合肥大學
HEFEI UNIVERSITY

Programming with Python

25. Schleifen mit `while`

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: 2025-10-30

Programming with Python

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

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

Outline

1. Einleitung
2. Die `while`-Schleife
3. Beispiel
4. `else` am Ende von Schleifen
5. Zusammenfassung

Einleitung

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].

Codex of Saint Gregory Nazianzenos. Greek manuscript of the ninth century CE. Public Domain. Source: [18].

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Dann beginnt dieser Algorithmus mit einer anfänglichen, ersten Annäherung x_0.

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Dann beginnt dieser Algorithmus mit einer anfänglichen, ersten Annäherung x_0.
- Sagen wir, $x_0 = 1$.

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Dann beginnt dieser Algorithmus mit einer anfänglichen, ersten Annäherung x_0.
- Sagen wir, $x_0 = 1$.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i[49,73].

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Dann beginnt dieser Algorithmus mit einer anfänglichen, ersten Annäherung x_0.
- Sagen wir, $x_0 = 1$.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

Quadratwurzel

- Alte Lehmtablets beweisen, dass Babylonier $\sqrt{2}$ vielleicht schon vor 4000 Jahren annähern konnten[32,73].
- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Dann beginnt dieser Algorithmus mit einer anfänglichen, ersten Annäherung x_0.
- Sagen wir, $x_0 = 1$.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Wir können uns grob vorstellen, wie das funktioniert.

Quadratwurzel

- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Dann beginnt dieser Algorithmus mit einer anfänglichen, ersten Annäherung x_0.
- Sagen wir, $x_0 = 1$.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Wir können uns grob vorstellen, wie das funktioniert.
- Ist x_i zu groß, also $x_i > \sqrt{a}$, dann ist $\frac{a}{x_i} < \sqrt{a} < x_i$.

Quadratwurzel

- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- Sagen wir, $x_0 = 1$.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Wir können uns grob vorstellen, wie das funktioniert.
- Ist x_i zu groß, also $x_i > \sqrt{a}$, dann ist $\frac{a}{x_i} < \sqrt{a} < x_i$.
- Ist x_i zu klein, also $x_i < \sqrt{a}$, dann ist $\frac{a}{x_i} > \sqrt{a} > x_i$.

Quadratwurzel

- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Wir können uns grob vorstellen, wie das funktioniert.
- Ist x_i zu groß, also $x_i > \sqrt{a}$, dann ist $\frac{a}{x_i} < \sqrt{a} < x_i$.
- Ist x_i zu klein, also $x_i < \sqrt{a}$, dann ist $\frac{a}{x_i} > \sqrt{a} > x_i$.
- Wir verwenden den Mittelwert von x_i und $\frac{a}{x_i}$ als nächste Annäherung und hofffen, damit näher an $\sqrt{a}$ zu kommen.

Quadratwurzel

- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Ist x_i zu groß, also $x_i > \sqrt{a}$, dann ist $\frac{a}{x_i} < \sqrt{a} < x_i$.
- Ist x_i zu klein, also $x_i < \sqrt{a}$, dann ist $\frac{a}{x_i} > \sqrt{a} > x_i$.
- Wir verwenden den Mittelwert von x_i und $\frac{a}{x_i}$ als nächste Annäherung und hofffen, damit näher an $\sqrt{a}$ zu kommen.
- Wenn $x_i = \sqrt{a}$, dann gilt $\frac{a}{x_i} = \sqrt{a}$ und $x_{i+1} = x_i$.

Quadratwurzel

- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Ist x_i zu klein, also $x_i < \sqrt{a}$, dann ist $\frac{a}{x_i} > \sqrt{a} > x_i$.
- Wir verwenden den Mittelwert von x_i und $\frac{a}{x_i}$ als nächste Annäherung und hofffen, damit näher an $\sqrt{a}$ zu kommen.
- Wenn $x_i = \sqrt{a}$, dann gilt $\frac{a}{x_i} = \sqrt{a}$ und $x_{i+1} = x_i$.
- Tatsächlich zu beweisen, dass das funktoniert und dass der Fehler tatsächlich kleiner wird ist kompliziert[73].

Quadratwurzel

- Der Mathematiker *Hero(n) von Alexandria* hat im ersten Jahrhundert Common Era (CE) gelebt. Er hat einen Algorithmus zum Berechnen der Quadratwurzel entwickelt, den wir heute als Heron's Methode kennen[49,73].
- Angenommen, wir wollen die Quadratwurzel $\sqrt{a}$ einer Zahl a finden.
- In jeder Iteration berechnet der Algorithmus eine neue Annäherung x_{i+1} basierend auf der aktuellen Schätzung x_i wie folgt[49,73]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{1}$$

- Wir verwenden den Mittelwert von x_i und $\frac{a}{x_i}$ als nächste Annäherung und hofffen, damit näher an $\sqrt{a}$ zu kommen.
- Wenn $x_i = \sqrt{a}$, dann gilt $\frac{a}{x_i} = \sqrt{a}$ und $x_{i+1} = x_i$.
- Tatsächlich zu beweisen, dass das funktoniert und dass der Fehler tatsächlich kleiner wird ist kompliziert[73].
- (Aber das machen wir hier nicht.)

Die `while`-Schleife

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.
- Es ist offensichtlich, dass wir die selbe Sache immer wieder machen wollen.

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.
- Es ist offensichtlich, dass wir die selbe Sache immer wieder machen wollen.
- Wir werden die Gleichung $x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$ wieder und wieder ausrechnen.

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.
- Es ist offensichtlich, dass wir die selbe Sache immer wieder machen wollen.
- Wir werden die Gleichung $x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$ wieder und wieder ausrechnen.
- Allerdings wird eine `for`-Schleife hier nicht funktionieren.

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.
- Es ist offensichtlich, dass wir die selbe Sache immer wieder machen wollen.
- Wir werden die Gleichung $x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$ wieder und wieder ausrechnen.
- Allerdings wird eine `for`-Schleife hier nicht funktionieren.
- Wir wissen ja nicht, wie viele Schritte wir brauchen, bis $x_i = x_{i+1}$…

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.
- Es ist offensichtlich, dass wir die selbe Sache immer wieder machen wollen.
- Wir werden die Gleichung $x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$ wieder und wieder ausrechnen.
- Allerdings wird eine `for`-Schleife hier nicht funktionieren.
- Wir wissen ja nicht, wie viele Schritte wir brauchen, bis $x_i = x_{i+1}$...
- Klar, wir könnten einfach eine sehr sehr große Zahl für die Anzahl der Iterationen nehmen und die Schleife mit `break` abbrechen ... aber das ist einfach zu häßlich.

Die `while`-Schleife

- Wenn wir Heron's Methode implementieren wollen, dann brauchen wir irgendeine Art Schleife.
- Es ist offensichtlich, dass wir die selbe Sache immer wieder machen wollen.
- Wir werden die Gleichung $x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$ wieder und wieder ausrechnen.
- Allerdings wird eine `for`-Schleife hier nicht funktionieren.
- Wir wissen ja nicht, wie viele Schritte wir brauchen, bis $x_i = x_{i+1}\ldots$
- Klar, wir könnten einfach eine sehr sehr große Zahl für die Anzahl der Iterationen nehmen und die Schleife mit `break` abbrechen ... aber das ist einfach zu häßlich.
- Wir brauchen ein `while`-Schleife[61].

