir die geraden Zahlen an die Liste `numbers` an.
- Am Ende wird die Liste zurückgeliefert.
- Um die Laufzeit dieser Funktion zu messen, importieren wir zuerst die Funktion `repeat` aus dem Modul `timeit`.
- Wir sagen `repeat`, dass es unsere Funktion `create_by_append` einmal (`number=1`) aufrufen und die gemessen Zeit zurückliefern soll.
- Es wird die Laufzeit dann in Sekunden als `float` liefern.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Um die Laufzeit dieser Funktion zu messen, importieren wir zuerst die Funktion `repeat` aus dem Modul `timeit`.
- Wir sagen `repeat`, dass es unsere Funktion `create_by_append` einmal (`number=1`) aufrufen und die gemessen Zeit zurückliefern soll.
- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Wir sagen `repeat`, dass es unsere Funktion `create_by_append` einmal (`number=1`) aufrufen und die gemessen Zeit zurückliefern soll.
- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

Die Dokumentation von `timeit` sagt[36]:

> *Note: it's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the* `min()` *of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.*
>
> — *[36], 2001*

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

Gute Praxis

Wenn wir die Laufzeit von Kode für bestimmte Inputs prüfen, dann ergibt es Sinn, mehrere Messungen zu machen und das Minimum der beobachteten Ergebnisse zu nehmen[36].

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

Gute Praxis

Wenn wir die Laufzeit von Kode für bestimmte Inputs prüfen, dann ergibt es Sinn, mehrere Messungen zu machen und das Minimum der beobachteten Ergebnisse zu nehmen[36]. Der Grund ist, das viele Faktoren (CPU-Temperatur, andere Prozesse, ...) die Laufzeit negativ beeinflussen können.

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

Gute Praxis

Wenn wir die Laufzeit von Kode für bestimmte Inputs prüfen, dann ergibt es Sinn, mehrere Messungen zu machen und das Minimum der beobachteten Ergebnisse zu nehmen[36]. Der Grund ist, das viele Faktoren (CPU-Temperatur, andere Prozesse, ...) die Laufzeit negativ beeinflussen können. Es gibt aber keinen Faktor, der unseren Kode schneller machen kann, als die Hardware zulässt.

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

Gute Praxis

Wenn wir die Laufzeit von Kode für bestimmte Inputs prüfen, dann ergibt es Sinn, mehrere Messungen zu machen und das Minimum der beobachteten Ergebnisse zu nehmen[36]. Der Grund ist, das viele Faktoren (CPU-Temperatur, andere Prozesse, ...) die Laufzeit negativ beeinflussen können. Es gibt aber keinen Faktor, der unseren Kode schneller machen kann, als die Hardware zulässt. Also gibt uns das Minimum den akuratesten Eindruck darauf, wie schnell unser Kode auf unserem Computer laufen kann.

Prüfen wir das nach

- Es wird die Laufzeit dann in Sekunden als `float` liefern.
- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.

Gute Praxis

Wenn wir die Laufzeit von Kode für bestimmte Inputs prüfen, dann ergibt es Sinn, mehrere Messungen zu machen und das Minimum der beobachteten Ergebnisse zu nehmen[36]. Der Grund ist, das viele Faktoren (CPU-Temperatur, andere Prozesse, ...) die Laufzeit negativ beeinflussen können. Es gibt aber keinen Faktor, der unseren Kode schneller machen kann, als die Hardware zulässt. Also gibt uns das Minimum den akuratesten Eindruck darauf, wie schnell unser Kode auf unserem Computer laufen kann. Beachte, dass Effekte wie Caching immer noch unsere Messwerte verunreinigen können.

