合肥大學
HEFEI UNIVERSITY

Programming with Python

23. Schleifen mit `for`

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO) | 应用优化研究所
School of Artificial Intelligence and Big Data | 人工智能与大数据学院
Hefei University | 合肥大学
Hefei, Anhui, China | 中国安徽省合肥市

Version: 2026-01-29

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

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

Outline

Einleitung

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.
- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.
- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.
- *Schleifen* (EN: *loops*) erlauben uns genau das, nämlich eine Aktion mehrfach auszuführen.

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.
- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.
- *Schleifen* (EN: *loops*) erlauben uns genau das, nämlich eine Aktion mehrfach auszuführen.
- Im Prinzip erlauben uns Alternativen mit `if...else` bestimmte Stücke Kode auszulassen.

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.
- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.
- *Schleifen* (EN: *loops*) erlauben uns genau das, nämlich eine Aktion mehrfach auszuführen.
- Im Prinzip erlauben uns Alternativen mit `if...else` bestimmte Stücke Kode auszulassen.
- Mit anderen Worten, sie erlauben dem Kontrollfluss, diese Instruktionen zu überspringen und mit denen, die dann folgen, weiterzumachen.

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.
- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.
- *Schleifen* (EN: *loops*) erlauben uns genau das, nämlich eine Aktion mehrfach auszuführen.
- Im Prinzip erlauben uns Alternativen mit `if...else` bestimmte Stücke Kode auszulassen.
- Mit anderen Worten, sie erlauben dem Kontrollfluss, diese Instruktionen zu überspringen und mit denen, die dann folgen, weiterzumachen.
- Schleifen sind ein komplementäres Konzept und erlauben uns bestimmte Teile unserers Kodes zu wiederholen.

Einleitung

- Wenn wir mit Daten arbeiten, dann wollen wir einen Befehl nicht nur auf ein spezifisches Datum anwenden.
- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.
- *Schleifen* (EN: *loops*) erlauben uns genau das, nämlich eine Aktion mehrfach auszuführen.
- Im Prinzip erlauben uns Alternativen mit `if...else` bestimmte Stücke Kode auszulassen.
- Mit anderen Worten, sie erlauben dem Kontrollfluss, diese Instruktionen zu überspringen und mit denen, die dann folgen, weiterzumachen.
- Schleifen sind ein komplementäres Konzept und erlauben uns bestimmte Teile unserers Kodes zu wiederholen.
- Das ist im Grunde äquivalent dazu, dem Kontrollfluss zu erlauben, zurück an den Anfang des Kodes zu springen, den er gerade ausgeführt hat.

Einleitung

- Stattdessen wollen wir den Befehl wiederholt auf viele oder sogar alle Daten anwenden.
- *Schleifen* (EN: *loops*) erlauben uns genau das, nämlich eine Aktion mehrfach auszuführen.
- Im Prinzip erlauben uns Alternativen mit `if...else` bestimmte Stücke Kode auszulassen.
- Mit anderen Worten, sie erlauben dem Kontrollfluss, diese Instruktionen zu überspringen und mit denen, die dann folgen, weiterzumachen.
- Schleifen sind ein komplementäres Konzept und erlauben uns bestimmte Teile unserers Kodes zu wiederholen.
- Das ist im Grunde äquivalent dazu, dem Kontrollfluss zu erlauben, zurück an den Anfang des Kodes zu springen, den er gerade ausgeführt hat.
- Bevor wir aber richtig loslegen, lassen Sie uns kurz über die Relevanz von dem nachdenken, was jetzt kommt.

Einleitung

- Mit anderen Worten, sie erlauben dem Kontrollfluss, diese Instruktionen zu überspringen und mit denen, die dann folgen, weiterzumachen.
- Schleifen sind ein komplementäres Konzept und erlauben uns bestimmte Teile unserers Kodes zu wiederholen.
- Das ist im Grunde äquivalent dazu, dem Kontrollfluss zu erlauben, zurück an den Anfang des Kodes zu springen, den er gerade ausgeführt hat.
- Bevor wir aber richtig loslegen, lassen Sie uns kurz über die Relevanz von dem nachdenken, was jetzt kommt.

Definition: Structured Programming Theorem

Das *Structured Program Theorem* sagt aus, dass jede berechenbare Funktion mit nur drei verschiedenen Kontrollflusselementen berechnet werden, nämlich *(1)* dem sequenziellen Ausführen von Befehlen, *(2)* dem selektiven Ausführen von Befehlen (also mit Alternativen), und *(3)* dem iterativen (repetitiven) ausführen von Befehlen (bis eine bestimmte Bedingung eintrifft)[7,8,33,62].

Einleitung

- Bevor wir aber richtig loslegen, lassen Sie uns kurz über die Relevanz von dem nachdenken, was jetzt kommt.

Definition: Structured Programming Theorem

Das *Structured Program Theorem* sagt aus, dass jede berechenbare Funktion mit nur drei verschiedenen Kontrollflusselementen berechnet werden, nämlich *(1)* dem sequenziellen Ausführen von Befehlen, *(2)* dem selektiven Ausführen von Befehlen (also mit Alternativen), und *(3)* dem iterativen (repetitiven) ausführen von Befehlen (bis eine bestimmte Bedingung eintrifft)[7,8,33,62].

- Wir kennen bereits die ersten beiden Keontrollflusselemente.

Einleitung

- Bevor wir aber richtig loslegen, lassen Sie uns kurz über die Relevanz von dem nachdenken, was jetzt kommt.

Definition: Structured Programming Theorem

Das *Structured Program Theorem* sagt aus, dass jede berechenbare Funktion mit nur drei verschiedenen Kontrollflusselementen berechnet werden, nämlich *(1)* dem sequenziellen Ausführen von Befehlen, *(2)* dem selektiven Ausführen von Befehlen (also mit Alternativen), und *(3)* dem iterativen (repetitiven) ausführen von Befehlen (bis eine bestimmte Bedingung eintrifft)[7,8,33,62].

- Wir kennen bereits die ersten beiden Keontrollflusselemente.
- Und jetzt lernen Sie das dritte Element, nämlich Schleifen.

Einleitung

- Bevor wir aber richtig loslegen, lassen Sie uns kurz über die Relevanz von dem nachdenken, was jetzt kommt.

Definition: Structured Programming Theorem

Das *Structured Program Theorem* sagt aus, dass jede berechenbare Funktion mit nur drei verschiedenen Kontrollflusselementen berechnet werden, nämlich *(1)* dem sequenziellen Ausführen von Befehlen, *(2)* dem selektiven Ausführen von Befehlen (also mit Alternativen), und *(3)* dem iterativen (repetitiven) ausführen von Befehlen (bis eine bestimmte Bedingung eintrifft)[7,8,33,62].

- Wir kennen bereits die ersten beiden Keontrollflusselemente.
- Und jetzt lernen Sie das dritte Element, nämlich Schleifen.
- Danach sind wir im Prinzip in der Lage, *jede* Berechnung durchzuführen, die mit unseren Computern möglich ist.

Einleitung

- Bevor wir aber richtig loslegen, lassen Sie uns kurz über die Relevanz von dem nachdenken, was jetzt kommt.

Definition: Structured Programming Theorem

Das *Structured Program Theorem* sagt aus, dass jede berechenbare Funktion mit nur drei verschiedenen Kontrollflusselementen berechnet werden, nämlich *(1)* dem sequenziellen Ausführen von Befehlen, *(2)* dem selektiven Ausführen von Befehlen (also mit Alternativen), und *(3)* dem iterativen (repetitiven) ausführen von Befehlen (bis eine bestimmte Bedingung eintrifft)[7,8,33,62].

- Und jetzt lernen Sie das dritte Element, nämlich Schleifen.
- Danach sind wir im Prinzip in der Lage, *jede* Berechnung durchzuführen, die mit unseren Computern möglich ist.
- Alles was dann folgt sind nur Kontepte, die dafür a sind, unser Leben als Programmierer leichter zu machen.

Die `for`-Schleife

Die `for`-Schleife

- Die Syntax einer `for`-Schleife ist einfach[49].

Die `for`-Schleife

- Die Syntax einer `for`-Schleife ist einfach[49].

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Die Syntax einer `for`-Schleife ist einfach[49].
- Das Schlüsselwort `for` wird von der Schleifenvariable gefolgt.

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Die Syntax einer `for`-Schleife ist einfach[49].
- Das Schlüsselwort `for` wird von der Schleifenvariable gefolgt.
- Danach kommt das Schlüsselwort `in`, die Sequenz über die wir iterieren wollen und dann ein Doppelpunkt (`:`).

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Die Syntax einer `for`-Schleife ist einfach[49].
- Das Schlüsselwort `for` wird von der Schleifenvariable gefolgt.
- Danach kommt das Schlüsselwort `in`, die Sequenz über die wir iterieren wollen und dann ein Doppelpunkt (`:`).
- Die Variable wird iterativ jeden Wert in der Sequenz annehmen.

```
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Die Syntax einer `for`-Schleife ist einfach[49].
- Das Schlüsselwort `for` wird von der Schleifenvariable gefolgt.
- Danach kommt das Schlüsselwort `in`, die Sequenz über die wir iterieren wollen und dann ein Doppelpunkt (`:`).
- Die Variable wird iterativ jeden Wert in der Sequenz annehmen.
- Der Schleifenkörper ist der eingerückte Block nach dem Schleifenkopf.

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Danach kommt das Schlüsselwort `in`, die Sequenz über die wir iterieren wollen und dann ein Doppelpunkt (`:`).
- Die Variable wird iterativ jeden Wert in der Sequenz annehmen.
- Der Schleifenkörper ist der eingerückte Block nach dem Schleifenkopf.
- Er wird für jeden Wert ausgeführt, den die Schleifenvariable annimmt und kann auf den Wert über die Schleifenvariable zugreifen.

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Die Variable wird iterativ jeden Wert in der Sequenz annehmen.
- Der Schleifenkörper ist der eingerückte Block nach dem Schleifenkopf.
- Er wird für jeden Wert ausgeführt, den die Schleifenvariable annimmt und kann auf den Wert über die Schleifenvariable zugreifen.
- Jede Ausführung des Schleifenkörpers wird Iteration genannt.

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Die `for`-Schleife

- Der Schleifenkörper ist der eingerückte Block nach dem Schleifenkopf.
- Er wird für jeden Wert ausgeführt, den die Schleifenvariable annimmt und kann auf den Wert über die Schleifenvariable zugreifen.
- Jede Ausführung des Schleifenkörpers wird Iteration genannt.
- Nach dem Schleifenkörper folgt eine Leerzeile, nach der es dann mit normalem Kode weitergeht.

```python
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.
- In seiner einfachsten Form wird eine `for`-Schleife aber auf eine `range` von Ganzzahlen angewandt[49].

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.
- In seiner einfachsten Form wird eine `for`-Schleife aber auf eine `range` von Ganzzahlen angewandt[49].
- Solche „Ranges“ sind Squenzen die im Grunde wie Slices funktionieren, die wir schon von Indizieren von Strings, Listen, und Tupeln kennen.

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.
- In seiner einfachsten Form wird eine `for`-Schleife aber auf eine `range` von Ganzzahlen angewandt[49].
- Solche „Ranges“ sind Squenzen die im Grunde wie Slices funktionieren, die wir schon von Indizieren von Strings, Listen, und Tupeln kennen.
- `range(5)` gibt uns eine Sequenz von Ganzzahlen, die mit 0 anfängt, in Schritten von 1 ansteigt, und *vor* 5 aufhört, also im Grunde 0..4.

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.
- In seiner einfachsten Form wird eine `for`-Schleife aber auf eine `range` von Ganzzahlen angewandt[49].
- Solche „Ranges“ sind Squenzen die im Grunde wie Slices funktionieren, die wir schon von Indizieren von Strings, Listen, und Tupeln kennen.
- `range(5)` gibt uns eine Sequenz von Ganzzahlen, die mit 0 anfängt, in Schritten von 1 ansteigt, und *vor* 5 aufhört, also im Grunde 0..4.
- `range(6, 9)` gibt uns eine Sequenz von Ganzzahlen, die mit 6 anfängt, in Schritten von 1 ansteigt, und *vor* 9 aufhört, also im Grunde 6..8.

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.
- In seiner einfachsten Form wird eine `for`-Schleife aber auf eine `range` von Ganzzahlen angewandt[49].
- Solche „Ranges“ sind Squenzen die im Grunde wie Slices funktionieren, die wir schon von Indizieren von Strings, Listen, und Tupeln kennen.
- `range(5)` gibt uns eine Sequenz von Ganzzahlen, die mit 0 anfängt, in Schritten von 1 ansteigt, und *vor* 5 aufhört, also im Grunde 0..4.
- `range(6, 9)` gibt uns eine Sequenz von Ganzzahlen, die mit 6 anfängt, in Schritten von 1 ansteigt, und *vor* 9 aufhört, also im Grunde 6..8.
- `range(20, 27, 2)` ist eine Sequenz von Ganzzahlen, die mit 20 anfängt, in Schritten von 2 ansteigt, und *vor* 27 aufhört, also $(20, 22, 24, 26)$.

Ranges

- Wir kennen bereits einige Kollektions-Datentypen, die wir als `sequence` zum drüber-iterieren nehmen könnten, nämlich `lists`, `tuples`, `sets`, und `dicts`.
- In seiner einfachsten Form wird eine `for`-Schleife aber auf eine `range` von Ganzzahlen angewandt[49].
- Solche „Ranges“ sind Squenzen die im Grunde wie Slices funktionieren, die wir schon von Indizieren von Strings, Listen, und Tupeln kennen.
- `range(5)` gibt uns eine Sequenz von Ganzzahlen, die mit 0 anfängt, in Schritten von 1 ansteigt, und *vor* 5 aufhört, also im Grunde 0..4.
- `range(6, 9)` gibt uns eine Sequenz von Ganzzahlen, die mit 6 anfängt, in Schritten von 1 ansteigt, und *vor* 9 aufhört, also im Grunde 6..8.
- `range(20, 27, 2)` ist eine Sequenz von Ganzzahlen, die mit 20 anfängt, in Schritten von 2 ansteigt, und *vor* 27 aufhört, also $(20, 22, 24, 26)$.
- Ranges, wie Slices, können auch mit negativen Schritten arbeiten: `range(40, 30, -3)` beginnt mit 40, steigt in Schritten von 3 *ab*, und hört *vor* 30 auf, ist also $(40, 37, 34, 31)$.