Die while-Schleife

- Wir brauchen ein `while`-Schleife[61].
- Wie die `for`-Schleife besteht eine `while`-Schleife aus einem Kopf und einem Körper.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Wir brauchen ein `while`-Schleife[61].
- Wie die `for`-Schleife besteht eine `while`-Schleife aus einem Kopf und einem Körper.
- Der Kopf beginnt mit `while` gefolgt von der Schleifenbedingung `booleanExpression` und ended mit dem Doppelpunkt `:`.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Wir brauchen ein `while`-Schleife[61].
- Wie die `for`-Schleife besteht eine `while`-Schleife aus einem Kopf und einem Körper.
- Der Kopf beginnt mit `while` gefolgt von der Schleifenbedingung `booleanExpression` und ended mit dem Doppelpunkt `:`.
- Der Schleifenkörper ist wieder ein mit vier Leerzeichen eingerückter Kodeblock.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Wie die `for`-Schleife besteht eine `while`-Schleife aus einem Kopf und einem Körper.
- Der Kopf beginnt mit `while` gefolgt von der Schleifenbedingung `booleanExpression` und ended mit dem Doppelpunkt `:`.
- Der Schleifenkörper ist wieder ein mit vier Leerzeichen eingerückter Kodeblock.
- Jedesmal bevor der Schleifenkörper ausgeführt wird, wird die Schleifenbedingung `booleanExpression` ausgerechnet.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Der Kopf beginnt mit `while` gefolgt von der Schleifenbedingung `booleanExpression` und ended mit dem Doppelpunkt `:`.
- Der Schleifenkörper ist wieder ein mit vier Leerzeichen eingerückter Kodeblock.
- Jedesmal bevor der Schleifenkörper ausgeführt wird, wird die Schleifenbedingung `booleanExpression` ausgerechnet.
- Nur wenn sie `True` ergibt, wird der Schleifenkörper ausgeführt.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Der Schleifenkörper ist wieder ein mit vier Leerzeichen eingerückter Kodeblock.
- Jedesmal bevor der Schleifenkörper ausgeführt wird, wird die Schleifenbedingung `booleanExpression` ausgerechnet.
- Nur wenn sie `True` ergibt, wird der Schleifenkörper ausgeführt.
- Nur dann kann es weitere Iteratinen geben, vor denen jeweils wieder die Schleifenbedingung ausgerechnet wird.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Jedesmal bevor der Schleifenkörper ausgeführt wird, wird die Schleifenbedingung `booleanExpression` ausgerechnet.
- Nur wenn sie `True` ergibt, wird der Schleifenkörper ausgeführt.
- Nur dann kann es weitere Iteratinen geben, vor denen jeweils wieder die Schleifenbedingung ausgerechnet wird.
- Wenn die Schleifenbedingung nicht wahr ergab, dann wird die Schleife sofort abgebrochen.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Die `while`-Schleife

- Nur wenn sie `True` ergibt, wird der Schleifenkörper ausgeführt.
- Nur dann kann es weitere Iteratinen geben, vor denen jeweils wieder die Schleifenbedingung ausgerechnet wird.
- Wenn die Schleifenbedingung nicht wahr ergab, dann wird die Schleife sofort abgebrochen.
- In anderen Worten, der Schleifenkörper wird so lange aus geführt wie der Ausdruck im Schleifenkopf wahr ist.

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

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Beispiel

Implementierung von Heron's Methode

- Implementieren wir nun Heron's Methode, um die Quadratwurzeln von 0.5, 2, und 3 zu berechnen.

Implementierung von Heron's Methode

- Implementieren wir nun Heron's Methode, um die Quadratwurzeln von 0.5, 2, und 3 zu berechnen.
- Wir beginning unser Programm mit einer äußeren `for`-Schleife, die ein Variable `number` über die Fließkommawerte `0.5`, `2.0`, und `3.0` iterieren lässt.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Implementieren wir nun Heron's Methode, um die Quadratwurzeln von 0.5, 2, und 3 zu berechnen.
- Wir beginning unser Programm mit einer äußeren `for`-Schleife, die ein Variable `number` über die Fließkommawerte `0.5`, `2.0`, und `3.0` iterieren lässt.
- Wir wollen den Algorithmus auf jeden dieser Werte anwenden.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess       # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Implementieren wir nun Heron's Methode, um die Quadratwurzeln von 0.5, 2, und 3 zu berechnen.
- Wir beginning unser Programm mit einer äußeren `for`-Schleife, die ein Variable `number` über die Fließkommawerte `0.5`, `2.0`, und `3.0` iterieren lässt.
- Wir wollen den Algorithmus auf jeden dieser Werte anwenden.
- Wir benutzen die beiden Variablen `guess` und `old_guess`.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Wir beginning unser Programm mit einer äußeren `for`-Schleife, die ein Variable `number` über die Fließkommawerte `0.5`, `2.0`, und `3.0` iterieren lässt.
- Wir wollen den Algorithmus auf jeden dieser Werte anwenden.
- Wir benutzen die beiden Variablen `guess` und `old_guess`.
- `guess` ist unsere aktuelle Annäherung von $\sqrt{\texttt{number}}$.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):  # Repeat until no change.
        old_guess = guess     # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- Wir wollen den Algorithmus auf jeden dieser Werte anwenden.
- Wir benutzen die beiden Variablen `guess` und `old_guess`.
- `guess` ist unsere aktuelle Annäherung von $\sqrt{\texttt{number}}$.
- `old_guess` ist die vorherige Annäherung, die wir uns merken müssen, damit wir sehen, wann wir fertig sind.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):  # Repeat until no change.
        old_guess = guess     # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Wir benutzen die beiden Variablen `guess` und `old_guess`.
- `guess` ist unsere aktuelle Annäherung von $\sqrt{\texttt{number}}$.
- `old_guess` ist die vorherige Annäherung, die wir uns merken müssen, damit wir sehen, wann wir fertig sind.
- Wir initialisieren `guess` mit `1.0` und `old_guess` mit einem anderen Wert, sagen wir `0.0`.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt      # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess       # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- `guess` ist unsere aktuelle Annäherung von $\sqrt{\texttt{number}}$.
- `old_guess` ist die vorherige Annäherung, die wir uns merken müssen, damit wir sehen, wann wir fertig sind.
- Wir initialisieren `guess` mit `1.0` und `old_guess` mit einem anderen Wert, sagen wir `0.0`.
- Unsere `while`-Schleife soll dan iterieren, so lange `guess != old_guess` und in jedem Schritt die Annäherung der Wurzel updaten.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- `old_guess` ist die vorherige Annäherung, die wir uns merken müssen, damit wir sehen, wann wir fertig sind.
- Wir initialisieren `guess` mit `1.0` und `old_guess` mit einem anderen Wert, sagen wir `0.0`.
- Unsere `while`-Schleife soll dan iterieren, so lange `guess != old_guess` und in jedem Schritt die Annäherung der Wurzel updaten.
- Die Schleifenbedingung hier ist sehr interessant und wichtig.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Wir initialisieren `guess` mit `1.0` und `old_guess` mit einem anderen Wert, sagen wir `0.0`.
- Unsere `while`-Schleife soll dan iterieren, so lange `guess != old_guess` und in jedem Schritt die Annäherung der Wurzel updaten.
- Die Schleifenbedingung hier ist sehr interessant und wichtig.
- Denken wir mal darüber nach.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Unsere `while`-Schleife soll dan iterieren, so lange `guess != old_guess` und in jedem Schritt die Annäherung der Wurzel updaten.
- Die Schleifenbedingung hier ist sehr interessant und wichtig.
- Denken wir mal darüber nach.
- Könnten wir die reellen Zahlen $\mathbb{R}$ mit unendlicher Genauigkeit darstellen, dann würde so eine Schleife für Zahlen `number` mit irrationalen Wurzen niemals aufhören, denn wir würden dann ja niemanls `guess == old_guess` erreichen.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt      # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- Die Schleifenbedingung hier ist sehr interessant und wichtig.
- Denken wir mal darüber nach.
- Könnten wir die reellen Zahlen $\mathbb{R}$ mit unendlicher Genauigkeit darstellen, dann würde so eine Schleife für Zahlen `number` mit irrationalen Wurzen niemals aufhören, denn wir würden dann ja niemanls `guess == old_guess` erreichen.
- Für irrationale Wurzen würde unser Algorithmus niemals terminieren.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Denken wir mal darüber nach.
- Könnten wir die reellen Zahlen $\mathbb{R}$ mit unendlicher Genauigkeit darstellen, dann würde so eine Schleife für Zahlen `number` mit irrationalen Wurzen niemals aufhören, denn wir würden dann ja niemanls `guess == old_guess` erreichen.
- Für irrationale Wurzen würde unser Algorithmus niemals terminieren.
- Aber wir haben nunmal den Datentyp `float` mit begrenzer Präzision.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ python3 while_loop_sqrt.py ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- Könnten wir die reellen Zahlen $\mathbb{R}$ mit unendlicher Genauigkeit darstellen, dann würde so eine Schleife für Zahlen `number` mit irrationalen Wurzen niemals aufhören, denn wir würden dann ja niemanls `guess == old_guess` erreichen.
- Für irrationale Wurzen würde unser Algorithmus niemals terminieren.
- Aber wir haben nunmal den Datentyp `float` mit begrenzer Präzision.
- Wir arbeiten also mit begrenzer Präzision und erreichen irgendwann den Punkt, wo wir unsere Annäherungsgenauigkeit nicht mehr erhöhen können.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Für irrationale Wurzen würde unser Algorithmus niemals terminieren.
- Aber wir haben nunmal den Datentyp `float` mit begrenzer Präzision.
- Wir arbeiten also mit begrenzer Präzision und erreichen irgendwann den Punkt, wo wir unsere Annäherungsgenauigkeit nicht mehr erhöhen können.
- Wir sollten also tatsächlich immer irgendwann `guess == old_guess` erreichen.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ python3 while_loop_sqrt.py ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Aber wir haben nunmal den Datentyp `float` mit begrenzer Präzision.
- Wir arbeiten also mit begrenzer Präzision und erreichen irgendwann den Punkt, wo wir unsere Annäherungsgenauigkeit nicht mehr erhöhen können.
- Wir sollten also tatsächlich immer irgendwann `guess == old_guess` erreichen.
- Die Unperfektheit der Fließkommazahlen führt jedoch zu einem unerwarteten Problem.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess     # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- Die Unperfektheit der Fließkommazahlen führt jedoch zu einem unerwarteten Problem.