Prüfen wir das nach

- Allerdings wäre eine einzige Messung nicht sehr verlässlich, sie könnte durch das Scheduling durch das Betriebssystem oder durch Garbage Collection vom Python-Interpreter verunreinigt werden[28].
- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.
- Also machen wir genau das.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Deshalb sagen wir der `repeat`-Funktion, 90 solcher Messungen durchzuführen (Argument `repeat=90`).
- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.
- Also machen wir genau das.
- `repeat` gibt uns eine Liste mit gemessenen Laufzeiten.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.
- Also machen wir genau das.
- `repeat` gibt uns eine Liste mit gemessenen Laufzeiten.
- Die Funktion `min` akzeptiert eine Sequenz von Elementen und gibt uns das kleinste Element daraus zurück.[4]

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Dadurch bekommen wir alle gemessenen Laufzeiten als `list[float]`.
- Also machen wir genau das.
- `repeat` gibt uns eine Liste mit gemessenen Laufzeiten.
- Die Funktion `min` akzeptiert eine Sequenz von Elementen und gibt uns das kleinste Element daraus zurück.[4]
- Also drucken wir das Ergebnis von `min` angewandt auf die von `repeat` zurückgelieferte Liste.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Also machen wir genau das.
- `repeat` gibt uns eine Liste mit gemessenen Laufzeiten.
- Die Funktion `min` akzeptiert eine Sequenz von Elementen und gibt uns das kleinste Element daraus zurück.[4]
- Also drucken wir das Ergebnis von `min` angewandt auf die von `repeat` zurückgelieferte Liste.
- Wir formatieren die Ausgabe als Millisekunden gerundet auf drei Dezimalstellen, damit es etwas lesbarer wird.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- `repeat` gibt uns eine Liste mit gemessenen Laufzeiten.
- Die Funktion `min` akzeptiert eine Sequenz von Elementen und gibt uns das kleinste Element daraus zurück.[4]
- Also drucken wir das Ergebnis von `min` angewandt auf die von `repeat` zurückgelieferte Liste.
- Wir formatieren die Ausgabe als Millisekunden gerundet auf drei Dezimalstellen, damit es etwas lesbarer wird.
- Nun wird das Portable Document Format (PDF)-Dokument dieser Slides automatisch von einer GitHub Action[5] gebaut.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Die Funktion `min` akzeptiert eine Sequenz von Elementen und gibt uns das kleinste Element daraus zurück.[4]
- Also drucken wir das Ergebnis von `min` angewandt auf die von `repeat` zurückgelieferte Liste.
- Wir formatieren die Ausgabe als Millisekunden gerundet auf drei Dezimalstellen, damit es etwas lesbarer wird.
- Nun wird das Portable Document Format (PDF)-Dokument dieser Slides automatisch von einer GitHub Action[5] gebaut.
- Diese führt alle Beispielprogramme aus und webt ihren Output in die Slides.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Also drucken wir das Ergebnis von `min` angewandt auf die von `repeat` zurückgelieferte Liste.
- Wir formatieren die Ausgabe als Millisekunden gerundet auf drei Dezimalstellen, damit es etwas lesbarer wird.
- Nun wird das Portable Document Format (PDF)-Dokument dieser Slides automatisch von einer GitHub Action[5] gebaut.
- Diese führt alle Beispielprogramme aus und webt ihren Output in die Slides.
- Dies wird wiederholt, jedesmal wenn ich die Slides update.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Wir formatieren die Ausgabe als Millisekunden gerundet auf drei Dezimalstellen, damit es etwas lesbarer wird.
- Nun wird das Portable Document Format (PDF)-Dokument dieser Slides automatisch von einer GitHub Action[5] gebaut.
- Diese führt alle Beispielprogramme aus und webt ihren Output in die Slides.
- Dies wird wiederholt, jedesmal wenn ich die Slides update.
- Das bedeutet, dass wenn ich diesen Text schreibe, ich nicht wissen kann, welchen Wert Sie letztendlich sehen werden.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Nun wird das Portable Document Format (PDF)-Dokument dieser Slides automatisch von einer GitHub Action[5] gebaut.
- Diese führt alle Beispielprogramme aus und webt ihren Output in die Slides.
- Dies wird wiederholt, jedesmal wenn ich die Slides update.
- Das bedeutet, dass wenn ich diesen Text schreibe, ich nicht wissen kann, welchen Wert Sie letztendlich sehen werden.
- Auf meiner lokalen Maschine habe ich `runtime/call: 30.1 ms.` bekommen.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Diese führt alle Beispielprogramme aus und webt ihren Output in die Slides.
- Dies wird wiederholt, jedesmal wenn ich die Slides update.
- Das bedeutet, dass wenn ich diesen Text schreibe, ich nicht wissen kann, welchen Wert Sie letztendlich sehen werden.
- Auf meiner lokalen Maschine habe ich `runtime/call: 30.1 ms.` bekommen.
- Egal.

```python
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Dies wird wiederholt, jedesmal wenn ich die Slides update.