Beispiel (1)

- Schauen wir uns mal ein Beispiel an.

Beispiel (1)

- Schauen wir uns mal ein Beispiel an.
- Zuerst wollen wir ein Dictionary bauen, in dem einige Ganzzahlen ihrem jeweiligen Quadrat zugeordnet sind.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Schauen wir uns mal ein Beispiel an.
- Zuerst wollen wir ein Dictionary bauen, in dem einige Ganzzahlen ihrem jeweiligen Quadrat zugeordnet sind.
- Wir benutzen vier `for`-Schleifen, um dieses Dictionary mit Daten zu füllen.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Schauen wir uns mal ein Beispiel an.
- Zuerst wollen wir ein Dictionary bauen, in dem einige Ganzzahlen ihrem jeweiligen Quadrat zugeordnet sind.
- Wir benutzen vier `for`-Schleifen, um dieses Dictionary mit Daten zu füllen.
- Wir benutzen jeweils `i` als Schleifenvariable.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484, 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Schauen wir uns mal ein Beispiel an.
- Zuerst wollen wir ein Dictionary bauen, in dem einige Ganzzahlen ihrem jeweiligen Quadrat zugeordnet sind.
- Wir benutzen vier `for`-Schleifen, um dieses Dictionary mit Daten zu füllen.
- Wir benutzen jeweils `i` als Schleifenvariable.
- In der ersten Schleife iteriert `i` über `range(5)`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Zuerst wollen wir ein Dictionary bauen, in dem einige Ganzzahlen ihrem jeweiligen Quadrat zugeordnet sind.
- Wir benutzen vier `for`-Schleifen, um dieses Dictionary mit Daten zu füllen.
- Wir benutzen jeweils `i` als Schleifenvariable.
- In der ersten Schleife iteriert `i` über `range(5)`.
- Bei der ersten Ausführung des Schleifenkörpers hat `i` den Wert `0`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Wir benutzen jeweils `i` als Schleifenvariable.
- In der ersten Schleife iteriert `i` über `range(5)`.
- Bei der ersten Ausführung des Schleifenkörpers hat `i` den Wert `0`.
- Der Schleifenkörper `squares[i] = i * i` ist also äquivalent zu `squares[0] = 0` und weist daher den Wert `0` dem Schlüssel `0` im Dictionary `squares` zu.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
    ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- In der ersten Schleife iteriert `i` über `range(5)`.
- Bei der ersten Ausführung des Schleifenkörpers hat `i` den Wert `0`.
- Der Schleifenkörper `squares[i] = i * i` ist also äquivalent zu `squares[0] = 0` und weist daher den Wert `0` dem Schlüssel `0` im Dictionary `squares` zu.
- In der zweiten Iteration hat `i` den Wert `1`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Bei der ersten Ausführung des Schleifenkörpers hat `i` den Wert `0`.
- Der Schleifenkörper `squares[i] = i * i` ist also äquivalent zu `squares[0] = 0` und weist daher den Wert `0` dem Schlüssel `0` im Dictionary `squares` zu.
- In der zweiten Iteration hat `i` den Wert `1`.
- Der Schleifenkörper `squares[i] = i * i` macht also `squares[1] = 1`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
    ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Der Schleifenkörper `squares[i] = i * i` ist also äquivalent zu `squares[0] = 0` und weist daher den Wert `0` dem Schlüssel `0` im Dictionary `squares` zu.
- In der zweiten Iteration hat `i` den Wert `1`.
- Der Schleifenkörper `squares[i] = i * i` macht also `squares[1] = 1`.
- In der dritten Iteration hat `i` den Wert `2` und wir führen `squares[2] = 4` aus.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- In der zweiten Iteration hat `i` den Wert `1`.
- Der Schleifenkörper `squares[i] = i * i` macht also `squares[1] = 1`.
- In der dritten Iteration hat `i` den Wert `2` und wir führen `squares[2] = 4` aus.
- Danach gilt `i = 3` und wir machen `squares[3] = 9`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
  ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Der Schleifenkörper `squares[i] = i * i` macht also `squares[1] = 1`.
- In der dritten Iteration hat `i` den Wert `2` und wir führen `squares[2] = 4` aus.
- Danach gilt `i = 3` und wir machen `squares[3] = 9`.
- Zuletzt erfolgt dann `squares[4] = 16` und die erste Schleife ist fertig.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- In der dritten Iteration hat `i` den Wert `2` und wir führen `squares[2] = 4` aus.
- Danach gilt `i = 3` und wir machen `squares[3] = 9`.
- Zuletzt erfolgt dann `squares[4] = 16` und die erste Schleife ist fertig.
- Mit der zweiten Schleife iterieren wir über `range(6, 9)`, also es gilt `i = 6`, dann `i = 7`, und schließlich `i = 8`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
  ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Danach gilt `i = 3` und wir machen `squares[3] = 9`.
- Zuletzt erfolgt dann `squares[4] = 16` und die erste Schleife ist fertig.
- Mit der zweiten Schleife iterieren wir über `range(6, 9)`, also es gilt `i = 6`, dann `i = 7`, und schließlich `i = 8`.
- Es erfolgt also `squares[6] = 36`, `squares[7] = 49`, und `squares[8] = 64`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Mit der zweiten Schleife iterieren wir über `range(6, 9)`, also es gilt `i = 6`, dann `i = 7`, und schließlich `i = 8`.
- Es erfolgt also `squares[6] = 36`, `squares[7] = 49`, und `squares[8] = 64`.
- Die dritte Schleife iteriert über `range(20, 27, 2)` und ihr Körper führt daher nacheinander `squares[20] = 400`, `squares[22] = 484`, `squares[24] = 576`, und `squares[26] = 676` aus.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
  ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Es erfolgt also `squares[6] = 36`, `squares[7] = 49`, und `squares[8] = 64`.
- Die dritte Schleife iteriert über `range(20, 27, 2)` und ihr Körper führt daher nacheinander `squares[20] = 400`, `squares[22] = 484`, `squares[24] = 576`, und `squares[26] = 676` aus.
- In der vierten Schleife, nimmt `i` die Werte der Sequenz `range(40, 30, -3)` an, die eine negative Schrittweite hat.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Es erfolgt also `squares[6] = 36`, `squares[7] = 49`, und `squares[8] = 64`.
- Die dritte Schleife iteriert über `range(20, 27, 2)` und ihr Körper führt daher nacheinander `squares[20] = 400`, `squares[22] = 484`, `squares[24] = 576`, und `squares[26] = 676` aus.
- In der vierten Schleife, nimmt `i` die Werte der Sequenz `range(40, 30, -3)` an, die eine negative Schrittweite hat.
- `i` wird daher zuerst `40`, dann `37`, dann `34`, und schließlich `31`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Die dritte Schleife iteriert über `range(20, 27, 2)` und ihr Körper führt daher nacheinander `squares[20] = 400`, `squares[22] = 484`, `squares[24] = 576`, und `squares[26] = 676` aus.
- In der vierten Schleife, nimmt `i` die Werte der Sequenz `range(40, 30, -3)` an, die eine negative Schrittweite hat.
- `i` wird daher zuerst `40`, dann `37`, dann `34`, und schließlich `31`.
- Nun drucken wir das Dictionary aus und bekommen das erwartete Ergebnis.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

Gute Praxis

Wenn der Wert einer Variable oder eines Parameters egal ist, dann sollten wie diese `_` nennen[40].

Beispiel (1)

Gute Praxis

Wenn der Wert einer Variable oder eines Parameters egal ist, dann sollten wie diese `_` nennen[40]. Diese Information ist nützlich für andere Programmierer sowie für statische Kodeanalyse.

Beispiel (1)

- Am Ende des Programmes probieren wir das mal aus.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
  ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Am Ende des Programmes probieren wir das mal aus.
- Wir wollen `"Hello World!"` dreimal ausgeben.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Am Ende des Programmes probieren wir das mal aus.
- Wir wollen `"Hello World!"` dreimal ausgeben.
- Wir könnten die Kodezeile einfach dreimal kopieren.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
  ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Am Ende des Programmes probieren wir das mal aus.
- Wir wollen `"Hello World!"` dreimal ausgeben.
- Wir könnten die Kodezeile einfach dreimal kopieren.
- Stattdessen packen wir sie in eine Schleife, die über `range(3)` iteriert.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Am Ende des Programmes probieren wir das mal aus.
- Wir wollen `"Hello World!"` dreimal ausgeben.
- Wir könnten die Kodezeile einfach dreimal kopieren.
- Stattdessen packen wir sie in eine Schleife, die über `range(3)` iteriert.
- Dabei ist uns der Wert der Schleifenvariable egal.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Am Ende des Programmes probieren wir das mal aus.
- Wir wollen `"Hello World!"` dreimal ausgeben.
- Wir könnten die Kodezeile einfach dreimal kopieren.
- Stattdessen packen wir sie in eine Schleife, die über `range(3)` iteriert.
- Dabei ist uns der Wert der Schleifenvariable egal.
- Also nennen wir sie einfach `_`.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
    ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Wir könnten die Kodezeile einfach dreimal kopieren.
- Stattdessen packen wir sie in eine Schleife, die über `range(3)` iteriert.
- Dabei ist uns der Wert der Schleifenvariable egal.
- Also nennen wir sie einfach `_`.
- Weil wir sie so nennen, ist jedem, der den Kode liest, sofort klar, dass uns der Wert der Schleifenvariablen völlig egal ist.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Dabei ist uns der Wert der Schleifenvariable egal.
- Also nennen wir sie einfach `_`.
- Weil wir sie so nennen, ist jedem, der den Kode liest, sofort klar, dass uns der Wert der Schleifenvariablen völlig egal ist.
- Wir machen deutlich, dass wir etwas dreimal tun wollen, dass der Index jeder Iteration der Schleife aber keine Rolle spielt.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484, 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Also nennen wir sie einfach `_`.
- Weil wir sie so nennen, ist jedem, der den Kode liest, sofort klar, dass uns der Wert der Schleifenvariablen völlig egal ist.
- Wir machen deutlich, dass wir etwas dreimal tun wollen, dass der Index jeder Iteration der Schleife aber keine Rolle spielt.
- Hätten wir die Variable stattdessen z. B. `d` genannt, dann würden Leute, die den Kode lesen, oder auch statische Analysetools sich wundern, warum wir `d` nirgendwo verwenden.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
  ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Wir machen deutlich, dass wir etwas dreimal tun wollen, dass der Index jeder Iteration der Schleife aber keine Rolle spielt.
- Hätten wir die Variable stattdessen z. B. `d` genannt, dann würden Leute, die den Kode lesen, oder auch statische Analysetools sich wundern, warum wir `d` nirgendwo verwenden.
- Es ist also immer sinnvoll, unsere Intentionen klar in unserem Kode auszudrücken.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ python3 for_loop_range.py ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
   ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (1)

- Wir machen deutlich, dass wir etwas dreimal tun wollen, dass der Index jeder Iteration der Schleife aber keine Rolle spielt.
- Hätten wir die Variable stattdessen z. B. `d` genannt, dann würden Leute, die den Kode lesen, oder auch statische Analysetools sich wundern, warum wir `d` nirgendwo verwenden.
- Es ist also immer sinnvoll, unsere Intentionen klar in unserem Kode auszudrücken.
- Denn „richtiger“ Kode ist viel komplizierter und jeder Hinweis, den wir geben, ist nützlich.

```python
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):   # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i      # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484,
 ↪ 24: 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

Beispiel (2)

- Benutzen wir nun diese neuen Informationen, um unser Programm zum Annähern von π zu verbessern.

Beispiel (2)

- Benutzen wir nun diese neuen Informationen, um unser Programm zum Annähern von π zu verbessern.
- Was wir in diesem Programm wieder und wieder tun, ist…

```python
from math import pi, sqrt   # We need sqrt. pi is for comparison.

# We use f-strings with Unicode escapes to print the current result.
# "\u03c0" is the Unicode escape for the Greek letter pi.
# "\u2248" is the Unicode escape for the "approximately equal" sign.
print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e = 6     # the number of edges: We start with a hexagon, i.e., e=6.
s = 1.0   # the side length: Initially 1, meaning the radius is also 1.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2   # We double the number of edges... ...now there are 12.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 24.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 48.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 96.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 192.
s = sqrt(2 - sqrt(4 - (s ** 2)))
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
```

↓ `python3 pi_liu_hui.py` ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

Beispiel (2)

- Benutzen wir nun diese neuen Informationen, um unser Programm zum Annähern von π zu verbessern.
- Was wir in diesem Programm wieder und wieder tun, ist:
  - Wir verdoppeln die Anzahl `e` der Seiten eines regelmäsigen `e`-gons mit `e *= 2`.

```python
from math import pi, sqrt   # We need sqrt. pi is for comparison.

# We use f-strings with Unicode escapes to print the current result.
# "\u03c0" is the Unicode escape for the Greek letter pi.
# "\u2248" is the Unicode escape for the "approximately equal" sign.
print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e = 6     # the number of edges: We start with a hexagon, i.e., e=6.
s = 1.0   # the side length: Initially 1, meaning the radius is also 1.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2    # We double the number of edges...  ...now there are 12.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 24.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 48.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 96.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 192.
s = sqrt(2 - sqrt(4 - (s ** 2)))
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
```

↓ `python3 pi_liu_hui.py` ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

Beispiel (2)

- Benutzen wir nun diese neuen Informationen, um unser Programm zum Annähern von π zu verbessern.
- Was wir in diesem Programm wieder und wieder tun, ist:
  - Wir verdoppeln die Anzahl `e` der Seiten eines regelmäsigen `e`-gons mit `e *= 2`.
  - Wir berechnen die Länge `s` der Seiten.

```python
from math import pi, sqrt   # We need sqrt. pi is for comparison.