Gute Praxis

Aufgrund der begrenzten Auflösung von Fließkommazahlen wird vom direkten Vergleichen von Ergebnissen von Berechnungen mit Fließkommawerten mit den strengen Operatoren `==` und `!=` abgeraten[4,51].

Implementierung von Heron's Methode

- Die Unperfektheit der Fließkommazahlen führt jedoch zu einem unerwarteten Problem.

Gute Praxis

Aufgrund der begrenzten Auflösung von Fließkommazahlen wird vom direkten Vergleichen von Ergebnissen von Berechnungen mit Fließkommawerten mit den strengen Operatoren `==` und `!=` abgeraten[4,51]. Das kann nämlich zu unerwarteten Ergebnissen führen.

Implementierung von Heron's Methode

- Die Unperfektheit der Fließkommazahlen führt jedoch zu einem unerwarteten Problem.

Gute Praxis

Aufgrund der begrenzten Auflösung von Fließkommazahlen wird vom direkten Vergleichen von Ergebnissen von Berechnungen mit Fließkommawerten mit den strengen Operatoren `==` und `!=` abgeraten[4,51]. Das kann nämlich zu unerwarteten Ergebnissen führen. Zum Beispiel ergibt `(0.1 + 0.2)== 0.3` nämlich `False`.

Implementierung von Heron's Methode

- Die Unperfektheit der Fließkommazahlen führt jedoch zu einem unerwarteten Problem.

Gute Praxis

Aufgrund der begrenzten Auflösung von Fließkommazahlen wird vom direkten Vergleichen von Ergebnissen von Berechnungen mit Fließkommawerten mit den strengen Operatoren `==` und `!=` abgeraten[4,51]. Das kann nämlich zu unerwarteten Ergebnissen führen. Zum Beispiel ergibt `(0.1 + 0.2)== 0.3` nämlich `False`. Funktionen wir `isclose` aus dem Modul `math`, die prüfen ob zwei Fließkommawerte basierend auf ihren relativen und absoluten Unterschieden annähernd gleich sind, können dieses Problem (zumindest etwas[35]) abschwächen[4].

Implementierung von Heron's Methode

- Wenn uns direkte Gleichheits- oder Ungleichheitsvergleiche von `floats` als Bedingungen bei `ifs` in Probleme bringen können, dann wird es Sie nicht überraschen, dass das in `whiles` nicht anders ist.

Gute Praxis

Aufgrund der begrenzten Auflösung von Fließkommazahlen wird vom direkten Vergleichen von Ergebnissen von Berechnungen mit Fließkommawerten mit den strengen Operatoren `==` und `!=` abgeraten[4,51]. Das kann nämlich zu unerwarteten Ergebnissen führen. Zum Beispiel ergibt `(0.1 + 0.2)== 0.3` nämlich `False`. Funktionen wir `isclose` aus dem Modul `math`, die prüfen ob zwei Fließkommawerte basierend auf ihren relativen und absoluten Unterschieden annähernd gleich sind, können dieses Problem (zumindest etwas[35]) abschwächen[4].

Implementierung von Heron's Methode

- Wenn uns direkte Gleichheits- oder Ungleichheitsvergleiche von `floats` als Bedingungen bei `ifs` in Probleme bringen können, dann wird es Sie nicht überraschen, dass das in `whiles` nicht anders ist.

Gute Praxis

Benutzen Sie keine strengen Gleichheits- oder Ungleichheits-Vergleiche von `floats` als Abbruchkriterien von Schleifen[51], denn sie können leicht zu Endlosschleifen führen, die niemals abbrechen.

Implementierung von Heron's Methode

- Wenn uns direkte Gleichheits- oder Ungleichheitsvergleiche von `floats` als Bedingungen bei `ifs` in Probleme bringen können, dann wird es Sie nicht überraschen, dass das in `whiles` nicht anders ist.

Gute Praxis

Benutzen Sie keine strengen Gleichheits- oder Ungleichheits-Vergleiche von `floats` als Abbruchkriterien von Schleifen[51], denn sie können leicht zu Endlosschleifen führen, die niemals abbrechen. Es kann immer komische Eingabewerte geben, die zu einer endlosen Oszillation zwischen Werten oder dem Auftauchen von `nan` führen können.

Implementierung von Heron's Methode

- Wenn uns direkte Gleichheits- oder Ungleichheitsvergleiche von `floats` als Bedingungen bei `ifs` in Probleme bringen können, dann wird es Sie nicht überraschen, dass das in `whiles` nicht anders ist.

Gute Praxis

Benutzen Sie keine strengen Gleichheits- oder Ungleichheits-Vergleiche von `floats` als Abbruchkriterien von Schleifen[51], denn sie können leicht zu Endlosschleifen führen, die niemals abbrechen. Es kann immer komische Eingabewerte geben, die zu einer endlosen Oszillation zwischen Werten oder dem Auftauchen von `nan` führen können. Das erste Problem kann zumindest wieder teilweise durch annäherungsweises Vergleichen mit Funktionen wie `isclose` abgeschwöächt werden[4].

Implementierung von Heron's Methode

- Wenn uns direkte Gleichheits- oder Ungleichheitsvergleiche von `floats` als Bedingungen bei `ifs` in Probleme bringen können, dann wird es Sie nicht überraschen, dass das in `whiles` nicht anders ist.
- OK, also benutzen wir `guess != old_guess` lieber nicht als Schleifenkriterium.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- OK, also benutzen wir `guess != old_guess` lieber nicht als Schleifenkriterium.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- OK, also benutzen wir `guess != old_guess` lieber nicht als Schleifenkriterium.
- Wir importieren Funktion `isclose` aus dem Modul `math`.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- OK, also benutzen wir `guess != old_guess` lieber nicht als Schleifenkriterium.
- Wir importieren Funktion `isclose` aus dem Modul `math`.
- Wir schreiben `not isclose(guess, old_guess)` anstatt von `guess != old_guess`.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):  # Repeat until no change.
        old_guess = guess     # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- OK, also benutzen wir `guess != old_guess` lieber nicht als Schleifenkriterium.
- Wir importieren Funktion `isclose` aus dem Modul `math`.
- Wir schreiben `not isclose(guess, old_guess)` anstatt von `guess != old_guess`.
- Die Funktion `isclose` betrachtet `guess` und `old_guess` als gleich wenn deren relativer Unterschied weniger als ein Teil pro Milliarde beträgt[4].