- Das bedeutet, dass wenn ich diesen Text schreibe, ich nicht wissen kann, welchen Wert Sie letztendlich sehen werden.
- Auf meiner lokalen Maschine habe ich `runtime/call: 30.1 ms.` bekommen.
- Egal.
- Um nun zu testen, ob List Comprehension nun wirklich schneller als die iterative Listenkonstruktion via `append` ist, schreiben wir Programm `list_of_numbers_comprehension.py`.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ python3 list_of_numbers_comprehension.py ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Auf meiner lokalen Maschine habe ich `runtime/call: 30.1 ms.` bekommen.
- Egal.
- Um nun zu testen, ob List Comprehension nun wirklich schneller als die iterative Listenkonstruktion via `append` ist, schreiben wir Programm `list_of_numbers_comprehension.py`.
- Dieses Programm macht fast das gleiche wie Programm `list_of_numbers_append.py` von eben.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Egal.
- Um nun zu testen, ob List Comprehension nun wirklich schneller als die iterative Listenkonstruktion via `append` ist, schreiben wir Programm `list_of_numbers_comprehension.py`.
- Dieses Programm macht fast das gleiche wie Programm `list_of_numbers_append.py` von eben.
- Es definiert eine Funktion `create_by_comprehension`, die die selbe Liste wie `create_by_append` erstellt.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Egal.
- Um nun zu testen, ob List Comprehension nun wirklich schneller als die iterative Listenkonstruktion via `append` ist, schreiben wir Programm `list_of_numbers_comprehension.py`.
- Dieses Programm macht fast das gleiche wie Programm `list_of_numbers_append.py` von eben.
- Es definiert eine Funktion `create_by_comprehension`, die die selbe Liste wie `create_by_append` erstellt.
- Nur das die Funktion List Comprehension verwendet.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Dieses Programm macht fast das gleiche wie Programm `list_of_numbers_append.py` von eben.
- Es definiert eine Funktion `create_by_comprehension`, die die selbe Liste wie `create_by_append` erstellt.
- Nur das die Funktion List Comprehension verwendet.
- Wir messen die Laufzeit dieser Funktion genauso wie vorhin.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Dieses Programm macht fast das gleiche wie Programm `list_of_numbers_append.py` von eben.
- Es definiert eine Funktion `create_by_comprehension`, die die selbe Liste wie `create_by_append` erstellt.
- Nur das die Funktion List Comprehension verwendet.
- Wir messen die Laufzeit dieser Funktion genauso wie vorhin.
- Natürlich ist das Ergebnis jedesmal anders, wenn ich die Slides update.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Es definiert eine Funktion `create_by_comprehension`, die die selbe Liste wie `create_by_append` erstellt.
- Nur das die Funktion List Comprehension verwendet.
- Wir messen die Laufzeit dieser Funktion genauso wie vorhin.
- Natürlich ist das Ergebnis jedesmal anders, wenn ich die Slides update.
- Auf meiner lokalen Machine habe ich `runtime/call: 28.7 ms.` bekommen.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Nur das die Funktion List Comprehension verwendet.
- Wir messen die Laufzeit dieser Funktion genauso wie vorhin.
- Natürlich ist das Ergebnis jedesmal anders, wenn ich die Slides update.
- Auf meiner lokalen Machine habe ich `runtime/call: 28.7 ms.` bekommen.
- Das bestätigt, das List Comprehension tatsächlich etwas schneller als iterative Konstruktion ist.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ python3 list_of_numbers_comprehension.py ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Wir messen die Laufzeit dieser Funktion genauso wie vorhin.
- Natürlich ist das Ergebnis jedesmal anders, wenn ich die Slides update.
- Auf meiner lokalen Machine habe ich `runtime/call: 28.7 ms.` bekommen.
- Das bestätigt, das List Comprehension tatsächlich etwas schneller als iterative Konstruktion ist.
- In älteren Python-Versionen ist der Unterschied größer.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ python3 list_of_numbers_comprehension.py ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Natürlich ist das Ergebnis jedesmal anders, wenn ich die Slides update.
- Auf meiner lokalen Machine habe ich `runtime/call: 28.7 ms.` bekommen.
- Das bestätigt, das List Comprehension tatsächlich etwas schneller als iterative Konstruktion ist.
- In älteren Python-Versionen ist der Unterschied größer.
- Aber er ist auch heute noch da und messbar.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Prüfen wir das nach

- Auf meiner lokalen Machine habe ich `runtime/call: 28.7 ms.` bekommen.
- Das bestätigt, das List Comprehension tatsächlich etwas schneller als iterative Konstruktion ist.
- In älteren Python-Versionen ist der Unterschied größer.
- Aber er ist auch heute noch da und messbar.
- Aber selbst wenn es keinen Unterschied gäbe ... List Comprehension ist kürzerer und eleganterer Kode.