# We use f-strings with Unicode escapes to print the current result.
# "\u03c0" is the Unicode escape for the Greek letter pi.
# "\u2248" is the Unicode escape for the "approximately equal" sign.
print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e = 6    # the number of edges: We start with a hexagon, i.e., e=6.
s = 1.0  # the side length: Initially 1, meaning the radius is also 1.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2   # We double the number of edges... ...now there are 12.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 24.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 48.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 96.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 192.
s = sqrt(2 - sqrt(4 - (s ** 2)))
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
```

↓ `python3 pi_liu_hui.py` ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

Beispiel (2)

- Benutzen wir nun diese neuen Informationen, um unser Programm zum Annähern von π zu verbessern.
- Was wir in diesem Programm wieder und wieder tun, ist:
  - Wir verdoppeln die Anzahl `e` der Seiten eines regelmäsigen `e`-gons mit `e *= 2`.
  - Wir berechnen die Länge `s` der Seiten.
  - Wir geben die neue Annäherung `e * s /2` von π aus.

```python
from math import pi, sqrt   # We need sqrt. pi is for comparison.

# We use f-strings with Unicode escapes to print the current result.
# "\u03c0" is the Unicode escape for the Greek letter pi.
# "\u2248" is the Unicode escape for the "approximately equal" sign.
print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e = 6     # the number of edges: We start with a hexagon, i.e., e=6.
s = 1.0   # the side length: Initially 1, meaning the radius is also 1.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2   # We double the number of edges...  ...now there are 12.
s = sqrt(2 - sqrt(4 - (s ** 2)))   # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 24.
s = sqrt(2 - sqrt(4 - (s ** 2)))   # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 48.
s = sqrt(2 - sqrt(4 - (s ** 2)))   # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 96.
s = sqrt(2 - sqrt(4 - (s ** 2)))   # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges...  ...now there are 192.
s = sqrt(2 - sqrt(4 - (s ** 2)))
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
```

↓ `python3 pi_liu_hui.py` ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

Beispiel (2)

- Benutzen wir nun diese neuen Informationen, um unser Programm zum Annähern von π zu verbessern.
- Was wir in diesem Programm wieder und wieder tun, ist:
  - Wir verdoppeln die Anzahl `e` der Seiten eines regelmäsigen `e`-gons mit `e *= 2`.
  - Wir berechnen die Länge `s` der Seiten.
  - Wir geben die neue Annäherung `e * s /2` von π aus.
- Wenn wir all das einfach in eine Schleife packen...

```python
"""We execute Liu Hui's method to approximate pi in a loop."""
from math import pi, sqrt

print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e: int = 6   # the number of edges: We start with a hexagon, i.e., e=6.
s: float = 1.0  # the side length: Initially 1, i.e., radius is also 1.

for _ in range(6):
    print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
    e *= 2  # We double the number of edges...
    s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length
```

↓ `python3 for_loop_pi_liu_hui.py` ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

Beispiel (2)

- Was wir in diesem Programm wieder und wieder tun, ist:
    - Wir verdoppeln die Anzahl `e` der Seiten eines regelmäsigen `e`-gons mit `e *= 2`.
    - Wir berechnen die Länge `s` der Seiten.
    - Wir geben die neue Annäherung `e * s /2` von π aus.
- Wenn wir all das einfach in eine Schleife packen...
- dann verkürzt sich das Programm von über 25 auf 10 Zeilen Kode!

```python
"""We execute Liu Hui's method to approximate pi in a loop."""
from math import pi, sqrt

print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e: int = 6  # the number of edges: We start with a hexagon, i.e., e=6.
s: float = 1.0  # the side length: Initially 1, i.e., radius is also 1.

for _ in range(6):
    print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
    e *= 2  # We double the number of edges...
    s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length
```

↓ python3 for_loop_pi_liu_hui.py ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

Beispiel (2)

- Was wir in diesem Programm wieder und wieder tun, ist:
  - Wir verdoppeln die Anzahl `e` der Seiten eines regelmäsigen `e`-gons mit `e *= 2`.
  - Wir berechnen die Länge `s` der Seiten.
  - Wir geben die neue Annäherung `e * s /2` von π aus.
- Wenn wir all das einfach in eine Schleife packen. . .
- dann verkürzt sich das Programm von über 25 auf 10 Zeilen Kode!
- Und die Ausgabe ist die gleiche!

```python
"""We execute Liu Hui's method to approximate pi in a loop."""
from math import pi, sqrt

print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e: int = 6   # the number of edges: We start with a hexagon, i.e., e=6.
s: float = 1.0  # the side length: Initially 1, i.e., radius is also 1.

for _ in range(6):
    print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
    e *= 2  # We double the number of edges...
    s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length
```

↓ `python3 for_loop_pi_liu_hui.py` ↓

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

continue und break

`continue` und `break`

- Schleifen haben oft komplizierte Körper, die Alternativen oder andere Schleifen beinhalten können.

`continue` und `break`

- Schleifen haben oft komplizierte Körper, die Alternativen oder andere Schleifen beinhalten können.
- Es ist nicht ungewöhnlich dass wir manchmal nach ein paar Berechnungen im Körper einer Schleife schon wissen, dass wir mit der nächsten Iteration weitermachen können, anstatt den Rest des Schleifenkörpers auszuführen.

`continue` und `break`

- Schleifen haben oft komplizierte Körper, die Alternativen oder andere Schleifen beinhalten können.
- Es ist nicht ungewöhnlich dass wir manchmal nach ein paar Berechnungen im Körper einer Schleife schon wissen, dass wir mit der nächsten Iteration weitermachen können, anstatt den Rest des Schleifenkörpers auszuführen.
- Manchmal wissen wir auch nach ein paar Schleifen-Iterationen bereits, dass wir eigentlich aufhören können anstatt die weitere Iterationen durchzuführen.

`continue` und `break`

- Schleifen haben oft komplizierte Körper, die Alternativen oder andere Schleifen beinhalten können.
- Es ist nicht ungewöhnlich dass wir manchmal nach ein paar Berechnungen im Körper einer Schleife schon wissen, dass wir mit der nächsten Iteration weitermachen können, anstatt den Rest des Schleifenkörpers auszuführen.
- Manchmal wissen wir auch nach ein paar Schleifen-Iterationen bereits, dass wir eigentlich aufhören können anstatt die weitere Iterationen durchzuführen.
- Für den ersten Fall gibt es das Schlüsselwort `continue`, für den zweiten Fall das Schlüsselwort `break`[49].

Beispiel

- Schauen wir uns mal ein Beispiel an.

Beispiel

- Schauen wir uns mal ein Beispiel an.
- Wir iterieren mit einer Variable `i` über die 15 Werte von `0` bis `14`, also über `range(15)`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Schauen wir uns mal ein Beispiel an.
- Wir iterieren mit einer Variable `i` über die 15 Werte von `0` bis `14`, also über `range(15)`.
- Im Schleifenkörper erstellen wir erst einen String `s` mit dem aktuellen Wert von `i` mit Hilfe des f-Strings `f"i is now {i}."`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Schauen wir uns mal ein Beispiel an.
- Wir iterieren mit einer Variable `i` über die 15 Werte von `0` bis `14`, also über `range(15)`.
- Im Schleifenkörper erstellen wir erst einen String `s` mit dem aktuellen Wert von `i` mit Hilfe des f-Strings `f"i is now {i}."`.
- Der letzte Befehl in der Schleife, `print(s)`, gibt diesen String aus.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Schauen wir uns mal ein Beispiel an.
- Wir iterieren mit einer Variable `i` über die 15 Werte von `0` bis `14`, also über `range(15)`.
- Im Schleifenkörper erstellen wir erst einen String `s` mit dem aktuellen Wert von `i` mit Hilfe des f-Strings `f"i is now {i}."`.
- Der letzte Befehl in der Schleife, `print(s)`, gibt diesen String aus.
- Anstatt das wir `i` von `0` bis `14` laufen lassen, wir überlegen uns, dass wir abbrechen wollen, sobald `i` größer als `10` wird.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Im Schleifenkörper erstellen wir erst einen String `s` mit dem aktuellen Wert von `i` mit Hilfe des f-Strings `f"i is now {i}."`.
- Der letzte Befehl in der Schleife, `print(s)`, gibt diesen String aus.
- Anstatt das wir `i` von `0` bis `14` laufen lassen, wir überlegen uns, dass wir abbrechen wollen, sobald `i` größer als `10` wird.
- Normalerweise würden wir dafür die `range` ändern, wir wollen hier aber einfach mal der `break`-Statement verwenden.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Der letzte Befehl in der Schleife, `print(s)`, gibt diesen String aus.
- Anstatt das wir `i` von `0` bis `14` laufen lassen, wir überlegen uns, dass wir abbrechen wollen, sobald `i` größer als `10` wird.
- Normalerweise würden wir dafür die `range` ändern, wir wollen hier aber einfach mal der `break`-Statement verwenden.
- Wir schreiben es also in eine Alternative hinein, mit Bedingung `if i > 10:`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Der letzte Befehl in der Schleife, `print(s)`, gibt diesen String aus.
- Anstatt das wir `i` von `0` bis `14` laufen lassen, wir überlegen uns, dass wir abbrechen wollen, sobald `i` größer als `10` wird.
- Normalerweise würden wir dafür die `range` ändern, wir wollen hier aber einfach mal der `break`-Statement verwenden.
- Wir schreiben es also in eine Alternative hinein, mit Bedingung `if i > 10:`.
- Also wenn `i > 10`, dann wird `break` ausgeführt, was die Schleife dann sofort beendet.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Normalerweise würden wir dafür die `range` ändern, wir wollen hier aber einfach mal der `break`-Statement verwenden.
- Wir schreiben es also in eine Alternative hinein, mit Bedingung `if i > 10:`.
- Also wenn `i > 10`, dann wird `break` ausgeführt, was die Schleife dann sofort beendet.
- Die aktuelle Iteration wird dann nicht mehr beendet und der Kode im restlichen Schleifenkörper wird überprungen und die Schleife verlassen.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Wir schreiben es also in eine Alternative hinein, mit Bedingung `if i > 10:`.
- Also wenn `i > 10`, dann wird `break` ausgeführt, was die Schleife dann sofort beendet.
- Die aktuelle Iteration wird dann nicht mehr beendet und der Kode im restlichen Schleifenkörper wird überprungen und die Schleife verlassen.
- Es geht dann nach der Schleife weiter, also mit `print("All done.")`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Also wenn `i > 10`, dann wird `break` ausgeführt, was die Schleife dann sofort beendet.
- Die aktuelle Iteration wird dann nicht mehr beendet und der Kode im restlichen Schleifenkörper wird überprungen und die Schleife verlassen.
- Es geht dann nach der Schleife weiter, also mit `print("All done.")`.
- Wenn `i > 10` aber nicht zutrifft, dann wird das `break` nicht ausgeführt aber stattdessen der restliche Kode im Schleifenkörper.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Die aktuelle Iteration wird dann nicht mehr beendet und der Kode im restlichen Schleifenkörper wird überprungen und die Schleife verlassen.
- Es geht dann nach der Schleife weiter, also mit `print("All done.")`.
- Wenn `i > 10` aber nicht zutrifft, dann wird das `break` nicht ausgeführt aber stattdessen der restliche Kode im Schleifenkörper.
- Für den Fall, dass `i` $\in 5..8$, dann wollen wir direkt mit der nächsten Schleifen-Iteration weitermachen.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Es geht dann nach der Schleife weiter, also mit `print("All done.")`.
- Wenn `i > 10` aber nicht zutrifft, dann wird das `break` nicht ausgeführt aber stattdessen der restliche Kode im Schleifenkörper.
- Für den Fall, dass `i` $\in 5..8$, dann wollen wir direkt mit der nächsten Schleifen-Iteration weitermachen.
- Wir wollen keinen Text ausgeben und daher das `print` im Schleifenkörper auch nicht aufrufen.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Es geht dann nach der Schleife weiter, also mit `print("All done.")`.
- Wenn `i > 10` aber nicht zutrifft, dann wird das `break` nicht ausgeführt aber stattdessen der restliche Kode im Schleifenkörper.
- Für den Fall, dass `i` $\in$ 5..8, dann wollen wir direkt mit der nächsten Schleifen-Iteration weitermachen.
- Wir wollen keinen Text ausgeben und daher das `print` im Schleifenkörper auch nicht aufrufen.
- Das können wir mit dem `continue`-Statement erreichen.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Wenn `i > 10` aber nicht zutrifft, dann wird das `break` nicht ausgeführt aber stattdessen der restliche Kode im Schleifenkörper.
- Für den Fall, dass `i` $\in$ 5..8, dann wollen wir direkt mit der nächsten Schleifen-Iteration weitermachen.
- Wir wollen keinen Text ausgeben und daher das `print` im Schleifenkörper auch nicht aufrufen.
- Das können wir mit dem `continue`-Statement erreichen.
- Wenn `5 <= i <= 8` zutrifft, dann machen wir `continue`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Für den Fall, dass `i` $\in$ 5..8, dann wollen wir direkt mit der nächsten Schleifen-Iteration weitermachen.
- Wir wollen keinen Text ausgeben und daher das `print` im Schleifenkörper auch nicht aufrufen.
- Das können wir mit dem `continue`-Statement erreichen.
- Wenn `5 <= i <= 8` zutrifft, dann machen wir `continue`.
- Das bedeutet, dass für `i == 5` der Kontrollfluss vom `continue`-Statement direkt zum Kopf der Schleife zurückkehrt.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Wir wollen keinen Text ausgeben und daher das `print` im Schleifenkörper auch nicht aufrufen.
- Das können wir mit dem `continue`-Statement erreichen.
- Wenn `5 <= i <= 8` zutrifft, dann machen wir `continue`.
- Das bedeutet, dass für `i == 5` der Kontrollfluss vom `continue`-Statement direkt zum Kopf der Schleife zurückkehrt.
- Die Schleife setzt `i = 6` und das selbe passiert wieder.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Das können wir mit dem `continue`-Statement erreichen.
- Wenn `5 <= i <= 8` zutrifft, dann machen wir `continue`.
- Das bedeutet, dass für `i == 5` der Kontrollfluss vom `continue`-Statement direkt zum Kopf der Schleife zurückkehrt.
- Die Schleife setzt `i = 6` und das selbe passiert wieder.
- Das geht so weiter, bis `i == 9`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Wenn `5 <= i <= 8` zutrifft, dann machen wir `continue`.
- Das bedeutet, dass für `i == 5` der Kontrollfluss vom `continue`-Statement direkt zum Kopf der Schleife zurückkehrt.
- Die Schleife setzt `i = 6` und das selbe passiert wieder.
- Das geht so weiter, bis `i == 9`.
- Die Bedingung `5 <= i <= 8` trifft nämlich nicht zu für alle `i` $\in 0..4 \cup 9..15$.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Das bedeutet, dass für `i == 5` der Kontrollfluss vom `continue`-Statement direkt zum Kopf der Schleife zurückkehrt.
- Die Schleife setzt `i = 6` und das selbe passiert wieder.
- Das geht so weiter, bis `i == 9`.
- Die Bedingung `5 <= i <= 8` trifft nämlich nicht zu für alle `i` $\in 0..4 \cup 9..15$.
- Das `print(s)` kann also nur in diesen Fällen erreicht werden.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Die Schleife setzt `i = 6` und das selbe passiert wieder.
- Das geht so weiter, bis `i == 9`.
- Die Bedingung `5 <= i <= 8` trifft nämlich nicht zu für alle `i` $\in 0..4 \cup 9..15$.
- Das `print(s)` kann also nur in diesen Fällen erreicht werden.
- Natürlich wird die Schleife abbrechen, sobald `i == 11`.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ python3 for_loop_continue_break.py ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Das geht so weiter, bis `i == 9`.
- Die Bedingung `5 <= i <= 8` trifft nämlich nicht zu für alle `i` $\in 0..4 \cup 9..15$.
- Das `print(s)` kann also nur in diesen Fällen erreicht werden.
- Natürlich wird die Schleife abbrechen, sobald `i == 11`.
- Das Programm wird also `s` nur für `i` $\in 0..4 \cup \{9, 10\}$ ausgeben, bevor letztendlich `All done.` ausgegeben wird.

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Beispiel

- Das `print(s)` kann also nur in diesen Fällen erreicht werden.
- Natürlich wird die Schleife abbrechen, sobald `i == 11`.
- Das Programm wird also `s` nur für `i` $\in 0..4 \cup \{9, 10\}$ ausgeben, bevor letztendlich `All done.` ausgegeben wird.
- Mit `break` und `continue` haben wir nun zwei neue Werkzeuge, mit denen wir entweder die die ganze Schleie abbrechen können oder die aktuelle Iteration beenden (und mit der nächsten weitermachen können, wenn es eine gibt).