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- Wir importieren Funktion `isclose` aus dem Modul `math`.
- Wir schreiben `not isclose(guess, old_guess)` anstatt von `guess != old_guess`.
- Die Funktion `isclose` betrachtet `guess` und `old_guess` als gleich wenn deren relativer Unterschied weniger als ein Teil pro Milliarde beträgt[4].
- Unsere Schleife wird also abbrechen, sobald die aktuelle und die vorherige Annäherung annähernd gleich sind.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Wir schreiben `not isclose(guess, old_guess)` anstatt von `guess != old_guess`.
- Die Funktion `isclose` betrachtet `guess` und `old_guess` als gleich wenn deren relativer Unterschied weniger als ein Teil pro Milliarde beträgt[4].
- Unsere Schleife wird also abbrechen, sobald die aktuelle und die vorherige Annäherung annähernd gleich sind.
- Der Körper der Schleife ist dann ganz einfach.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess     # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Die Funktion `isclose` betrachtet `guess` und `old_guess` als gleich wenn deren relativer Unterschied weniger als ein Teil pro Milliarde beträgt[4].
- Unsere Schleife wird also abbrechen, sobald die aktuelle und die vorherige Annäherung annähernd gleich sind.
- Der Körper der Schleife ist dann ganz einfach.
- Erstmal speichern wir die aktuelle Annäherung in der alten via `old_guess = guess`.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Unsere Schleife wird also abbrechen, sobald die aktuelle und die vorherige Annäherung annähernd gleich sind.
- Der Körper der Schleife ist dann ganz einfach.
- Erstmal speichern wir die aktuelle Annäherung in der alten via `old_guess = guess`.
- Dann updaten wir die Annhäherung nach Heron's Formel, also in dem wir `guess` gleich `0.5 * (guess + number/guess)` setzen.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt      # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0         # This will hold the current guess.
    old_guess: float = 0.0     # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):  # Repeat until no change.
        old_guess = guess     # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Der Körper der Schleife ist dann ganz einfach.
- Erstmal speichern wir die aktuelle Annäherung in der alten via `old_guess = guess`.
- Dann updaten wir die Annhäherung nach Heron's Formel, also in dem wir `guess` gleich `0.5 * (guess + number/guess)` setzen.
- Fertig. Wir haben Heron's Methode implementiert.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt      # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0              # This will hold the current guess.
    old_guess: float = 0.0          # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess       # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ python3 while_loop_sqrt.py ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Erstmal speichern wir die aktuelle Annäherung in der alten via `old_guess = guess`.
- Dann updaten wir die Annhäherung nach Heron's Formel, also in dem wir `guess` gleich `0.5 * (guess + number/guess)` setzen.
- Fertig. Wir haben Heron's Methode implementiert.
- Jetzt drucken wir noch die Ergebnisse der Berechnungen aus.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ python3 while_loop_sqrt.py ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron’s Methode

- Dann updaten wir die Annhäherung nach Heron’s Formel, also in dem wir `guess` gleich `0.5 * (guess + number/guess)` setzen.
- Fertig. Wir haben Heron’s Methode implementiert.
- Jetzt drucken wir noch die Ergebnisse der Berechnungen aus.
- Zum Vergleich drucken wir noch die Ergebnisse der Funktion `sqrt` aus dem Modul `math` mit dazu.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Fertig. Wir haben Heron's Methode implementiert.
- Jetzt drucken wir noch die Ergebnisse der Berechnungen aus.
- Zum Vergleich drucken wir noch die Ergebnisse der Funktion `sqrt` aus dem Modul `math` mit dazu.
- Wie Sie sehen liefert unser Algorithmus jeweils fast das gleiche Ergebnis.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

Implementierung von Heron's Methode

- Jetzt drucken wir noch die Ergebnisse der Berechnungen aus.
- Zum Vergleich drucken wir noch die Ergebnisse der Funktion `sqrt` aus dem Modul `math` mit dazu.
- Wie Sie sehen liefert unser Algorithmus jeweils fast das gleiche Ergebnis.
- Beachten Sie auch, wie wir mir Unicode Escape-Sequenzen die Sonderzeichen $\sqrt{\cdot}$ und $\approx$ als `\u221A` und `\u2248` ausdrücken, um sie schön in unser Terminal gedruckt zu bekommen.

```python
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0           # This will hold the current guess.
    old_guess: float = 0.0       # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

else am Ende von Schleifen

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.
- Wir würden sie mit `ok = True` initialisieren.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.
- Wir würden sie mit `ok = True` initialisieren.
- Wenn wir `break` aufrufen wollen, dann setzten wir `ok` erstmal auf `False` und machen dann `break`.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.
- Wir würden sie mit `ok = True` initialisieren.
- Wenn wir `break` aufrufen wollen, dann setzten wir `ok` erstmal auf `False` und machen dann `break`.
- Nach der Schleife würden wir die Aktion $\mathcal{A}$ dann in ein `if` packen und einfach `ok` als Bedingung nehmen.

`else` am Ende von Schleifen

- Sie haben gelernt, dass wir eine Schleife immer mit `break` beenden können.
- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.
- Wir würden sie mit `ok = True` initialisieren.
- Wenn wir `break` aufrufen wollen, dann setzten wir `ok` erstmal auf `False` und machen dann `break`.
- Nach der Schleife würden wir die Aktion $\mathcal{A}$ dann in ein `if` packen und einfach `ok` als Bedingung nehmen.
- Das ist absolut OK ... aber Python bietet uns eine viel einfachere Methode an.

`else` am Ende von Schleifen

- Sagen wir, Sie wollen eine Aktion $\mathcal{A}$ ausführen, wenn eine Schleife normal beendet wurde, also wenn `break` nicht benutzt wurde.
- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.
- Wir würden sie mit `ok = True` initialisieren.
- Wenn wir `break` aufrufen wollen, dann setzten wir `ok` erstmal auf `False` und machen dann `break`.
- Nach der Schleife würden wir die Aktion $\mathcal{A}$ dann in ein `if` packen und einfach `ok` als Bedingung nehmen.
- Das ist absolut OK … aber Python bietet uns eine viel einfachere Methode an.
- Wir können einen `else`-Block ans Ende der Schleife schreiben und der wird dann nur ausgeführt, wenn die Schleife normal beendet wurde.

`else` am Ende von Schleifen

- Das können wir auch jetzt schon.
- Wir können eine Boolesche Variable `ok` deklarieren die wahr ist, wenn die Schleife normal beendet wurde.
- Wir würden sie mit `ok = True` initialisieren.
- Wenn wir `break` aufrufen wollen, dann setzten wir `ok` erstmal auf `False` und machen dann `break`.
- Nach der Schleife würden wir die Aktion $\mathcal{A}$ dann in ein `if` packen und einfach `ok` als Bedingung nehmen.
- Das ist absolut OK ... aber Python bietet uns eine viel einfachere Methode an.
- Wir können einen `else`-Block ans Ende der Schleife schreiben und der wird dann nur ausgeführt, wenn die Schleife normal beendet wurde.
- Das geht sowohl mit `for` als auch mit `while`.

else am Ende von Schleifen

- Wir können einen `else`-Block ans Ende der Schleife schreiben und der wird dann nur ausgeführt, wenn die Schleife normal beendet wurde.
- Das geht sowohl mit `for` als auch mit `while`.

```python
"""The syntax of a for-loop with else staement 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.
    # ...
else:
    else statement 1  # The body of the 'else' block is only executed if
    else statement 2  # 'break' was never invoked in the for loop body.
    # ...