```python
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ python3 list_of_numbers_comprehension.py ↓

```
==== list comprehension ====
runtime/call: 55.4 ms.
```

```
==== iterative list construction via append ====
runtime/call: 60.7 ms.
```

Zusammenfassung

Zusammenfassung

- Mit List Comprehension haben wir eine elegante, compakte, und mächtige Methode zum Erstellen von Listen kennengelernt.

Zusammenfassung

- Mit List Comprehension haben wir eine elegante, compakte, und mächtige Methode zum Erstellen von Listen kennengelernt.
- Sie generalisiert die Idee von Listen-Literalen und kombiniert sie mit verschachtelbaren `for`-Schleifen.

Zusammenfassung

- Mit List Comprehension haben wir eine elegante, compakte, und mächtige Methode zum Erstellen von Listen kennengelernt.
- Sie generalisiert die Idee von Listen-Literalen und kombiniert sie mit verschachtelbaren `for`-Schleifen.
- Nicht nur das diese Methode, Listen zu konstruieren, kompakter ist als Elemente in einer Schleife an eine Liste anzufügen, sie ist auch noch schneller.

Zusammenfassung

- Mit List Comprehension haben wir eine elegante, compakte, und mächtige Methode zum Erstellen von Listen kennengelernt.
- Sie generalisiert die Idee von Listen-Literalen und kombiniert sie mit verschachtelbaren `for`-Schleifen.
- Nicht nur das diese Methode, Listen zu konstruieren, kompakter ist als Elemente in einer Schleife an eine Liste anzufügen, sie ist auch noch schneller.
- Comprehension lässt sich auch auf ein paar andere Kollektionstypen anwenden...

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] David Lee Applegate, Robert E. Bixby, Vašek Chvátal und William John Cook. *The Traveling Salesman Problem: A Computational Study*. 2. Aufl. Bd. 17 der Reihe Princeton Series in Applied Mathematics. Princeton, NJ, USA: Princeton University Press, 2007. ISBN: 978-0-691-12993-8 (siehe S. 127).

[2] Jacek Błażewicz, Wolfgang Domschke und Erwin Pesch. "The Job Shop Scheduling Problem: Conventional and New Solution Techniques". *European Journal of Operational Research* 93(1):1–33, Aug. 1996. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/0377-2217(95)00362-2 (siehe S. 126).

[3] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 126).

[4] "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. 73–99).

[5] Eric Chapman. *GitHub Actions*. Birmingham, England, UK: Packt Publishing Ltd, März 2024. ISBN: 978-1-80512-862-5 (siehe S. 73–102).

[6] Bo Chen, Chris N. Potts und Gerhard J. Woeginger. "A Review of Machine Scheduling: Complexity, Algorithms and Approximability". In: *Handbook of Combinatorial Optimization*. Hrsg. von Panos Miltiades Pardalos, Ding-Zhu Du und Ronald Lewis Graham. 1. Aufl. Boston, MA, USA: Springer, 1998, S. 1493–1641. ISBN: 978-1-4613-7987-4. doi:10.1007/978-1-4613-0303-9_25. See also pages 21–169 in volume 3/3 by Norwell, MA, USA: Kluwer Academic Publishers. (Siehe S. 126, 128).