```python
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

Schleifen verschachteln

Schleifen verschachteln

- Genau wir Alternativen, können wir Schleifen beliebig verschachteln (EN: *nest*).

Schleifen verschachteln

- Genau wir Alternativen, können wir Schleifen beliebig verschachteln (EN: *nest*).
- Die Körper von Schleifen können andere Schleifen und Alternativen beinhalten.

Schleifen verschachteln

- Genau wir Alternativen, können wir Schleifen beliebig verschachteln (EN: *nest*).
- Die Körper von Schleifen können andere Schleifen und Alternativen beinhalten.
- Schleifen können auch in den Verzweigungen von Alternativen beinhaltet sein.

Schleifen verschachteln

- Genau wir Alternativen, können wir Schleifen beliebig verschachteln (EN: *nest*).
- Die Körper von Schleifen können andere Schleifen und Alternativen beinhalten.
- Schleifen können auch in den Verzweigungen von Alternativen beinhaltet sein.
- Probieren wir das mal aus, in dem wir eine Liste aller Primzahlen kleiner als 200 erstellen!

Schleifen verschachteln

- Genau wir Alternativen, können wir Schleifen beliebig verschachteln (EN: *nest*).
- Die Körper von Schleifen können andere Schleifen und Alternativen beinhalten.
- Schleifen können auch in den Verzweigungen von Alternativen beinhaltet sein.
- Probieren wir das mal aus, in dem wir eine Liste aller Primzahlen kleiner als 200 erstellen!

Definition: Primezahl

Eine Primzahl (EN: *prime number*) $p \in \mathbb{N}_1$ ist eine positive Ganzzahl $p > 1$, also größer als eins, die keine positiven ganzzahligen Teiler anders als 1 und p selbst hat[17,61,77].

Beispiel

Definition: Primzahl

Eine Primzahl $p \in \mathbb{N}_1$ ist eine positive Ganzzahl $p > 1$, also größer als eins, die keine positiven ganzzahligen Teiler anders als 1 und p selbst hat[17,61,77].

Beispiel

- In unserem Programm wollen wir alle Primzahlen $p < 200$ in einer Liste `primes` speichern.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- In unserem Programm wollen wir alle Primzahlen $p < 200$ in einer Liste `primes` speichern.
- Wir wissen, dass `2` eine Primzahl ist – und zwar die einzige gerade Primzahl – also können wir sie direkt bei der Initialisierung der Liste hineinschreiben.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- In unserem Programm wollen wir alle Primzahlen $p < 200$ in einer Liste `primes` speichern.
- Wir wissen, dass `2` eine Primzahl ist – und zwar die einzige gerade Primzahl – also können wir sie direkt bei der Initialisierung der Liste hineinschreiben.
- Wir setzen also anfangs `primes = [2]`.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- In unserem Programm wollen wir alle Primzahlen $p < 200$ in einer Liste `primes` speichern.
- Wir wissen, dass `2` eine Primzahl ist – und zwar die einzige gerade Primzahl – also können wir sie direkt bei der Initialisierung der Liste hineinschreiben.
- Wir setzen also anfangs `primes = [2]`.
- Dann brauchen wir uns nachher nur noch um ungerade Zahlen aus dem Intervall 3..199 zu kümmern.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wir wissen, dass `2` eine Primzahl ist – und zwar die einzige gerade Primzahl – also können wir sie direkt bei der Initialisierung der Liste hineinschreiben.
- Wir setzen also anfangs `primes = [2]`.
- Dann brauchen wir uns nachher nur noch um ungerade Zahlen aus dem Intervall 3..199 zu kümmern.
- Aus Interesse wollen wir auch die Anhahl der Divisionen in der Variable `n_divisions` mitzählen, die wir machen, bis wir die ganze Liste von Primzahlen zusammenhaben.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wir setzen also anfangs `primes = [2]`.
- Dann brauchen wir uns nachher nur noch um ungerade Zahlen aus dem Intervall 3..199 zu kümmern.
- Aus Interesse wollen wir auch die Anhahl der Divisionen in der Variable `n_divisions` mitzählen, die wir machen, bis wir die ganze Liste von Primzahlen zusammenhaben.
- Um alle Primzahlen aus 2..199 zu finden, lassen wir die Schleifenvariable `candidate` über `range(3, 200, 2)` iterieren.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Dann brauchen wir uns nachher nur noch um ungerade Zahlen aus dem Intervall 3..199 zu kümmern.
- Aus Interesse wollen wir auch die Anhahl der Divisionen in der Variable `n_divisions` mitzählen, die wir machen, bis wir die ganze Liste von Primzahlen zusammenhaben.
- Um alle Primzahlen aus 2..199 zu finden, lassen wir die Schleifenvariable `candidate` über `range(3, 200, 2)` iterieren.
- Das ist die Sequenz von Ganzzahlen, die mit `3` anfängt, mit Schrittweite 2 steigt, und genau vor `200` aufhört.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Aus Interesse wollen wir auch die Anhahl der Divisionen in der Variable `n_divisions` mitzählen, die wir machen, bis wir die ganze Liste von Primzahlen zusammenhaben.
- Um alle Primzahlen aus 2..199 zu finden, lassen wir die Schleifenvariable `candidate` über `range(3, 200, 2)` iterieren.
- Das ist die Sequenz von Ganzzahlen, die mit `3` anfängt, mit Schrittweite 2 steigt, und genau vor `200` aufhört.
- Gerade Zahlen außer 2 sind keine Primzahlen, also sollte das OK sein.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Um alle Primzahlen aus 2..199 zu finden, lassen wir die Schleifenvariable `candidate` über `range(3, 200, 2)` iterieren.
- Das ist die Sequenz von Ganzzahlen, die mit `3` anfängt, mit Schrittweite 2 steigt, und genau vor `200` aufhört.
- Gerade Zahlen außer 2 sind keine Primzahlen, also sollte das OK sein.
- `candidate` wird also iterative `3`, `5`, `7`, . . . , und irgendwann `195`, `197`, und `199` werden.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Das ist die Sequenz von Ganzzahlen, die mit `3` anfängt, mit Schrittweite 2 steigt, und genau vor `200` aufhört.
- Gerade Zahlen außer 2 sind keine Primzahlen, also sollte das OK sein.
- `candidate` wird also iterative `3`, `5`, `7`, ..., und irgendwann `195`, `197`, und `199` werden.
- Für jeden Wert von `candidate` fangen wir mit der Annahme an, dass sie eine Primzahl ist und versuchen das Gegenteil zu beweisen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Gerade Zahlen außer 2 sind keine Primzahlen, also sollte das OK sein.
- `candidate` wird also iterative `3`, `5`, `7`, ..., und irgendwann `195`, `197`, und `199` werden.
- Für jeden Wert von `candidate` fangen wir mit der Annahme an, dass sie eine Primzahl ist und versuchen das Gegenteil zu beweisen.
- Wir setzen `is_prime = True` und versuchen, einen ganzzahligen Teiler für `candidate` zu finden.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- `candidate` wird also iterative `3`, `5`, `7`, . . . , und irgendwann `195`, `197`, und `199` werden.
- Für jeden Wert von `candidate` fangen wir mit der Annahme an, dass sie eine Primzahl ist und versuchen das Gegenteil zu beweisen.
- Wir setzen `is_prime = True` und versuchen, einen ganzzahligen Teiler für `candidate` zu finden.
- Wenn uns das gelingt, dann setzen wir `is_prime = False`.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Für jeden Wert von `candidate` fangen wir mit der Annahme an, dass sie eine Primzahl ist und versuchen das Gegenteil zu beweisen.
- Wir setzen `is_prime = True` und versuchen, einen ganzzahligen Teiler für `candidate` zu finden.
- Wenn uns das gelingt, dann setzen wir `is_prime = False`.
- Wenn wir keinen Teiler finden, dann bleibt `is_prime` `True`.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wir setzen `is_prime = True` und versuchen, einen ganzzahligen Teiler für `candidate` zu finden.
- Wenn uns das gelingt, dann setzen wir `is_prime = False`.
- Wenn wir keinen Teiler finden, dann bleibt `is_prime` `True`.
- Dann würden wir `candidate` zur Liste `primes` hinzufügen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wenn uns das gelingt, dann setzen wir `is_prime = False`.
- Wenn wir keinen Teiler finden, dann bleibt `is_prime` `True`.
- Dann würden wir `candidate` zur Liste `primes` hinzufügen.
- Das ist der Plan.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wenn wir keinen Teiler finden, dann bleibt `is_prime` `True`.
- Dann würden wir `candidate` zur Liste `primes` hinzufügen.
- Das ist der Plan.
- Wir werden diesen Plan mit einer geschachtelten / inneren Schleife implementieren.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Dann würden wir `candidate` zur Liste `primes` hinzufügen.
- Das ist der Plan.
- Wir werden diesen Plan mit einer geschachtelten / inneren Schleife implementieren.
- Weil die Schleifenvariable `candidate` immer ungerade ist, kommen auch nur ungerade Zahlen als Teiler in Frage.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199].
```

Beispiel

- Das ist der Plan.
- Wir werden diesen Plan mit einer geschachtelten / inneren Schleife implementieren.
- Weil die Schleifenvariable `candidate` immer ungerade ist, kommen auch nur ungerade Zahlen als Teiler in Frage.
- Natürlich müssen diese auch größer oder gleich `3` sein.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wir werden diesen Plan mit einer geschachtelten / inneren Schleife implementieren.
- Weil die Schleifenvariable `candidate` immer ungerade ist, kommen auch nur ungerade Zahlen als Teiler in Frage.
- Natürlich müssen diese auch größer oder gleich `3` sein.
- Wir müssen auch nur dann prüfen, ob eine Zahl `check` ein Teiler von `candidate`, wenn sie nicht größer als $\sqrt{\texttt{candidate}}$ ist.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Weil die Schleifenvariable `candidate` immer ungerade ist, kommen auch nur ungerade Zahlen als Teiler in Frage.
- Natürlich müssen diese auch größer oder gleich `3` sein.
- Wir müssen auch nur dann prüfen, ob eine Zahl `check` ein Teiler von `candidate`, wenn sie nicht größer als $\sqrt{\texttt{candidate}}$ ist.
- Wenn wir nämlich drei Ganzzahlen a, b, und c hätten, so das $a = b * c$, dann gilt $c = \frac{a}{b}$.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Natürlich müssen diese auch größer oder gleich `3` sein.
- Wir müssen auch nur dann prüfen, ob eine Zahl `check` ein Teiler von `candidate`, wenn sie nicht größer als $\sqrt{\texttt{candidate}}$ ist.
- Wenn wir nämlich drei Ganzzahlen a, b, und c hätten, so das $a = b * c$, dann gilt $c = \frac{a}{b}$.
- Wenn nun aber $b > \sqrt{a}$, dann muss $c < \frac{a}{\sqrt{a}}$, was heißt dass $c < \sqrt{a}$.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wir müssen auch nur dann prüfen, ob eine Zahl `check` ein Teiler von `candidate`, wenn sie nicht größer als $\sqrt{\texttt{candidate}}$ ist.
- Wenn wir nämlich drei Ganzzahlen a, b, und c hätten, so das $a = b * c$, dann gilt $c = \frac{a}{b}$.
- Wenn nun aber $b > \sqrt{a}$, dann muss $c < \frac{a}{\sqrt{a}}$, was heißt dass $c < \sqrt{a}$.
- Andersherum, weil $a = \sqrt{a} * \sqrt{a}$ per Definition gilt, wäre es unmöglich, dass $a = b * c$ gilt wenn sowohl $b > \sqrt{a}$ und $c \geq \sqrt{a}$.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wenn wir nämlich drei Ganzzahlen a, b, und c hätten, so das $a = b * c$, dann gilt $c = \frac{a}{b}$.
- Wenn nun aber $b > \sqrt{a}$, dann muss $c < \frac{a}{\sqrt{a}}$, was heißt dass $c < \sqrt{a}$.
- Andersherum, weil $a = \sqrt{a} * \sqrt{a}$ per Definition gilt, wäre es unmöglich, dass $a = b * c$ gilt wenn sowohl $b > \sqrt{a}$ und $c \geq \sqrt{a}$.
- Mit anderen Worten, wenn $b > \sqrt{a}$ ein Teiler von a ist, dann gibt es einen Teiler c mit $c < \sqrt{a} < b$.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wenn nun aber $b > \sqrt{a}$, dann muss $c < \frac{a}{\sqrt{a}}$, was heißt dass $c < \sqrt{a}$.
- Andersherum, weil $a = \sqrt{a} * \sqrt{a}$ per Definition gilt, wäre es unmöglich, dass $a = b * c$ gilt wenn sowohl $b > \sqrt{a}$ und $c \geq \sqrt{a}$.
- Mit anderen Worten, wenn $b > \sqrt{a}$ ein Teiler von a ist, dann gibt es einen Teiler c mit $c < \sqrt{a} < b$.
- Und den würden wir mit `check` finden, bevor es $\sqrt{\texttt{candidate}}$ erreicht.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Andersherum, weil $a = \sqrt{a} * \sqrt{a}$ per Definition gilt, wäre es unmöglich, dass $a = b * c$ gilt wenn sowohl $b > \sqrt{a}$ und $c \geq \sqrt{a}$.
- Mit anderen Worten, wenn $b > \sqrt{a}$ ein Teiler von a ist, dann gibt es einen Teiler c mit $c < \sqrt{a} < b$.
- Und den würden wir mit `check` finden, bevor es $\sqrt{\texttt{candidate}}$ erreicht.
- Die meisten Ganzzahlen haben keine ganzzahligen Quadratwurzeln.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Mit anderen Worten, wenn $b > \sqrt{a}$ ein Teiler von a ist, dann gibt es einen Teiler c mit $c < \sqrt{a} < b$.
- Und den würden wir mit `check` finden, bevor es $\sqrt{\texttt{candidate}}$ erreicht.
- Die meisten Ganzzahlen haben keine ganzzahligen Quadratwurzeln.
- Da ganzzahlige Teiler aber keine Nachkommastellen haben können, reicht es aus, wenn wir $\lfloor\sqrt{\texttt{candidate}}\rfloor$ als Obergrenze nehmen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Und den würden wir mit `check` finden, bevor es $\sqrt{\texttt{candidate}}$ erreicht.
- Die meisten Ganzzahlen haben keine ganzzahligen Quadratwurzeln.
- Da ganzzahlige Teiler aber keine Nachkommastellen haben können, reicht es aus, wenn wir $\lfloor\sqrt{\texttt{candidate}}\rfloor$ als Obergrenze nehmen.
- In Python kann so eine „abgerundete“ Quadratwurzel $\lfloor\sqrt{a}\rfloor$ einer Ganzzahl a mit der Funktion `isqrt` aus dem Modul `math` berechnet werden.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Die meisten Ganzzahlen haben keine ganzzahligen Quadratwurzeln.
- Da ganzzahlige Teiler aber keine Nachkommastellen haben können, reicht es aus, wenn wir $\lfloor\sqrt{\texttt{candidate}}\rfloor$ als Obergrenze nehmen.
- In Python kann so eine „abgerundete“ Quadratwurzel $\lfloor\sqrt{a}\rfloor$ einer Ganzzahl a mit der Funktion `isqrt` aus dem Modul `math` berechnet werden.
- Wir importieren diese Funktion daher am Anfang unseres Programms.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Da ganzzahlige Teiler aber keine Nachkommastellen haben können, reicht es aus, wenn wir $\lfloor\sqrt{\texttt{candidate}}\rfloor$ als Obergrenze nehmen.
- In Python kann so eine „abgerundete“ Quadratwurzel $\lfloor\sqrt{a}\rfloor$ einer Ganzzahl a mit der Funktion `isqrt` aus dem Modul `math` berechnet werden.
- Wir importieren diese Funktion daher am Anfang unseres Programms.
- Darum können wir unsere zweite, innere Schleife mit der Schleifenvariablen `check` über `range(3, isqrt(candidate)+ 1, 2)` iterieren lassen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- In Python kann so eine „abgerundete“ Quadratwurzel $\lfloor\sqrt{a}\rfloor$ einer Ganzzahl a mit der Funktion `isqrt` aus dem Modul `math` berechnet werden.
- Wir importieren diese Funktion daher am Anfang unseres Programms.
- Darum können wir unsere zweite, innere Schleife mit der Schleifenvariablen `check` über `range(3, isqrt(candidate)+ 1, 2)` iterieren lassen.
- Für alle `candidate <= 8` wird diese Schleife gar nicht ausgeführt, denn `isqrt(8)== 2`.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wir importieren diese Funktion daher am Anfang unseres Programms.
- Darum können wir unsere zweite, innere Schleife mit der Schleifenvariablen `check` über `range(3, isqrt(candidate)+ 1, 2)` iterieren lassen.
- Für alle `candidate <= 8` wird diese Schleife gar nicht ausgeführt, denn `isqrt(8)== 2`.
- Dann bleibt `is_prime` `True` und wir hängen `candidate` in `primes` an.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Darum können wir unsere zweite, innere Schleife mit der Schleifenvariablen `check` über `range(3, isqrt(candidate)+ 1, 2)` iterieren lassen.
- Für alle `candidate <= 8` wird diese Schleife gar nicht ausgeführt, denn `isqrt(8)== 2`.
- Dann bleibt `is_prime` `True` und wir hängen `candidate` in `primes` an.
- Für `candidate > 3`, geht `check` dann von `3` bis $\lfloor\sqrt{\texttt{candidate}}\rfloor$.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Für alle `candidate <= 8` wird diese Schleife gar nicht ausgeführt, denn `isqrt(8)== 2`.
- Dann bleibt `is_prime` `True` und wir hängen `candidate` in `primes` an.
- Für `candidate > 3`, geht `check` dann von `3` bis $\left\lfloor\sqrt{\texttt{candidate}}\right\rfloor$.
- Im Körper der inneren Schleife müssen wir nun gucken, ob `check` ein ganzzahliger Teiler von `candidate` ist.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Dann bleibt `is_prime` `True` und wir hängen `candidate` in `primes` an.
- Für `candidate > 3`, geht `check` dann von `3` bis $\lfloor\sqrt{\texttt{candidate}}\rfloor$.
- Im Körper der inneren Schleife müssen wir nun gucken, ob `check` ein ganzzahliger Teiler von `candidate` ist.
- Das geht, in dem wir den Rest der Division `canddiate`/`check` berechnen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Für `candidate > 3`, geht `check` dann von `3` bis $\lfloor\sqrt{\texttt{candidate}}\rfloor$.
- Im Körper der inneren Schleife müssen wir nun gucken, ob `check` ein ganzzahliger Teiler von `candidate` ist.
- Das geht, in dem wir den Rest der Division `canddiate`/`check` berechnen.
- Das geht mit der Modulo-Division `%`, der genau das macht.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Im Körper der inneren Schleife müssen wir nun gucken, ob `check` ein ganzzahliger Teiler von `candidate` ist.
- Das geht, in dem wir den Rest der Division `canddiate`/`check` berechnen.
- Das geht mit der Modulo-Division `%`, der genau das macht.
- Wenn `candidate %check` gleich `0`, dann können wir `candidate` ohne Rest durch `check` teilen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Das geht, in dem wir den Rest der Division `canddiate`/`check` berechnen.
- Das geht mit der Modulo-Division `%`, der genau das macht.
- Wenn `candidate %check` gleich `0`, dann können wir `candidate` ohne Rest durch `check` teilen.
- Dann ist `candidate` durch `check` teilbar und kann keine Primzahl sein.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Das geht mit der Modulo-Division `%`, der genau das macht.
- Wenn `candidate %check` gleich `0`, dann können wir `candidate` ohne Rest durch `check` teilen.
- Dann ist `candidate` durch `check` teilbar und kann keine Primzahl sein.
- Dann können `is_prime = False` setzen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wenn `candidate %check` gleich `0`, dann können wir `candidate` ohne Rest durch `check` teilen.
- Dann ist `candidate` durch `check` teilbar und kann keine Primzahl sein.
- Dann können `is_prime = False` setzen.
- Genaugenommen können wir dann auch gleich die innere Schleife mit `break` abbrechen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199].
```