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

else am Ende von Schleifen

- Wir können einen `else`-Block ans Ende der Schleife schreiben und der wird dann nur ausgeführt, wenn die Schleife normal beendet wurde.
- Das geht sowohl mit `for` als auch mit `while`.

```python
"""The syntax of a while-loop with else statement in Python."""

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...
else:
    else statement 1  # The body of the 'else' block is only executed if
    else statement 2  # 'break' was never invoked in the while loop.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Beispiel: Binäre Suche

- Wir benutzen nun dieses Konstrukt, um eine binäre Suche[7,38,48] zu implementieren.

Beispiel: Binäre Suche

- Wir benutzen nun dieses Konstrukt, um eine binäre Suche[7,38,48] zu implementieren.
- Binäre Suche findet den Index eines Elements in einer *sortierten* Sequence `data` von Werten.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir benutzen nun dieses Konstrukt, um eine binäre Suche[7,38,48] zu implementieren.
- Binäre Suche findet den Index eines Elements in einer *sortierten* Sequence `data` von Werten.
- Das Kernkonzept von binärer Suche ist, dass wir immer ein Segment S der Liste betrachten, in dem wir das gesuchte Element E vermuten.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir benutzen nun dieses Konstrukt, um eine binäre Suche[7,38,48] zu implementieren.
- Binäre Suche findet den Index eines Elements in einer *sortierten* Sequence `data` von Werten.
- Das Kernkonzept von binärer Suche ist, dass wir immer ein Segment S der Liste betrachten, in dem wir das gesuchte Element E vermuten.
- In jedem Schritt wollen wir die Größe des Segments verringern, in dem wir die *Hälfte*, in der E nicht seien kann, ausschlißen.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Binäre Suche findet den Index eines Elements in einer *sortierten* Sequence `data` von Werten.
- Das Kernkonzept von binärer Suche ist, dass wir immer ein Segment S der Liste betrachten, in dem wir das gesuchte Element E vermuten.
- In jedem Schritt wollen wir die Größe des Segments verringern, in dem wir die *Hälfte*, in der E nicht seien kann, ausschlißen.
- Wir tuen das, in dem wir uns das Element M in der Mitte von S angucken.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Das Kernkonzept von binärer Suche ist, dass wir immer ein Segment S der Liste betrachten, in dem wir das gesuchte Element E vermuten.
- In jedem Schritt wollen wir die Größe des Segments verringern, in dem wir die *Hälfte*, in der E nicht seien kann, ausschlißen.
- Wir tuen das, in dem wir uns das Element M in der Mitte von S angucken.
- Die gesamte Sequenz `data` und daher auch das Segment S sind sortiert.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- In jedem Schritt wollen wir die Größe des Segments verringern, in dem wir die *Hälfte*, in der E nicht seien kann, ausschlißen.
- Wir tuen das, in dem wir uns das Element M in der Mitte von S angucken.
- Die gesamte Sequenz `data` und daher auch das Segment S sind sortiert.
- Wenn M größer als E ist, dann kann E nur in der ersten Hälfte von S, nämlich in dem Teil vor M.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir tuen das, in dem wir uns das Element M in der Mitte von S angucken.
- Die gesamte Sequenz `data` und daher auch das Segment S sind sortiert.
- Wenn M größer als E ist, dann kann E nur in der ersten Hälfte von S, nämlich in dem Teil vor M.
- Ist M dagegen kleiner als E, dann kann E nur in der zweiten Hälfte von S sein, nämlich in dem Teil nach M.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Die gesamte Sequenz `data` und daher auch das Segment S sind sortiert.
- Wenn M größer als E ist, dann kann E nur in der ersten Hälfte von S, nämlich in dem Teil vor M.
- Ist M dagegen kleiner als E, dann kann E nur in der zweiten Hälfte von S sein, nämlich in dem Teil nach M.
- Andernfalls, also falls $E = M$, dann haben wir E gefunden.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wenn M größer als E ist, dann kann E nur in der ersten Hälfte von S, nämlich in dem Teil vor M.
- Ist M dagegen kleiner als E, dann kann E nur in der zweiten Hälfte von S sein, nämlich in dem Teil nach M.
- Andernfalls, also falls $E = M$, dann haben wir E gefunden.
- Wenn wir E noch nicht gefunden haben aber die ausgewählte Hälfte von S leer ist (vielleicht hatte S ja nur ein oder zwei Elemente) … dann ist E nicht in S und darum auch nicht in `data`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Ist M dagegen kleiner als E, dann kann E nur in der zweiten Hälfte von S sein, nämlich in dem Teil nach M.
- Andernfalls, also falls $E = M$, dann haben wir E gefunden.
- Wenn wir E noch nicht gefunden haben aber die ausgewählte Hälfte von S leer ist (vielleicht hatte S ja nur ein oder zwei Elemente) … dann ist E nicht in S und darum auch nicht in `data`.
- Wenn $n =$ `len(data)`, dann können wir höchstens $\log_2 n$ Mal unsere Sequenz halbieren. Also ist die Zeitkomplexität von binärer Suche in $\mathcal{O}(\log n)$[7,38,48].

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wenn wir E noch nicht gefunden haben aber die ausgewählte Hälfte von S leer ist (vielleicht hatte S ja nur ein oder zwei Elemente) ... dann ist E nicht in S und darum auch nicht in `data`.
- Wenn $n =$ `len(data)`, dann können wir höchstens $\log_2 n$ Mal unsere Sequenz halbieren. Also ist die Zeitkomplexität von binärer Suche in $\mathcal{O}(\log n)$[7,38,48].
- In unserem Beispielprogramm wollen wir die Indizes von ein paar Zeichen in einer alphabetisch sortierten Sequenz `data = "abdfjlmoqsuvwyz"` von Buchstaben finden.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wenn $n =$ `len(data)`, dann können wir höchstens $\log_2 n$ Mal unsere Sequenz halbieren. Also ist die Zeitkomplexität von binärer Suche in $\mathcal{O}(\log n)$[7,38,48].
- In unserem Beispielprogramm wollen wir die Indizes von ein paar Zeichen in einer alphabetisch sortierten Sequenz `data = "abdfjlmoqsuvwyz"` von Buchstaben finden.
- Wir könnten dazu natürlich die String-Methode `find` zum Suchen der Zeichen verwenden.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wenn $n =$ `len(data)`, dann können wir höchstens $\log_2 n$ Mal unsere Sequenz halbieren. Also ist die Zeitkomplexität von binärer Suche in $\mathcal{O}(\log n)$[7,38,48].
- In unserem Beispielprogramm wollen wir die Indizes von ein paar Zeichen in einer alphabetisch sortierten Sequenz `data = "abdfjlmoqsuvwyz"` von Buchstaben finden.
- Wir könnten dazu natürlich die String-Methode `find` zum Suchen der Zeichen verwenden.
- Diese Methode durchsucht Strings aber linear.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- In unserem Beispielprogramm wollen wir die Indizes von ein paar Zeichen in einer alphabetisch sortierten Sequenz `data = "abdfjlmoqsuvwyz"` von Buchstaben finden.
- Wir könnten dazu natürlich die String-Methode `find` zum Suchen der Zeichen verwenden.
- Diese Methode durchsucht Strings aber linear.
- Wir wollen den Umstand, dass die Zeichen alphabetisch sortiert sind, ausnutzen und implementieren daher binäre Suche.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir könnten dazu natürlich die String-Methode `find` zum Suchen der Zeichen verwenden.
- Diese Methode durchsucht Strings aber linear.
- Wir wollen den Umstand, dass die Zeichen alphabetisch sortiert sind, ausnutzen und implementieren daher binäre Suche.
- Wir suchen die Zeichen `"a"`, `"c"`, `"o"`, `"p"`, `"w"`, und `"z"`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Diese Methode durchsucht Strings aber linear.
- Wir wollen den Umstand, dass die Zeichen alphabetisch sortiert sind, ausnutzen und implementieren daher binäre Suche.
- Wir suchen die Zeichen `"a"`, `"c"`, `"o"`, `"p"`, `"w"`, und `"z"`.
- Vier davon sind in `data`, aber `"c"` und `"p"` sind nicht.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir wollen den Umstand, dass die Zeichen alphabetisch sortiert sind, ausnutzen und implementieren daher binäre Suche.
- Wir suchen die Zeichen `"a"`, `"c"`, `"o"`, `"p"`, `"w"`, und `"z"`.
- Vier davon sind in `data`, aber `"c"` und `"p"` sind nicht.
- Wir suchen sie trotzdem.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir suchen die Zeichen `"a"`, `"c"`, `"o"`, `"p"`, `"w"`, und `"z"`.
- Vier davon sind in `data`, aber `"c"` und `"p"` sind nicht.
- Wir suchen sie trotzdem.
- Wir lassen eine Varriable `search` über die Liste `["a", "c", "o", "p", "w", "z"]` in der äußeren Schleife iterieren.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Vier davon sind in `data`, aber `"c"` und `"p"` sind nicht.
- Wir suchen sie trotzdem.
- Wir lassen eine Varriable `search` über die Liste `["a", "c", "o", "p", "w", "z"]` in der äußeren Schleife iterieren.
- In der inneren Schleife implementieren wir die binäre Suche.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir suchen sie trotzdem.
- Wir lassen eine Varriable `search` über die Liste `["a", "c", "o", "p", "w", "z"]` in der äußeren Schleife iterieren.
- In der inneren Schleife implementieren wir die binäre Suche.
- Die Suche benutzt zwei Indizes, `lower` und `upper`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir lassen eine Varriable `search` über die Liste `["a", "c", "o", "p", "w", "z"]` in der äußeren Schleife iterieren.
- In der inneren Schleife implementieren wir die binäre Suche.
- Die Suche benutzt zwei Indizes, `lower` und `upper`.
- `lower` ist das *inklusive* untere Ende des Segments S in dem `search` beinhaltet seien könnte.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- In der inneren Schleife implementieren wir die binäre Suche.
- Die Suche benutzt zwei Indizes, `lower` und `upper`.
- `lower` ist das *inklusive* untere Ende des Segments S in dem `search` beinhaltet seien könnte.
- Es wird daher mit `0` initialisiert.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Die Suche benutzt zwei Indizes, `lower` und `upper`.
- `lower` ist das *inklusive* untere Ende des Segments S in dem `search` beinhaltet seien könnte.
- Es wird daher mit `0` initialisiert.
- `upper` ist das *exklusive* obere Ende des Segments S in dem `search` beinhaltet seien könnte.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- `lower` ist das *inklusive* untere Ende des Segments S in dem `search` beinhaltet seien könnte.