[7] Stephen Arthur Cook. "The Complexity of Theorem-Proving Procedures". In: *Third Annual ACM Symposium on Theory of Computing (STOC'1971)*. 3.–5. Mai 1971, Shaker Heights, OH, USA. Hrsg. von Michael A. Harrison, Ranan B. Banerji und Jeffrey D. Ullman. New York, NY, USA: Association for Computing Machinery (ACM), 1971, S. 151–158. ISBN: 978-1-4503-7464-4. doi:10.1145/800157.805047 (siehe S. 128).

[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. 126).

[9] *PDF 32000-1:2008 – Document Management – Portable Document Format – Part 1: PDF 1.7*. 1. Aufl. San Jose, CA, USA: Adobe Systems Incorporated, 1. Juli 2008. URL: https://pdf-lib.js.org/assets/with_large_page_count.pdf (besucht am 2024-12-12) (siehe S. 127).

References II

[10] "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. 126).

[11] 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. 126).

[12] Michael T. Goodrich. *A Gentle Introduction to NP-Completeness*. Irvine, CA, USA: University of California, Irvine, Apr. 2022. URL: https://ics.uci.edu/~goodrich/teach/cs165/notes/NPComplete.pdf (besucht am 2025-08-01) (siehe S. 128).

[13] 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. 126).

[14] Gregory Z. Gutin und Abraham P. Punnen, Hrsg. *The Traveling Salesman Problem and its Variations*. Bd. 12. Combinatorial Optimization (COOP). New York, NY, USA: Springer New York, Mai 2002. ISSN: 1388-3011. doi:10.1007/b101971 (siehe S. 127).

[15] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 127).

[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. 126).

[17] 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. 127).

References III

[18] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan und David B. Shmoys. "Sequencing and Scheduling: Algorithms and Complexity". In: *Production Planning and Inventory*. Hrsg. von Stephen C. Graves, Alexander Hendrik George Rinnooy Kan und Paul H. Zipkin. Bd. IV der Reihe Handbooks of Operations Research and Management Science. Amsterdam, The Netherlands: Elsevier B.V., 1993. Kap. 9, S. 445–522. ISSN: 0927-0507. ISBN: 978-0-444-87472-6. doi:10.1016/S0927-0507(05)80189-6. URL: http://alexandria.tue.nl/repository/books/339776.pdf (besucht am 2023-12-06) (siehe S. 126, 128).

[19] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan und David B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Estimation, Simulation, and Control – Wiley-Interscience Series in Discrete Mathematics and Optimization. Chichester, West Sussex, England, UK: Wiley Interscience, Sep. 1985. ISSN: 0277-2698. ISBN: 978-0-471-90413-7 (siehe S. 127).

[20] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 127).

[21] Michael Lee, Ivan Levkivskyi und Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), 14. März 2019. URL: https://peps.python.org/pep-0586 (besucht am 2024-12-17) (siehe S. 126).

[22] 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. 127).

[23] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 127).

[24] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 127).

[25] 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. 127).

[26] 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. 126).

References IV

[27] 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. 126).

[28] Tim Peters. "Algorithms – Introduction". In: *Python Cookbook*. Hrsg. von David Ascher. 1. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Juli 2002. Kap. 17. ISBN: 978-0-596-00167-4. URL: https://www.oreilly.com/library/view/python-cookbook/0596001673/ch17.html (besucht am 2024-11-07) (siehe S. 73–93).

[29] *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. 126).

[30] Sanatan Rai und George Vairaktarakis. "NP-Complete Problems and Proof Methodology". In: *Encyclopedia of Optimization*. Hrsg. von Christodoulos A. Floudas und Panos Miltiades Pardalos. 2. Aufl. Boston, MA, USA: Springer, Sep. 2008, S. 2675–2682. ISBN: 978-0-387-74758-3. doi:10.1007/978-0-387-74759-0_462 (siehe S. 128).

[31] 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. 126).

[32] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 126).

[33] 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. 126).

[34] *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).

References V

[35] .“Literals”. In: *Static Typing with Python*. Hrsg. von The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org/en/latest/spec/literal.html (besucht am 2025-08-29) (siehe S. 126).