Beispiel

- Dann ist `candidate` durch `check` teilbar und kann keine Primzahl sein.
- Dann können `is_prime = False` setzen.
- Genaugenommen können wir dann auch gleich die innere Schleife mit `break` abbrechen.
- Sobald wir wissen, dass `candidate` keine Primzahl ist, müssen wir keine weiteren potentiellen Teiler prüfen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
 ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
 ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
 ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Dann können `is_prime = False` setzen.
- Genaugenommen können wir dann auch gleich die innere Schleife mit `break` abbrechen.
- Sobald wir wissen, dass `candidate` keine Primzahl ist, müssen wir keine weiteren potentiellen Teiler prüfen.
- Beachten Sie, dass wir alle Modulo-Divisonen zählen, in dem wir `n_divisions += 1` am Anfang der inneren Schleife machen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199].
```

Beispiel

- Genaugenommen können wir dann auch gleich die innere Schleife mit `break` abbrechen.
- Sobald wir wissen, dass `candidate` keine Primzahl ist, müssen wir keine weiteren potentiellen Teiler prüfen.
- Beachten Sie, dass wir alle Modulo-Divisonen zählen, in dem wir `n_divisions += 1` am Anfang der inneren Schleife machen.
- Nach der inneren Schleife prüfen wir, ob `is_prime` noch `True` ist.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Sobald wir wissen, dass `candidate` keine Primzahl ist, müssen wir keine weiteren potentiellen Teiler prüfen.
- Beachten Sie, dass wir alle Modulo-Divisonen zählen, in dem wir `n_divisions += 1` am Anfang der inneren Schleife machen.
- Nach der inneren Schleife prüfen wir, ob `is_prime` noch `True` ist.
- Wenn ja, dann hängen wir `candidate` an die Liste `primes` an, in dem wir die `append`-Methode der Liste aufrufen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Beachten Sie, dass wir alle Modulo-Divisonen zählen, in dem wir `n_divisions += 1` am Anfang der inneren Schleife machen.
- Nach der inneren Schleife prüfen wir, ob `is_prime` noch `True` ist.
- Wenn ja, dann hängen wir `candidate` an die Liste `primes` an, in dem wir die `append`-Methode der Liste aufrufen.
- Nach dem wir mit der äußeren Schleife fertig sind, drucken wir sowohl die Anzahl `n_divisions` der benötigten Divsionen, die Anzahl `len(primes)` der gefundenen Primzahlen, als auch die Liste `primes` der Primzahlen selbst aus.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Nach der inneren Schleife prüfen wir, ob `is_prime` noch `True` ist.
- Wenn ja, dann hängen wir `candidate` an die Liste `primes` an, in dem wir die `append`-Methode der Liste aufrufen.
- Nach dem wir mit der äußeren Schleife fertig sind, drucken wir sowohl die Anzahl `n_divisions` der benötigten Divsionen, die Anzahl `len(primes)` der gefundenen Primzahlen, als auch die Liste `primes` der Primzahlen selbst aus.
- Wir erfahren, dass wir mit 252 Divisionen alle 46 Primzahlen im Intervall 2..199 finden konnten.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel

- Wenn ja, dann hängen wir `candidate` an die Liste `primes` an, in dem wir die `append`-Methode der Liste aufrufen.
- Nach dem wir mit der äußeren Schleife fertig sind, drucken wir sowohl die Anzahl `n_divisions` der benötigten Divsionen, die Anzahl `len(primes)` der gefundenen Primzahlen, als auch die Liste `primes` der Primzahlen selbst aus.
- Wir erfahren, dass wir mit 252 Divisionen alle 46 Primzahlen im Intervall 2..199 finden konnten.
- (OK, wir haben ignoriert bzw. wissen nicht, ob `isqrt` irgendwelche Divisionen durchführt, aber egal.).

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
 ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
 ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
 ↪ 179, 181, 191, 193, 197, 199].
```

Iterieren über Sequenzen

Iterieren über Sequenzen

- Wir haben bei der Einfüührung der `for`-Schleife gesagt, dass sie über eine *Sequenz* von Daten iteriert.

Iterieren über Sequenzen

- Wir haben bei der Einfüührung der `for`-Schleife gesagt, dass sie über eine *Sequenz* von Daten iteriert.
- Wir haben auch gesagt, dass wir schon verschiedene Kontainerdatentypen kennen, die wir als Sequenzen behandeln können.

Iterieren über Sequenzen

- Wir haben bei der Einfüührung der `for`-Schleife gesagt, dass sie über eine *Sequenz* von Daten iteriert.
- Wir haben auch gesagt, dass wir schon verschiedene Kontainerdatentypen kennen, die wir als Sequenzen behandeln können.
- Wir müssten also über `lists`, `tuples`, `sets`, und `dicts` iterieren können. . .

Beispiel (1)

- Schauen wir uns das mal an.

Beispiel (1)

- Schauen wir uns das mal an.
- In unserem Beispielprogramm bauen wir uns eine Liste `txt` mit Strings zusammen, die wir später ausgeben wollen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Schauen wir uns das mal an.
- In unserem Beispielprogramm bauen wir uns eine Liste `txt` mit Strings zusammen, die wir später ausgeben wollen.
- Zuerst iterieren wir über eine Liste `lst` mit den vier Zahlen `[1, 2, 3, 50]`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Schauen wir uns das mal an.
- In unserem Beispielprogramm bauen wir uns eine Liste `txt` mit Strings zusammen, die wir später ausgeben wollen.
- Zuerst iterieren wir über eine Liste `lst` mit den vier Zahlen `[1, 2, 3, 50]`.
- Das geht mit `for i in lst`...