- Es wird daher mit `0` initialisiert.
- `upper` ist das *exklusive* obere Ende des Segments S in dem `search` beinhaltet seien könnte.
- Wir initialisieren es mit `len(data)`: Da es *exklusiv* ist, ist es 1 größer als der größte gültige Index `len(data)- 1`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Es wird daher mit `0` initialisiert.
- `upper` ist das *exklusive* obere Ende des Segments S in dem `search` beinhaltet seien könnte.
- Wir initialisieren es mit `len(data)`: Da es *exklusiv* ist, ist es 1 größer als der größte gültige Index `len(data)- 1`.
- Unser Segment S ist nicht leer, beinhaltet also mindestens ein Element, so lange wie `lower < upper` gilt.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- `upper` ist das *exklusive* obere Ende des Segments S in dem `search` beinhaltet seien könnte.
- Wir initialisieren es mit `len(data)`: Da es *exklusiv* ist, ist es 1 größer als der größte gültige Index `len(data)- 1`.
- Unser Segment S ist nicht leer, beinhaltet also mindestens ein Element, so lange wie `lower < upper` gilt.
- Das ist also unsere Schleifenbedingung für die innere Schleife.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir initialisieren es mit `len(data)`: Da es *exklusiv* ist, ist es 1 größer als der größte gültige Index `len(data)- 1`.
- Unser Segment S ist nicht leer, beinhaltet also mindestens ein Element, so lange wie `lower < upper` gilt.
- Das ist also unsere Schleifenbedingung für die innere Schleife.
- In der binäre-Suche-Schleife berechnen wir erstmal den Index `mid` des mittleren Elements als `mid = (lower + upper)// 2`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Das ist also unsere Schleifenbedingung für die innere Schleife.
- In der binäre-Suche-Schleife berechnen wir erstmal den Index `mid` des mittleren Elements als `mid = (lower + upper)// 2`.
- Warnung: Interessanterweise ist das so nur korrekt in Python 3 mit seinen beliebig großen Ganzzahlen. In Sprachen wir C or Java, wo Ganzzahlen feste Grenzen haben, müssen wir stattdessen `mid = lower + (upper - lower)// 2` rechnen[38].

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- In der binäre-Suche-Schleife berechnen wir erstmal den Index `mid` des mittleren Elements als `mid = (lower + upper)// 2`.
- Warnung: Interessanterweise ist das so nur korrekt in Python 3 mit seinen beliebig großen Ganzzahlen. In Sprachen wir C or Java, wo Ganzzahlen feste Grenzen haben, müssen wir stattdessen `mid = lower + (upper - lower)//2` rechnen[38].
- Egal. Wir bekommen den Wert `mid_str` als das eine Zeichen an dem Index `mid` via `mid_str = data[mid]`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Warnung: Interessanterweise ist das so nur korrekt in Python 3 mit seinen beliebig großen Ganzzahlen. In Sprachen wir C or Java, wo Ganzzahlen feste Grenzen haben, müssen wir stattdessen `mid = lower + (upper - lower)//2` rechnen[38].
- Egal. Wir bekommen den Wert `mid_str` als das eine Zeichen an dem Index `mid` via `mid_str = data[mid]`.
- Wir wissen, wenn `mid_str < search`, dann kann unser Zeichen `search` an keiner Stelle in 0..`mid` liegen.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Egal. Wir bekommen den Wert `mid_str` als das eine Zeichen an dem Index `mid` via `mid_str = data[mid]`.
- Wir wissen, wenn `mid_str < search`, dann kann unser Zeichen `search` an keiner Stelle in 0..`mid` liegen.
- In diesem Fall können wir den inklusiven unteren Index `lower` auf `mid + 1` setzen.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Egal. Wir bekommen den Wert `mid_str` als das eine Zeichen an dem Index `mid` via `mid_str = data[mid]`.
- Wir wissen, wenn `mid_str < search`, dann kann unser Zeichen `search` an keiner Stelle in 0..`mid` liegen.
- In diesem Fall können wir den inklusiven unteren Index `lower` auf `mid + 1` setzen.
- Andernfalls, wenn `mid_str > search`, dann wissen wir, dass `search` unmöglich irgendwo an den Indizes in `mid`..`len(data)- 1` liegen kann.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir wissen, wenn `mid_str < search`, dann kann unser Zeichen `search` an keiner Stelle in 0..`mid` liegen.
- In diesem Fall können wir den inklusiven unteren Index `lower` auf `mid + 1` setzen.
- Andernfalls, wenn `mid_str > search`, dann wissen wir, dass `search` unmöglich irgendwo an den Indizes in `mid`..`len(data)- 1` liegen kann.
- Wir können also den *exklusiven* oberen Index `upper` auf `mid` setzen, was alle Elemente ab Index `mid` ausschließt.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- In diesem Fall können wir den inklusiven unteren Index `lower` auf `mid + 1` setzen.
- Andernfalls, wenn `mid_str > search`, dann wissen wir, dass `search` unmöglich irgendwo an den Indizes in `mid`..`len(data)- 1` liegen kann.
- Wir können also den *exklusiven* oberen Index `upper` auf `mid` setzen, was alle Elemente ab Index `mid` ausschließt.
- Wenn nun weder `mid_str < search` noch `mid_str > search` gilt, dann muss `mid_str == search` gelten.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Andernfalls, wenn `mid_str > search`, dann wissen wir, dass `search` unmöglich irgendwo an den Indizes in `mid`..`len(data)- 1` liegen kann.
- Wir können also den *exklusiven* oberen Index `upper` auf `mid` setzen, was alle Elemente ab Index `mid` ausschließt.
- Wenn nun weder `mid_str < search` noch `mid_str > search` gilt, dann muss `mid_str == search` gelten.
- Dann haben wir `search` gefunden – es liegt an Index `mid`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir können also den *exklusiven* oberen Index `upper` auf `mid` setzen, was alle Elemente ab Index `mid` ausschließt.
- Wenn nun weder `mid_str < search` noch `mid_str > search` gilt, dann muss `mid_str == search` gelten.
- Dann haben wir `search` gefunden – es liegt an Index `mid`.
- Wir geben dieses Ergebnis aus (wobei die `!r` im f-String Anführungszeichen um unsere Strings machen).

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wenn nun weder `mid_str < search` noch `mid_str > search` gilt, dann muss `mid_str == search` gelten.
- Dann haben wir `search` gefunden – es liegt an Index `mid`.
- Wir geben dieses Ergebnis aus (wobei die `!r` im f-String Anführungszeichen um unsere Strings machen).
- Nach dem Ausgeben dieser Information verlassen wir die `while`-Schleife mit `break`.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Dann haben wir `search` gefunden – es liegt an Index `mid`.
- Wir geben dieses Ergebnis aus (wobei die `!r` im f-String Anführungszeichen um unsere Strings machen).
- Nach dem Ausgeben dieser Information verlassen wir die `while`-Schleife mit `break`.
- Was pasiert nun, wenn wir `search` nicht finden, weil es nicht in `data` drin ist?

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Wir geben dieses Ergebnis aus (wobei die `!r` im f-String Anführungszeichen um unsere Strings machen).
- Nach dem Ausgeben dieser Information verlassen wir die `while`-Schleife mit `break`.
- Was pasiert nun, wenn wir `search` nicht finden, weil es nicht in `data` drin ist?
- Dann werden wir nichts ausgeben und auch nie `break` machen.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Nach dem Ausgeben dieser Information verlassen wir die `while`-Schleife mit `break`.
- Was pasiert nun, wenn wir `search` nicht finden, weil es nicht in `data` drin ist?
- Dann werden wir nichts ausgeben und auch nie `break` machen.
- In jeder Iterattion wird aber entweder `lower` größer oder `upper` kleiner.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Was pasiert nun, wenn wir `search` nicht finden, weil es nicht in `data` drin ist?
- Dann werden wir nichts ausgeben und auch nie `break` machen.
- In jeder Iterattion wird aber entweder `lower` größer oder `upper` kleiner.
- Irgendwann muss also `lower < upper` `False` werden.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Dann werden wir nichts ausgeben und auch nie `break` machen.
- In jeder Iterattion wird aber entweder `lower` größer oder `upper` kleiner.
- Irgendwann muss also `lower < upper` `False` werden.
- Die Schleife beendet sich also normal, weil ihre Schleifenbedingung `False` wird.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)        # *Exclusive* upper index.
    lower: int = 0                # Lowest possible index = 0 (inclusive).
    while lower < upper:          # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:      # If mid_str < search, then clearly...
            lower = mid + 1       # ...the index of search must be < mid.
        elif mid_str > search:    # If mid_str > search, then clearly...
            upper = mid           # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- In jeder Iterattion wird aber entweder `lower` größer oder `upper` kleiner.
- Irgendwann muss also `lower < upper` `False` werden.
- Die Schleife beendet sich also normal, weil ihre Schleifenbedingung `False` wird.
- Dann und nur dann wird der Körper des `else`-Statement am Ende der Schleife ausgeführt.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Irgendwann muss also `lower < upper` `False` werden.
- Die Schleife beendet sich also normal, weil ihre Schleifenbedingung `False` wird.
- Dann und nur dann wird der Körper des `else`-Statement am Ende der Schleife ausgeführt.
- Dann und nur dann geben wir aus, dass wir den String `search` nicht finden konnten.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ python3 while_loop_search.py ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Die Schleife beendet sich also normal, weil ihre Schleifenbedingung `False` wird.
- Dann und nur dann wird der Körper des `else`-Statement am Ende der Schleife ausgeführt.
- Dann und nur dann geben wir aus, dass wir den String `search` nicht finden konnten.
- Unsere binäre Suche funktioniert.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Beispiel: Binäre Suche

- Dann und nur dann wird der Körper des `else`-Statement am Ende der Schleife ausgeführt.
- Dann und nur dann geben wir aus, dass wir den String `search` nicht finden konnten.
- Unsere binäre Suche funktioniert.
- Wow. Wir können schon ziemlich coole Sachen implementieren.