[36] “`timeit` – Measure Execution Time of Small Code Snippets”. In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/timeit.html (besucht am 2024-11-07) (siehe S. 73–76, 87–92).

[37] 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. 126, 128).

[38] 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. 127).

[39] Barry Warsaw. *List Comprehensions*. Python Enhancement Proposal (PEP) 202. Beaverton, OR, USA: Python Software Foundation (PSF), 13. Juli 2000. URL: https://peps.python.org/pep-0202 (besucht am 2024-11-08) (siehe S. 5–10, 12–14).

[40] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 127).

[41] Thomas Weise (汤卫思), Raymond Chiong, Jörg Lässig, Ke Tang (唐珂), Shigeyoshi Tsutsui, Wenxiang Chen (陈文祥), Zbigniew „Zbyszek“ Michalewicz und Xin Yao (姚新). “Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem”. *IEEE Computational Intelligence Magazine (CIM)* 9(3):40–52, Aug. 2014. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1556-603X. doi:10.1109/MCI.2014.2326101 (siehe S. 63–76, 127).

[42] *What does PDF mean?* San Jose, CA, USA: Adobe Systems Incorporated, 2024. URL: https://www.adobe.com/acrobat/about-adobe-pdf.html (besucht am 2024-12-12) (siehe S. 127).

Glossary (in English) I

C is a programming language, which is very successful in system programming situations[8,29].

f-string let you include the results of expressions in strings[3,10,11,13,26,33]. 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 Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[32,37]. Learn more at `https://git-scm.com`.

GitHub is a website where software projects can be hosted and managed via the Git VCS[27,37]. Learn more at `https://github.com`.

JSSP The *Job Shop Scheduling Problem*[2,18] is one of the most prominent and well-studied scheduling tasks. In a JSSP instance, there are k machines and m jobs. Each job must be processed once by each machine in a job-specific sequence and has a job-specific processing time on each machine. The goal is to find an assignment of jobs to machines that results in an overall shortest makespan, i.e., the schedule which can complete all the jobs in the shortest time. The JSSP is $\mathcal{NP}$-complete[6,18].

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles[16,31]. Ruff is an example for a linter used in the Python world.

literal A literal is a specific concrete value, something that is written down as-is[21,35]. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

modulo division is, in Python, done by the operator `%` that computes the remainder of a division. `15 % 6` gives us `3`.

Glossary (in English) II

Mypy is a static type checking tool for Python[22] that makes use of type hints. Learn more at https://github.com/python/mypy and in[40].

PDF The *Portable Document Format*[9,42] is the format in which provide this book. It is the standard format for the exchange of documents in the internet.

Python The Python programming language[15,20,23,40], i.e., what you will learn about in our book[40]. Learn more at https://python.org.

Ruff is a linter and code formatting tool for Python[24,25]. Learn more at https://docs.astral.sh/ruff or in[40].

(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 for string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`.

TSP In an instance of the *Traveling Salesperson Problem*, also known as *Traveling Salesman Problem*, a set of n cities or locations as well as the distances between them are defined[1,14,19,41]. The goal is to find the shortest round-trip tour that starts at one city, visits all the other cities one time each, and returns to the origin. The TSP is one of the most well-known $\mathcal{NP}$-hard combinatorial optimization problems[14].

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[17,38]. 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.

Glossary (in English) III

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[37]. 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. For example, 5..9 is equivalent to $\{5, 6, 7, 8, 9\}$

$\mathcal{NP}$ is the class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic machine (such as a normal computer)[12].

$\mathcal{NP}$-complete A decision problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and all problems in $\mathcal{NP}$ are reducible to it in polynomial time[12,30]. A problem is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and if it is in $\mathcal{NP}$.

$\mathcal{NP}$-hard Algorithms that guarantee to find the correct solutions of $\mathcal{NP}$-hard problems[6,7,18] need a runtime that is exponential in the problem scale in the worst case. A problem is $\mathcal{NP}$-hard if all problems in $\mathcal{NP}$ are reducible to it in polynomial time[12].

$\mathbb{R}$ 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 $\mathbb{Z} \subset \mathbb{R}$.