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- In unserem Beispielprogramm bauen wir uns eine Liste `txt` mit Strings zusammen, die wir später ausgeben wollen.
- Zuerst iterieren wir über eine Liste `lst` mit den vier Zahlen `[1, 2, 3, 50]`.
- Das geht mit `for i in lst`...
- `i` ist die Schleifenvariable und wird Schritt-für-Schritt alle Werde aus `lst` annehmen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Zuerst iterieren wir über eine Liste `lst` mit den vier Zahlen `[1, 2, 3, 50]`.
- Das geht mit `for i in lst`...
- `i` ist die Schleifenvariable und wird Schritt-für-Schritt alle Werde aus `lst` annehmen.
- Im Körper (EN: *body*) der Schleife rufen wir dann `txt.append(f"{i = }")` auf.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Das geht mit `for i in lst`...
- `i` ist die Schleifenvariable und wird Schritt-für-Schritt alle Werde aus `lst` annehmen.
- Im Körper (EN: *body*) der Schleife rufen wir dann `txt.append(f"{i = }")` auf.
- Dieser f-String ergibt `"i = 1"` in der ersten Iteration, `"i = 2"` in der zweiten, `"i = 3"` in der dritten, und `"i = 50"` in der letzten Iteration.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- `i` ist die Schleifenvariable und wird Schritt-für-Schritt alle Werde aus `lst` annehmen.
- Im Körper (EN: *body*) der Schleife rufen wir dann `txt.append(f"{i = }")` auf.
- Dieser f-String ergibt `"i = 1"` in der ersten Iteration, `"i = 2"` in der zweiten, `"i = 3"` in der dritten, und `"i = 50"` in der letzten Iteration.
- Diese Strings werden alle via `append` an die Liste `txt` angehängt.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Im Körper (EN: *body*) der Schleife rufen wir dann `txt.append(f"{i = }")` auf.
- Dieser f-String ergibt `"i = 1"` in der ersten Iteration, `"i = 2"` in der zweiten, `"i = 3"` in der dritten, und `"i = 50"` in der letzten Iteration.
- Diese Strings werden alle via `append` an die Liste `txt` angehängt.
- Machen wir nun mit einem Tupel `tp` mit den viel Fließkommazahlen `(7.6, 9.4, 8.1)` weiter.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():       # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
    ↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
    ↪  False
```

Beispiel (1)

- Dieser f-String ergibt `"i = 1"` in der ersten Iteration, `"i = 2"` in der zweiten, `"i = 3"` in der dritten, und `"i = 50"` in der letzten Iteration.
- Diese Strings werden alle via `append` an die Liste `txt` angehängt.
- Machen wir nun mit einem Tupel `tp` mit den viel Fließkommazahlen `(7.6, 9.4, 8.1)` weiter.
- Hier funktioniert es ganz genauso.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Diese Strings werden alle via `append` an die Liste `txt` angehängt.
- Machen wir nun mit einem Tupel `tp` mit den viel Fließkommazahlen `(7.6, 9.4, 8.1)` weiter.
- Hier funktioniert es ganz genauso.
- `for f in tp` lässt die Schleifenvariable `f` nacheinander die Werte `7.6`, `9.4`, und `8.1` annehmen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():     # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
  ↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
  ↪  False
```

Beispiel (1)

- Machen wir nun mit einem Tupel `tp` mit den viel Fließkommazahlen `(7.6, 9.4, 8.1)` weiter.
- Hier funktioniert es ganz genauso.
- `for f in tp` lässt die Schleifenvariable `f` nacheinander die Werte `7.6`, `9.4`, und `8.1` annehmen.
- Wir verwandeln sie wieder via einem f-String `f"f={f}"` in Text um an hängen diesen via `append` an die Liste `txt` an.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():     # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Hier funktioniert es ganz genauso.
- `for f in tp` lässt die Schleifenvariable `f` nacheinander die Werte `7.6`, `9.4`, und `8.1` annehmen.
- Wir verwandeln sie wieder via einem f-String `f"f={f}"` in Text um an hängen diesen via `append` an die Liste `txt` an.
- Als drittes Beispiel erstellen wir eine Menge `st`, aus drei Zeichenketten als `{"u", "v", "w"}`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():     # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():   # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- `for f in tp` lässt die Schleifenvariable `f` nacheinander die Werte `7.6`, `9.4`, und `8.1` annehmen.
- Wir verwandeln sie wieder via einem f-String `f"f={f}"` in Text um an hängen diesen via `append` an die Liste `txt` an.
- Als drittes Beispiel erstellen wir eine Menge `st`, aus drei Zeichenketten als `{"u", "v", "w"}`.
- Wir können über die Elemente dieser Menge iterieren, in dem wir `for s in st` schreiben.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wir verwandeln sie wieder via einem f-String `f"f={f}"` in Text um an hängen diesen via `append` an die Liste `txt` an.
- Als drittes Beispiel erstellen wir eine Menge `st`, aus drei Zeichenketten als `{"u", "v", "w"}`.
- Wir können über die Elemente dieser Menge iterieren, in dem wir `for s in st` schreiben.
- Die Schleifenvariable `s` nimmt dann die Werte `"w"`, `"u"`, und `"v"` in beliebiger Reihenfolge an.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():       # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Als drittes Beispiel erstellen wir eine Menge `st`, aus drei Zeichenketten als `{"u", "v", "w"}`.
- Wir können über die Elemente dieser Menge iterieren, in dem wir `for s in st` schreiben.
- Die Schleifenvariable `s` nimmt dann die Werte `"w"`, `"u"`, und `"v"` in beliebiger Reihenfolge an.
- Erinnern Sie sich: Mengen sind in Python ungeordnet.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wir können über die Elemente dieser Menge iterieren, in dem wir `for s in st` schreiben.
- Die Schleifenvariable `s` nimmt dann die Werte `"w"`, `"u"`, und `"v"` in beliebiger Reihenfolge an.
- Erinnern Sie sich: Mengen sind in Python ungeordnet.
- Wenn wir das Programm zweimal ausführen, können wir also eventuall verschiedene Reiehenfolgen beobachten.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Die Schleifenvariable `s` nimmt dann die Werte `"w"`, `"u"`, und `"v"` in beliebiger Reihenfolge an.
- Erinnern Sie sich: Mengen sind in Python ungeordnet.
- Wenn wir das Programm zweimal ausführen, können wir also eventuall verschiedene Reiehenfolgen beobachten.
- So oder so, wir können über die Werte in der Menge `st` iterieren.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():     # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
   ↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
   ↪ False
```

Beispiel (1)

- Erinnern Sie sich: Mengen sind in Python ungeordnet.
- Wenn wir das Programm zweimal ausführen, können wir also eventuall verschiedene Reiehenfolgen beobachten.
- So oder so, wir können über die Werte in der Menge `st` iterieren.
- Wieder speichern wir diese als schön formatierte Texte in der Liste `txt`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wenn wir das Programm zweimal ausführen, können wir also eventuall verschiedene Reihenfolgen beobachten.
- So oder so, wir können über die Werte in der Menge `st` iterieren.
- Wieder speichern wir diese als schön formatierte Texte in der Liste `txt`.
- Diesmal benutzen wir den f-String `f"s = {s!r}"`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- So oder so, wir können über die Werte in der Menge `st` iterieren.
- Wieder speichern wir diese als schön formatierte Texte in der Liste `txt`.
- Diesmal benutzen wir den f-String `f"s = {s!r}"`.
- Der `!r` Format-Specifier convertiert die Werte von `s` zu ihrer *Repräsentation*, was im Grunde Anführungszeichen um die Strings setzt (und ggf. Escape-Sequenzen für nicht-druckbare Zeichen benutzt).

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wieder speichern wir diese als schön formatierte Texte in der Liste `txt`.
- Diesmal benutzen wir den f-String `f"s = {s!r}"`.
- Der `!r` Format-Specifier convertiert die Werte von `s` zu ihrer *Repräsentation*, was im Grunde Anführungszeichen um die Strings setzt (und ggf. Escape-Sequenzen für nicht-druckbare Zeichen benutzt).
- So werden `"s = 'v'"`, `"s = 'u'"`, und `"s = 'w'"` an `txt` in beliebiger Reihenfolge angehängt.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪  False
```

Beispiel (1)

- Diesmal benutzen wir den f-String `f"s = {s!r}"`.
- Der `!r` Format-Specifier convertiert die Werte von `s` zu ihrer *Repräsentation*, was im Grunde Anführungszeichen um die Strings setzt (und ggf. Escape-Sequenzen für nicht-druckbare Zeichen benutzt).
- So werden `"s = 'v'"`, `"s = 'u'"`, und `"s = 'w'"` an `txt` in beliebiger Reihenfolge angehängt.
- Als viertes und letztes Beispiel erstellen wir ein Dictionary `dc` das Fließkommazahlen zu Booleschen Werten zurodnet.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():       # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Der `!r` Format-Specifier convertiert die Werte von `s` zu ihrer *Repräsentation*, was im Grunde Anführungszeichen um die Strings setzt (und ggf. Escape-Sequenzen für nicht-druckbare Zeichen benutzt).
- So werden `"s = 'v'"`, `"s = 'u'"`, und `"s = 'w'"` an `txt` in beliebiger Reihenfolge angehängt.
- Als viertes und letztes Beispiel erstellen wir ein Dictionary `dc` das Fließkommazahlen zu Booleschen Werten zurodnet.
- Es beinhaltet nur die beiden Einträge `{1.1: True, 2.5: False}`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- So werden `"s = 'v'"`, `"s = 'u'"`, und `"s = 'w'"` an `txt` in beliebiger Reihenfolge angehängt.
- Als viertes und letztes Beispiel erstellen wir ein Dictionary `dc` das Fließkommazahlen zu Booleschen Werten zurodnet.
- Es beinhaltet nur die beiden Einträge `{1.1: True, 2.5: False}`.
- Dictionaries sind etwas speziell.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Als viertes und letztes Beispiel erstellen wir ein Dictionary `dc` das Fließkommazahlen zu Booleschen Werten zurodnet.
- Es beinhaltet nur die beiden Einträge `{1.1: True, 2.5: False}`.
- Dictionaries sind etwas speziell.
- Sie ordnen Werte zu Schlüsseln zu.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Es beinhaltet nur die beiden Einträge `{1.1: True, 2.5: False}`.
- Dictionaries sind etwas speziell.
- Sie ordnen Werte zu Schlüsseln zu.
- Wenn wir mit dem ganzen Dictionary `dc` als Kollektion arbeiten, dann können wir auf drei Arten darauf zugreifen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
   ↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
   ↪ False
```

Beispiel (1)

- Dictionaries sind etwas speziell.
- Sie ordnen Werte zu Schlüsseln zu.
- Wenn wir mit dem ganzen Dictionary `dc` als Kollektion arbeiten, dann können wir auf drei Arten darauf zugreifen.
- Wenn wir direkt über `dc` iterieren, dann können wir alle Schlüssel sehen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪  False
```

Beispiel (1)

- Sie ordnen Werte zu Schlüsseln zu.
- Wenn wir mit dem ganzen Dictionary `dc` als Kollektion arbeiten, dann können wir auf drei Arten darauf zugreifen.
- Wenn wir direkt über `dc` iterieren, dann können wir alle Schlüssel sehen.
- Das ist das selbe, als über `dc.keys()` zu iterieren.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Wenn wir mit dem ganzen Dictionary `dc` als Kollektion arbeiten, dann können wir auf drei Arten darauf zugreifen.
- Wenn wir direkt über `dc` iterieren, dann können wir alle Schlüssel sehen.
- Das ist das selbe, als über `dc.keys()` zu iterieren.
- Iterieren wir über `dc.values()`, dann sehen wir alle Werte in `dc`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wenn wir direkt über `dc` iterieren, dann können wir alle Schlüssel sehen.
- Das ist das selbe, als über `dc.keys()` zu iterieren.
- Iterieren wir über `dc.values()`, dann sehen wir alle Werte in `dc`.
- Iterieren wir über `dc.items()`, dann sehen wir alle Schlüssel-Wert-Paare als Tupels.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
    ↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
    ↪ False
```

Beispiel (1)

- Das ist das selbe, als über `dc.keys()` zu iterieren.
- Iterieren wir über `dc.values()`, dann sehen wir alle Werte in `dc`.
- Iterieren wir über `dc.items()`, dann sehen wir alle Schlüssel-Wert-Paare als Tupels.
- Wir probieren alle drei Varianten aus.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪  False
```

Beispiel (1)

- Iterieren wir über `dc.values()`, dann sehen wir alle Werte in `dc`.
- Iterieren wir über `dc.items()`, dann sehen wir alle Schlüssel-Wert-Paare als Tupels.
- Wir probieren alle drei Varianten aus.
- Wir iterieren zuerst über die Schlüssel mit `for k in dc`, wodurch `k` erst den Wert `1.1` und dann `2.5` annimmt.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Iterieren wir über `dc.items()`, dann sehen wir alle Schlüssel-Wert-Paare als Tupels.
- Wir probieren alle drei Varianten aus.
- Wir iterieren zuerst über die Schlüssel mit `for k in dc`, wodurch `k` erst den Wert `1.1` und dann `2.5` annimmt.
- Wir iterieren über die Werte mit `for v in dc.values()`, wodurch `v` erst `True` und dann `False` wird.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Wir probieren alle drei Varianten aus.
- Wir iterieren zuerst über die Schlüssel mit `for k in dc`, wodurch `k` erst den Wert `1.1` und dann `2.5` annimmt.
- Wir iterieren über die Werte mit `for v in dc.values()`, wodurch `v` erst `True` und dann `False` wird.
- Zu guter Letzt iterieren wir über die Schlüssel-Wert-Paare.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wir iterieren zuerst über die Schlüssel mit `for k in dc`, wodurch `k` erst den Wert `1.1` und dann `2.5` annimmt.
- Wir iterieren über die Werte mit `for v in dc.values()`, wodurch `v` erst `True` und dann `False` wird.
- Zu guter Letzt iterieren wir über die Schlüssel-Wert-Paare.
- Schauen Sie genau hin!