```python
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)       # *Exclusive* upper index.
    lower: int = 0               # Lowest possible index = 0 (inclusive).
    while lower < upper:         # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:     # If mid_str < search, then clearly...
            lower = mid + 1      # ...the index of search must be < mid.
        elif mid_str > search:   # If mid_str > search, then clearly...
            upper = mid          # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

Zusammenfassung

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.
- Wir können `else` am Ende benutzen, um eine Aktion auszuführen, wenn die Schleife normal terminiert hat.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.
- Wir können `else` am Ende benutzen, um eine Aktion auszuführen, wenn die Schleife normal terminiert hat.
- Wir können jetzt richtig coole Sachen machen.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.
- Wir können `else` am Ende benutzen, um eine Aktion auszuführen, wenn die Schleife normal terminiert hat.
- Wir können jetzt richtig coole Sachen machen:
  - Wir können LIU Hui (刘徽) seine Methode zum Annähern von π beliebig oft in einer Schleife ausführen.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.
- Wir können `else` am Ende benutzen, um eine Aktion auszuführen, wenn die Schleife normal terminiert hat.
- Wir können jetzt richtig coole Sachen machen:
  - Wir können LIU Hui (刘徽) seine Methode zum Annähern von π beliebig oft in einer Schleife ausführen.
  - Wir können Wurzeln mit dem Algorithmus von Heron berechnen.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.
- Wir können `else` am Ende benutzen, um eine Aktion auszuführen, wenn die Schleife normal terminiert hat.
- Wir können jetzt richtig coole Sachen machen:
  - Wir können LIU Hui (刘徽) seine Methode zum Annähern von π beliebig oft in einer Schleife ausführen.
  - Wir können Wurzeln mit dem Algorithmus von Heron berechnen.
  - Wir können binäre Suche implementieren.

Zusammenfassung

- Wir beherrschen nun zwei Arten von Schleifen, die `for`- und die `while`-Schleife.
- Wir können mit `continue` zur nächsten Iteration springen.
- Wir können die Schleifen mit `break` beenden.
- Wir können `else` am Ende benutzen, um eine Aktion auszuführen, wenn die Schleife normal terminiert hat.
- Wir können jetzt richtig coole Sachen machen:
  - Wir können LIU Hui (刘徽) seine Methode zum Annähern von π beliebig oft in einer Schleife ausführen.
  - Wir können Wurzeln mit dem Algorithmus von Heron berechnen.
  - Wir können binäre Suche implementieren.
- Das ist doch schon sehr nett.

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. 145, 157).

[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. 145, 157).

[3] Paul Gustav Heinrich Bachmann. *Die Analytische Zahlentheorie / Dargestellt von Paul Bachmann*. Bd. Zweiter Theil der Reihe Zahlentheorie: Versuch einer Gesamtdarstellung dieser Wissenschaft in ihren Haupttheilen. Leipzig, Sachsen, Germany: B. G. Teubner, 1894. ISBN: 978-1-4181-6963-3. URL: `http://gallica.bnf.fr/ark:/12148/bpt6k994750` (besucht am 2023-12-13) (siehe S. 160).

[4] Christopher Barker. *A Function for Testing Approximate Equality*. Python Enhancement Proposal (PEP) 485. Beaverton, OR, USA: Python Software Foundation (PSF), 20. Jan. 2015. URL: `https://peps.python.org/pep-0485` (besucht am 2024-09-02) (siehe S. 52–67).

[5] 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. 157, 159).

[6] 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. 157).

[7] Jon Bentley. *Programming Pearls*. 2. Aufl. Reading, MA, USA: Addison-Wesley Professional, 7. Okt. 1999. ISBN: 978-0-201-65788-3 (siehe S. 87–100).

[8] 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. 159).

[9] 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. 156).

[10] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: 978-0-321-35668-0 (siehe S. 157).

[11] 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. 158).

References II

[12] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 157).

[13] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 156).

[14] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 157).

[15] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 157, 158).

[16] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, Juli 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (siehe S. 156).

[17] 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. 158).

[18] Philip Chrysopoulos. "The Ancient Greek Who Invented the World's First Steam Turbine". In: *Greek Reporter*. Cheyenne, WY, USA: Greekreporter COM LLC, 13. Dez. 2023. URL: https://greekreporter.com/2023/12/13/ancient-greek-world-first-steam-turbine (besucht am 2025-09-02) (siehe S. 6).

[19] 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. 159).

[20] 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. 158).

[21] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 158).

References III

[22] 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. 158).

[23] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 158).

[24] 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. 159).

[25] 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. 156).

[26] 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. 157, 158).

[27] "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. 157).

[28] 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[29]. (Siehe S. 147).

[29] 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[28] for a translation. (Siehe S. 147, 159).

[30] 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. 158).

References IV

[31] 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. 159).

[32] David Fowler und Eleanor Robson. “Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context”. *Historia Mathematica* 25(4):366–378, Nov. 1998. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0315-0860. doi:10.1006/hmat.1998.2209. URL: https://www.ux1.eiu.edu/~cfcid/Classes/4900/Class%20Notes/Babylonian%20Approximations.pdf (besucht am 2024-09-25). Article NO. HM982209 (siehe S. 5–12).