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Wir iterieren über die Werte mit `for v in dc.values()`, wodurch `v` erst `True` und dann `False` wird.
- Zu guter Letzt iterieren wir über die Schlüssel-Wert-Paare.
- Schauen Sie genau hin!
- Wir *könnten* schreiben `for t in dc.items()`, wodurch wir eine Variable `t` die Werte `(1.1, True)` und dann `2.5: False` annehmen würde.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪  False
```

Beispiel (1)

- Zu guter Letzt iterieren wir über die Schlüssel-Wert-Paare.
- Schauen Sie genau hin!
- Wir *könnten* schreiben `for t in dc.items()`, wodurch wir eine Variable `t` die Werte `(1.1, True)` und dann `2.5: False` annehmen würde.
- Aber wir haben ja vom automatischen „Auspacken“ von Tupeln gelernt.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Schauen Sie genau hin!
- Wir *könnten* schreiben `for t in dc.items()`, wodurch wir eine Variable `t` die Werte `(1.1, True)` und dann `2.5: False` annehmen würde.
- Aber wir haben ja vom automatischen „Auspacken“ von Tupeln gelernt.
- Stattdessen schreiben wir also `for k, v in dc.items()` hin.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Wir *könnten* schreiben `for t in dc.items()`, wodurch wir eine Variable `t` die Werte `(1.1, True)` und dann `2.5: False` annehmen würde.
- Aber wir haben ja vom automatischen „Auspacken“ von Tupeln gelernt.
- Stattdessen schreiben wir also `for k, v in dc.items()` hin.
- Das ist eine sowas wie eine Akürzung für `for t in dc.items()` gefolgt von `k, v = t`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Aber wir haben ja vom automatischen „Auspacken“ von Tupeln gelernt.
- Stattdessen schreiben wir also `for k, v in dc.items()` hin.
- Das ist eine sowas wie eine Akürzung für `for t in dc.items()` gefolgt von `k, v = t`.
- Es packt die Tupels der in der Sequenz `dc.items` direkt aus.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Stattdessen schreiben wir also `for k, v in dc.items()` hin.
- Das ist eine sowas wie eine Akürzung für `for t in dc.items()` gefolgt von `k, v = t`.
- Es packt die Tupels der in der Sequenz `dc.items` direkt aus.
- Wir bekommen also Paare `k=1.1`, `v=True` und `k=2.5`, `v=False`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Das ist eine sowas wie eine Akürzung für `for t in dc.items()` gefolgt von `k, v = t`.
- Es packt die Tupels der in der Sequenz `dc.items` direkt aus.
- Wir bekommen also Paare `k=1.1`, `v=True` und `k=2.5`, `v=False`.
- Und wieder hängen wir diese als Text an unsere Liste `txt` an.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Es packt die Tupels der in der Sequenz `dc.items` direkt aus.
- Wir bekommen also Paare `k=1.1`, `v=True` und `k=2.5`, `v=False`.
- Und wieder hängen wir diese als Text an unsere Liste `txt` an.
- Nach diesen ganzen Schleifen haben wir nun eine Liste `txt` mit 16 Strings.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

Beispiel (1)

- Wir bekommen also Paare `k=1.1`, `v=True` und `k=2.5`, `v=False`.
- Und wieder hängen wir diese als Text an unsere Liste `txt` an.
- Nach diesen ganzen Schleifen haben wir nun eine Liste `txt` mit 16 Strings.
- Wir wollen diese zu einzigen Zeichenkette zusammenführen, wobei wir `", "` als Separator benutzen wollen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
  ↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
  ↪ False
```

Beispiel (1)

- Und wieder hängen wir diese als Text an unsere Liste `txt` an.
- Nach diesen ganzen Schleifen haben wir nun eine Liste `txt` mit 16 Strings.
- Wir wollen diese zu einzigen Zeichenkette zusammenfühgen, wobei wir `", "` als Separator benutzen wollen.
- Das könnten wir mit einer Schleife machen.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():       # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Nach diesen ganzen Schleifen haben wir nun eine Liste `txt` mit 16 Strings.
- Wir wollen diese zu einzigen Zeichenkette zusammenführen, wobei wir `", "` als Separator benutzen wollen.
- Das könnten wir mit einer Schleife machen.
- Python bietet aber eine einfachere und schnellere Methode dafür an.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Wir wollen diese zu einzigen Zeichenkette zusammenführen, wobei wir `", "` als Separator benutzen wollen.
- Das könnten wir mit einer Schleife machen.
- Python bietet aber eine einfachere und schnellere Methode dafür an.
- Die Methode `join` der Klasse `str`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Das könnten wir mit einer Schleife machen.
- Python bietet aber eine einfachere und schnellere Methode dafür an.
- Die Methode `join` der Klasse `str`.
- Für jeden String `z`, akzeptiert `z.join(seq)` eine Sequenz `seq` von Strings.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Python bietet aber eine einfachere und schnellere Methode dafür an.
- Die Methode `join` der Klasse `str`.
- Für jeden String `z`, akzeptiert `z.join(seq)` eine Sequenz `seq` von Strings.
- Es hängt alle Strings in `seq` hintereinander an, wobei jeweils `z` als Separator zwischen zwei Strings eingefügt wird.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Die Methode `join` der Klasse `str`.
- Für jeden String `z`, akzeptiert `z.join(seq)` eine Sequenz `seq` von Strings.
- Es hängt alle Strings in `seq` hintereinander an, wobei jeweils `z` als Separator zwischen zwei Strings eingefügt wird.
- Daher produziert `", ".join(txt)` als Ergebnis `i = 1, i = 2, i = 3, i = 50, f = 7.6, ...`.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (1)

- Für jeden String `z`, akzeptiert `z.join(seq)` eine Sequenz `seq` von Strings.
- Es hängt alle Strings in `seq` hintereinander an, wobei jeweils `z` als Separator zwischen zwei Strings eingefügt wird.
- Daher produziert `", ".join(txt)` als Ergebnis `i = 1, i = 2, i = 3, i = 50, f = 7.6, ...`.
- Dieser Text wird mit der `print` ausgegeben.

```python
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():      # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():    # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u
↪ ', s = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5:
↪ False
```

Beispiel (2)

- Verwenden wir das Konzept der Iteration über Sequenzen nun für etwas Sinnvolles.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Verwenden wir das Konzept der Iteration über Sequenzen nun für etwas Sinnvolles.
- Wir verbessern unser Programm zum Finden von Primzahlen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Verwenden wir das Konzept der Iteration über Sequenzen nun für etwas Sinnvolles.
- Wir verbessern unser Programm zum Finden von Primzahlen.
- Als wir mit dem Originalprogramm angefangen haben, dann haben wir sofort die Zahl `2` als eine Primzahl in unsere Liste `primes` gespeichert.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Verwenden wir das Konzept der Iteration über Sequenzen nun für etwas Sinnvolles.
- Wir verbessern unser Programm zum Finden von Primzahlen.
- Als wir mit dem Originalprogramm angefangen haben, dann haben wir sofort die Zahl `2` als eine Primzahl in unsere Liste `primes` gespeichert.
- Wir haben dann alle ungeraden Zahlen kleiner als 200 als potentielle Prinzahlen `candidates` durchprobiert.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Wir verbessern unser Programm zum Finden von Primzahlen.
- Als wir mit dem Originalprogramm angefangen haben, dann haben wir sofort die Zahl `2` als eine Primzahl in unsere Liste `primes` gespeichert.
- Wir haben dann alle ungeraden Zahlen kleiner als 200 als potentielle Prinzahlen `candidates` durchprobiert.
- Für jede potentielle Primzahl `candidate`, haben wir alle alle (ungeraden) Zahlen `check` aus `range(3, isqrt(candidate)+ 1, 2)` als mögliche Divisoren getestet.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ python3 for_loop_nested_primes.py ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Wir haben dann alle ungeraden Zahlen kleiner als 200 als potentielle Prinzahlen `candidates` durchprobiert.
- Für jede potentielle Primzahl `candidate`, haben wir alle alle (ungeraden) Zahlen `check` aus `range(3, isqrt(candidate)+ 1, 2)` als mögliche Divisoren getestet.
- Wenn wir darüber nachdenken, erkennen wir, dass wir eigentlich nur Primzahlen als mögliche Divisoren testen müssen.

```python
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Für jede potentielle Primzahl `candidate`, haben wir alle alle (ungeraden) Zahlen `check` aus `range(3, isqrt(candidate)+ 1, 2)` als mögliche Divisoren getestet.
- Wenn wir darüber nachdenken, erkennen wir, dass wir eigentlich nur Primzahlen als mögliche Divisoren testen müssen.
- Wir müssen eigentlich niemals prüfen, ob `candidate` durch 9 teilbar ist, denn wir prüfen ja schon, ob es durch 3 geteilt werden kann.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break  # we can stop the inner loop here.
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Für jede potentielle Primzahl `candidate`, haben wir alle alle (ungeraden) Zahlen `check` aus `range(3, isqrt(candidate)+ 1, 2)` als mögliche Divisoren getestet.
- Wenn wir darüber nachdenken, erkennen wir, dass wir eigentlich nur Primzahlen als mögliche Divisoren testen müssen.
- Wir müssen eigentlich niemals prüfen, ob `candidate` durch 9 teilbar ist, denn wir prüfen ja schon, ob es durch 3 geteilt werden kann.
- Wir müssen auch niemals prüfen, ob wir es durch 55 teilen können, weil wir ja schon 5 geprüft haben.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Wenn wir darüber nachdenken, erkennen wir, dass wir eigentlich nur Primzahlen als mögliche Divisoren testen müssen.
- Wir müssen eigentlich niemals prüfen, ob `candidate` durch 9 teilbar ist, denn wir prüfen ja schon, ob es durch 3 geteilt werden kann.
- Wir müssen auch niemals prüfen, ob wir es durch 55 teilen können, weil wir ja schon 5 geprüft haben.
- Aber wie bekommen wir eine Sequenz, in der nur Primzahlen sind?

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Wir müssen eigentlich niemals prüfen, ob `candidate` durch 9 teilbar ist, denn wir prüfen ja schon, ob es durch 3 geteilt werden kann.
- Wir müssen auch niemals prüfen, ob wir es durch 55 teilen können, weil wir ja schon 5 geprüft haben.
- Aber wie bekommen wir eine Sequenz, in der nur Primzahlen sind?
- Hm. Wir bauen die ja gerade selber.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break  # we can stop the inner loop here.
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Wir müssen auch niemals prüfen, ob wir es durch 55 teilen können, weil wir ja schon 5 geprüft haben.
- Aber wie bekommen wir eine Sequenz, in der nur Primzahlen sind?
- Hm. Wir bauen die ja gerade selber.
- `primes` beinhaltet ja alle Primzahlen, die kleiner als `candidate` sind.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Aber wie bekommen wir eine Sequenz, in der nur Primzahlen sind?
- Hm. Wir bauen die ja gerade selber.
- `primes` beinhaltet ja alle Primzahlen, die kleiner als `candidate` sind.
- In einem effizienteren Programm ersetzen wir also einfach den Kopf der inneren Schleife mit `for check in primes:`.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:   # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Hm. Wir bauen die ja gerade selber.
- `primes` beinhaltet ja alle Primzahlen, die kleiner als `candidate` sind.
- In einem effizienteren Programm ersetzen wir also einfach den Kopf der inneren Schleife mit `for check in primes:`.
- Als kleine Performanzverbesserung schreiben wir diesmal 2 nicht gleich am Anfang des Programms in die Liste.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:   # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- `primes` beinhaltet ja alle Primzahlen, die kleiner als `candidate` sind.
- In einem effizienteren Programm ersetzen wir also einfach den Kopf der inneren Schleife mit `for check in primes:`.
- Als kleine Performanzverbesserung schreiben wir diesmal 2 nicht gleich am Anfang des Programms in die Liste.
- Stattdessen fügen wir die 2 am Ende des Programms in die Liste ein.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:   # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- In einem effizienteren Programm ersetzen wir also einfach den Kopf der inneren Schleife mit `for check in primes:`.
- Als kleine Performanzverbesserung schreiben wir diesmal 2 nicht gleich am Anfang des Programms in die Liste.
- Stattdessen fügen wir die 2 am Ende des Programms in die Liste ein.
- Dadurch prüfen wir dann nicht, ob die ungeraden Zahlen `candidate` durch 2 teilbar sind.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:   # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Als kleine Performanzverbesserung schreiben wir diesmal 2 nicht gleich am Anfang des Programms in die Liste.
- Stattdessen fügen wir die 2 am Ende des Programms in die Liste ein.
- Dadurch prüfen wir dann nicht, ob die ungeraden Zahlen `candidate` durch 2 teilbar sind.
- Die äußere Schleife wird Schritt-für-Schritt Primzahlen an `primes` anhängen.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:   # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Stattdessen fügen wir die 2 am Ende des Programms in die Liste ein.
- Dadurch prüfen wir dann nicht, ob die ungeraden Zahlen `candidate` durch 2 teilbar sind.
- Die äußere Schleife wird Schritt-für-Schritt Primzahlen an `primes` anhängen.
- Für jeden Wert der Schleifenvariablen `candidate`, beinhaltet `primes` alle Primzahlen, die kleiner als `candidate` sind (außer 2).

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Dadurch prüfen wir dann nicht, ob die ungeraden Zahlen `candidate` durch 2 teilbar sind.
- Die äußere Schleife wird Schritt-für-Schritt Primzahlen an `primes` anhängen.
- Für jeden Wert der Schleifenvariablen `candidate`, beinhaltet `primes` alle Primzahlen, die kleiner als `candidate` sind (außer 2).
- Natürlich müssen wir nur die Werte von `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
   ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
   ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Die äußere Schleife wird Schritt-für-Schritt Primzahlen an `primes` anhängen.
- Für jeden Wert der Schleifenvariablen `candidate`, beinhaltet `primes` alle Primzahlen, die kleiner als `candidate` sind (außer 2).
- Natürlich müssen wir nur die Werte von `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind.
- Darum speichern wir diesen Wert in einer neuen Variablen `limit`, so dass wir ihn nicht in der inneren Schleife wieder und wieder berechnen müssen.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Für jeden Wert der Schleifenvariablen `candidate`, beinhaltet `primes` alle Primzahlen, die kleiner als `candidate` sind (außer 2).
- Natürlich müssen wir nur die Werte von `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind.
- Darum speichern wir diesen Wert in einer neuen Variablen `limit`, so dass wir ihn nicht in der inneren Schleife wieder und wieder berechnen müssen.
- Mit `break` können wir dann die innere Schleife abbrechen, wenn wir dieses `limit` erreichen.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break  # we can stop the inner loop here.
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
    ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Natürlich müssen wir nur die Werte von `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind.
- Darum speichern wir diesen Wert in einer neuen Variablen `limit`, so dass wir ihn nicht in der inneren Schleife wieder und wieder berechnen müssen.
- Mit `break` können wir dann die innere Schleife abbrechen, wenn wir dieses `limit` erreichen.
- Nach dem die äußere Schleife fertig ist, fügen wir noch die `2` an Index 0 in die Liste ein.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Darum speichern wir diesen Wert in einer neuen Variablen `limit`, so dass wir ihn nicht in der inneren Schleife wieder und wieder berechnen müssen.
- Mit `break` können wir dann die innere Schleife abbrechen, wenn wir dieses `limit` erreichen.
- Nach dem die äußere Schleife fertig ist, fügen wir noch die `2` an Index 0 in die Liste ein.
- Die Liste ist dann genau die selbe, wie beim Originalprogramm.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Beispiel (2)

- Mit `break` können wir dann die innere Schleife abbrechen, wenn wir dieses `limit` erreichen.
- Nach dem die äußere Schleife fertig ist, fügen wir noch die `2` an Index 0 in die Liste ein.
- Die Liste ist dann genau die selbe, wie beim Originalprogramm.
- Allerdings brauchen wir nun nur 224 Divisionen anstatt von 252.

```python
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break          # we can stop the inner loop here.
        n_divisions += 1   # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False        # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  ↪ 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
  ↪ 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  ↪ 179, 181, 191, 193, 197, 199].
```

Zusammenfassung

Zusammenfassung

- Mit `for`-Schleifen sind wir nun wieder ein ganzes Stück näher an „richtiges“ Programmieren gekommen.

Zusammenfassung

- Mit `for`-Schleifen sind wir nun wieder ein ganzes Stück näher an „richtiges“ Programmieren gekommen.
- Wir können jetzt richtige Algorithmen implementieren, wie das Berechnen von Primzahlen.

Zusammenfassung

- Mit `for`-Schleifen sind wir nun wieder ein ganzes Stück näher an „richtiges“ Programmieren gekommen.
- Wir können jetzt richtige Algorithmen implementieren, wie das Berechnen von Primzahlen.
- Wir können mit `for`-Schleifen über Ganzzahl-Sequenzen (`range`) iterieren oder über die Daten in einer Kollektion.

Zusammenfassung

- Mit `for`-Schleifen sind wir nun wieder ein ganzes Stück näher an „richtiges“ Programmieren gekommen.
- Wir können jetzt richtige Algorithmen implementieren, wie das Berechnen von Primzahlen.
- Wir können mit `for`-Schleifen über Ganzzahl-Sequenzen (`range`) iterieren oder über die Daten in einer Kollektion.
- Wir können Schleifen mit `break` frühzeitig abbrechen.

Zusammenfassung

- Mit `for`-Schleifen sind wir nun wieder ein ganzes Stück näher an „richtiges“ Programmieren gekommen.
- Wir können jetzt richtige Algorithmen implementieren, wie das Berechnen von Primzahlen.
- Wir können mit `for`-Schleifen über Ganzzahl-Sequenzen (`range`) iterieren oder über die Daten in einer Kollektion.
- Wir können Schleifen mit `break` frühzeitig abbrechen.
- Wir können auch mit `continue` in die nächste Iteration springen.

Zusammenfassung

- Mit `for`-Schleifen sind wir nun wieder ein ganzes Stück näher an „richtiges“ Programmieren gekommen.
- Wir können jetzt richtige Algorithmen implementieren, wie das Berechnen von Primzahlen.
- Wir können mit `for`-Schleifen über Ganzzahl-Sequenzen (`range`) iterieren oder über die Daten in einer Kollektion.
- Wir können Schleifen mit `break` frühzeitig abbrechen.
- Wir können auch mit `continue` in die nächste Iteration springen.
- Und wir können Schleifen und Alternativen beliebig ineinander verschachteln.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[2] (siehe S. 242, 252).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[1] (siehe S. 242, 252).

[3] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 252, 253).

[4] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 252).

[5] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (besucht am 2025-02-05) (siehe S. 253).

[6] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 251).

[7] Corrado Böhm. "On a Family of Turing Machines and the Related Programming Language". *ICC Bulletin* 3(3):185–194, Juli 1964. Rome, Italy: International Computation Centre (ICC). URL: https://www.cs.tufts.edu/comp/150FP/archive/corrado-bohm/turing-machine-pl.pdf (besucht am 2025-09-01) (siehe S. 5–17).

[8] Corrado Böhm und Guiseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules". *Communications of the ACM (CACM)* 9(5):366–371, Mai 1966. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/355592.365646. URL: https://cs.unibo.it/~martini/PP/bohm-jac.pdf (besucht am 2025-09-01) (siehe S. 5–17).

[9] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 252).

[10] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 252).

References II

[11] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 251).

[12] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 252).

[13] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 252, 253).

[14] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (besucht am 2025-01-09) (siehe S. 253).

[15] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 253).

[16] Edgar Frank „Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 2025-01-05) (siehe S. 253).

[17] Richard Crandall und Carl Pomerance. *Prime Numbers: A Computational Perspective*. 2. Aufl. New York, NY, USA: Springer New York, 4. Aug. 2005. ISBN: 978-0-387-25282-7. doi:10.1007/0-387-28979-8 (siehe S. 103–108).

[18] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 253).

[19] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: https://dataschool.com/how-to-teach-people-sql (besucht am 2025-02-27) (siehe S. 253).

[20] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 253).

[21] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 253).

References III

[22] 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. 251).

[23] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 252).

[24] "Escape Sequences". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 2.4.1.1. URL: https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences (besucht am 2025-08-05) (siehe S. 251).

[25] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (besucht am 2024-09-24). Translation of[26]. (Siehe S. 244).

[26] Leonhard Euler. "De Fractionibus Continuis Dissertation". *Commentarii Academiae Scientiarum Petropolitanae* 9:98–137, 1737–1744. Petropolis (St. Petersburg), Russia: Typis Academiae. URL: https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1070 (besucht am 2024-09-24). See[25] for a translation. (Siehe S. 244, 254).

[27] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 252).

[28] Michael Filaseta. "The Transcendence of e and π". In: *Math 785: Transcendental Number Theory*. Columbia, SC, USA: University of South Carolina, Frühling 2011. Kap. 6. URL: https://people.math.sc.edu/filaseta/gradcourses/Math785/Math785Notes6.pdf (besucht am 2024-07-05) (siehe S. 254).

[29] "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. 252).

References IV

[30] 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. 252).

[31] 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. 252).

[32] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 253).

[33] David Harel. "On Folk Theorems". *Communications of the ACM (CACM)* 23(7):379–389, Juli 1980. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/358886.358892 (siehe S. 5–17).

[34] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 253).

[35] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 252).

[36] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 252, 253).

[37] 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. 252).

[38] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 253).

References V

[39] Arthur Jones, Kenneth R. Pearson und Sidney A. Morris. “Transcendence of e and π”. In: *Abstract Algebra and Famous Impossibilities*. Universitext (UTX). New York, NY, USA: Springer New York, 1991. Kap. 9, S. 115–161. ISSN: 0172-5939. ISBN: 978-1-4419-8552-1. doi:10.1007/978-1-4419-8552-1_8 (siehe S. 254).

[40] Tobias Kohn und Guido van Rossum. *Structural Pattern Matching: Motivation and Rationale*. Python Enhancement Proposal (PEP) 635. Beaverton, OR, USA: Python Software Foundation (PSF), 12. Sep. 2020–8. Feb. 2021. URL: https://peps.python.org/pep-0635 (besucht am 2024-09-23) (siehe S. 35–65).

[41] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 253).

[42] 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. 252).

[43] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. “Client-Server Architecture”. In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: https://www.britannica.com/technology/client-server-architecture (besucht am 2025-01-20) (siehe S. 251).

[44] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O’Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 252).

[45] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 252).

[46] “Mathematical Functions and Operators”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.3. URL: https://www.postgresql.org/docs/17/functions-math.html (besucht am 2025-02-27) (siehe S. 254).

[47] 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. 252).

[48] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 253).

References VI

[49] “More Control Flow Tools”. In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 4. URL: https://docs.python.org/3/tutorial/controlflow.html (besucht am 2025-09-03) (siehe S. 19–24, 28–34, 75–78).

[50] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O’Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 251).

[51] Ivan Niven. “The Transcendence of π”. *The American Mathematical Monthly* 46(8):469–471, Okt. 1939. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.2307/2302515 (siehe S. 254).

[52] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O’Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 252).

[53] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 251).

[54] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: https://www.postgresql.org/docs/17/index.html (besucht am 2025-02-25).

[55] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O’Reilly Media, Inc., März 2024 (siehe S. 252).

[56] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/ 3IEC9899:2017 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. 251).

[57] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 251).

[58] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 252).

References VII

[59] Mike Reichardt, Michael Gundall und Hans D. Schotten. “Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients”. In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 252).

[60] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 251).

[61] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. 2. Aufl. Bd. 126 der Reihe Progress in Mathematics (PM). New York, NY, USA: Springer Science+Business Media, LLC, 1. Okt. 1994–30. Sep. 2012. ISSN: 0743-1643. ISBN: 978-0-8176-3743-9. doi:10.1007/978-1-4612-0251-6. Boston, MA, USA: Birkhäuser (siehe S. 103–108).

[62] Daniel R. Schlegel. “The Böhm-Jacopini Theorem and an Introduction to Structured Programming with Python”. In: *CSE 111: Great Ideas in Computer Science*. Buffalo, NY, USA: Universityat Buffalo (UB), The State University of New York, Sommer 2011. Kap. 5. URL: https://danielschlegel.org/teaching/111/lecture5.html (besucht am 2025-09-01) (siehe S. 5–17).

[63] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 252).

[64] 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. 252).

[65] John Miles Smith und Philip Yen-Tang Chang. “Optimizing the Performance of a Relational Algebra Database Interface”. *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 253).

[66] “SQL Commands”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (besucht am 2025-02-25) (siehe S. 253).

[67] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 249, 253).

References VIII

[68] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of[67] (siehe S. 253).

[69] "String Constants". In: Kap. 4.1.2.1. URL: https://www.postgresql.org/docs/17/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS (besucht am 2025-08-23) (siehe S. 251).

[70] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 253).

[71] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 252).

[72] *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/tutorial (besucht am 2025-04-26).

[73] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 252).

[74] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 252).

[75] Thomas Weise (汤卫思). *Databases*. 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), 2025. URL: https://thomasweise.github.io/databases (besucht am 2025-01-05) (siehe S. 251, 253).

[76] 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. 251, 252).

[77] Eric Wolfgang Weisstein. "Prime Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/PrimeNumber.html (besucht am 2024-09-24) (siehe S. 103–108).

References IX

[78] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: https://www.ibm.com/think/topics/relational-databases (besucht am 2025-01-05) (siehe S. 253).

[79] Ulf Michael „Monty“ Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 252).

[80] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (besucht am 2025-01-11) (siehe S. 251).

[81] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 251).

Glossary (in English) I

Bash is a the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[11,50,81]. Learn more at https://www.gnu.org/software/bash.

C is a programming language, which is very successful in system programming situations[22,56].

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples for clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as psql.

client-server architecture is a system design where a central server receives requests from one or multiple clients[6,43,53,57,60]. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[75].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[80].

escape sequence Escaping is the process of presenting „forbidden“ characters or symbols in a sequence of characters or symbols. In Python[76], string escapes allow us to include otherwise impossible characters, such as string delimiters, in a string. Each such character is represented by an *escape sequence*, which usually starts with the backslash character („\“)[24]. In Python strings, the escape sequence `\"`, for example, stands for `"`, the escape sequence `\\` stands for `\`, and the escape sequence `\n` stands for a newline or linebreak character. In Python f-strings, the escape sequence `{{` stands for a single curly brace `{`. In PostgreSQL[75], similar C-style escapes (starting with „\“) are supported[69].

Glossary (in English) II

f-string let you include the results of expressions in strings[12,29–31,47,64]. 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"..."`.

IT information technology

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[13,36].

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[3,35,63,73,74]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

MariaDB An open source relational database management system that has forked off from MySQL[1,2,4,23,45,58]. See https://mariadb.org for more information.

Microsoft Windows is a commercial proprietary operating system[10]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

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

MySQL An open source relational database management system[9,23,59,71,79]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

PostgreSQL An open source object-relational DBMS[27,52,55,71]. See https://postgresql.org for more information.

psql is the client program used to access the PostgreSQL DBMS server.

Python The Python programming language[37,42,44,76], i.e., what you will learn about in our book[76]. Learn more at https://python.org.

Glossary (in English) III

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other[16,32,34,65,70,75,78].

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers[13] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer"[41].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[14,18–20,38,48,66–68,70]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[66].

(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"`.

terminal A terminal is a text-based window where you can enter commands and execute them[3,15]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl + Alt + T opens a terminal, which then runs a Bash shell inside.

Ubuntu is a variant of the open source operating system Linux[15,36]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

WWW World Wide Web[5,21]

Glossary (in English) IV

π is the ratio of the circumference U of a circle and its diameter d, i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number[28,39,51], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. In Python, it is provided by the `math` module as constant `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793`[46].

$i..j$ 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 $\{5, 6, 7, 8, 9\}$

e is Euler's number[26], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number[28,39], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as constant `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045`[46].

$\mathbb{N}_1$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N}_1 \subset \mathbb{Z}$.

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