[33] “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. 157).

[34] 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. 157).

[35] David Goldberg. “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. *ACM Computing Surveys (CSUR)* 23(1):5–48, März 1991. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/103162.103163. URL: https://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/goldberg-floating-point.pdf (besucht am 2025-09-03) (siehe S. 52–56, 150).

[36] 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. 157).

[37] 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. 158).

References V

[38] Mohammad Hammoud. “Binary Search”. In: *15-122: Principles of Imperative Computation*. Doha, Qatar: Carnegie Mellon University Qatar, Frühling 2024. Kap. Lecture 06. URL: https://web2.qatar.cmu.edu/~mhhammou/15122-s24/lectures/06-binsearch/slides (besucht am 2024-09-26) (siehe S. 87–117).

[39] 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. 158).

[40] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O’Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 157).

[41] 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. 157, 159).

[42] 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. 158).

[43] *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. 158).

[44] *Information Technology – Universal Coded Character Set (UCS)*. International Standard ISO/IEC 10646:2020. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Dez. 2020 (siehe S. 159).

[45] 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. 159).

[46] Donald Ervin Knuth. “Big Omicron and Big Omega and Big Theta”. *ACM SIGACT News* 8(2):18–24, Apr.–Juni 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5700. doi:10.1145/1008328.1008329 (siehe S. 160).

References VI

[47] Donald Ervin Knuth. *Fundamental Algorithms*. 3. Aufl. Bd. 1 der Reihe The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1997. ISBN: 978-0-201-89683-1 (siehe S. 160).

[48] Donald Ervin Knuth. *Sorting and Searching*. Bd. 3 der Reihe The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1998. ISBN: 978-0-201-89685-5 (siehe S. 87–100).

[49] Olga Kosheleva. "Babylonian Method of Computing the Square Root: Justifications based on Fuzzy Techniques and on Computational Complexity". In: *Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS'2009)*. 14.–19. Juni 2009, Cincinnati, OH, USA. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2009. ISBN: 978-1-4244-4575-2. doi:10.1109/NAFIPS.2009.5156463. URL: https://www.cs.utep.edu/vladik/2009/olg09-05a.pdf (besucht am 2024-09-25) (siehe S. 5–18).

[50] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 158).

[51] Vincent Lafage. *Revisiting "What Every Computer Scientist Should Know About Floating-Point Arithmetic"*. arXiv.org: Computing Research Repository (CoRR) abs/2012.02492. Ithaca, NY, USA: Cornell Universiy Library, 4. Dez. 2020. doi:10.48550/arXiv.2012.02492. URL: https://arxiv.org/abs/2012.02492 (besucht am 2025-09-03). arXiv:2012.02492v1 [math.NA] 4 Dec 2020, see also[35] (siehe S. 52–60).

[52] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig, Sachsen, Germany: B. G. Teubner, 1909. ISBN: 978-0-8218-2650-8 (siehe S. 160).

[53] 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. 158).

[54] 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. 156).

[55] Marc Loy, Patrick Niemeyer und Daniel Leuck. *Learning Java*. 5. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2020. ISBN: 978-1-4920-5627-0 (siehe S. 157).

References VII

[56] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 158).

[57] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 157).

[58] "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. 159).

[59] 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. 157).

[60] 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. 158).

[61] "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. 20–29).

[62] 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. 156).

[63] 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. 159).

[64] 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. 158).

[65] 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. 156).

[66] *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).

References VIII

[67] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 158).

[68] *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. 156).

[69] 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. 156).

[70] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 157).

[71] 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. 158).

[72] 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. 156).

[73] Larkin Ridgway Scott. "Numerical Algorithms". In: *Numerical Analysis*. Princeton, NJ, USA: Princeton University Press, 18. Apr. 2011. Kap. 1. ISBN: 978-1-4008-3896-7. doi:10.1515/9781400838967-002. URL: https://assets.press.princeton.edu/chapters/s9487.pdf (besucht am 2024-09-25) (siehe S. 5–18).

[74] Syamal K. Sen und Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Kap. 5.5, S. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (siehe S. 156).

[75] 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. 157).

References IX

[76] 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. 157).

[77] 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. 158).

[78] "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. 158).

[79] 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. 153, 158).

[80] 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[79] (siehe S. 158).

[81] "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. 157).

[82] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 158).

[83] 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. 158).

[84] The Editors of Encyclopaedia Britannica, Hrsg. *Encyclopaedia Britannica*. Chicago, IL, USA: Encyclopædia Britannica, Inc.

[85] *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).

References X

[86] *The Unicode Standard, Version 15.1: Archived Code Charts*. South San Francisco, CA, USA: The Unicode Consortium, 25. Aug. 2023. URL: https://www.unicode.org/Public/15.1.0/charts/CodeCharts.pdf (besucht am 2024-07-26) (siehe S. 159).

[87] 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. 157).

[88] *Unicode®15.1.0*. South San Francisco, CA, USA: The Unicode Consortium, 12. Sep. 2023. ISBN: 978-1-936213-33-7. URL: https://www.unicode.org/versions/Unicode15.1.0 (besucht am 2024-07-26) (siehe S. 159).

[89] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 157).

[90] 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. 156–158).

[91] 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. 157, 158).

[92] *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. 158).

[93] 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. 158).

[94] 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. 156).

[95] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 156).

References XI

[96] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenberg und Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 26. Juli 1999–20. März 2024. URL: https://www.britannica.com/topic/chronology/Christian (besucht am 2025-08-27) (siehe S. 156).

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[13,62,95]. Learn more at `https://www.gnu.org/software/bash`.

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ[16]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 CE[74,96].

C is a programming language, which is very successful in system programming situations[25,68].

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ[16]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 before Common Era (BCE)[74,96].

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[9,54,65,69,72]. 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*[90].

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[94].

Glossary (in English) II

escape sequence Escaping is the process of presenting „forbidden“ characters or symbols in a sequence of characters or symbols. In Python[91], 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 („\“)[27]. 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[90], similar C-style escapes (starting with „\“) are supported[81].

f-string let you include the results of expressions in strings[14,33,34,36,59,76]. 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

Java is another very successful programming language, with roots in the C family of languages[10,55].

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[15,41].

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[5,40,75,87,89]. 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[1,2,6,26,57,70]. See https://mariadb.org for more information.

Microsoft Windows is a commercial proprietary operating system[12]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

Glossary (in English) III

MySQL An open source relational database management system[11,26,71,83,93]. 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[30,64,67,83]. See https://postgresql.org for more information.

psql is the client program used to access the PostgreSQL DBMS server.

Python The Python programming language[42,53,56,91], i.e., what you will learn about in our book[91]. Learn more at https://python.org.

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[20,37,39,77,82,90,92].

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[15] 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“[50].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[17,21–23,43,60,78–80,82]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[78].

(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"`.

Glossary (in English) IV

terminal A terminal is a text-based window where you can enter commands and execute them[5,19]. 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[19,41]. 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.

Unicode A standard for assigning characters to numbers[44,86,88]. The Unicode standard supports basically all characters from all languages that are currently in use, as well as many special symbols. It is the predominantly used way to represent characters in computers and is regularly updated and improved.

WWW World Wide Web[8,24]

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[31,45,63], 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`[58].

$i..j$ 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[29], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number[31,45], 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`[58].

Glossary (in English) V

$\boldsymbol{\Omega}(g(x))$ If $f(x) = \boldsymbol{\Omega}(g(x))$, then there exist positive numbers $x_{\mathbf{0}} \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $f(x) \geq c * g(x) \geq 0 \forall x \geq x_{\mathbf{0}}$[46,47]. In other words, $\boldsymbol{\Omega}(g(x))$ describes a lower bound for function growth.

$\mathcal{O}(g(x))$ If $f(x) = \mathcal{O}(g(x))$, then there exist positive numbers $x_{\mathbf{0}} \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $0 \leq f(x) \leq c * g(x) \forall x \geq x_{\mathbf{0}}$[3,46,47,52]. In other words, $\mathcal{O}(g(x))$ describes an upper bound for function growth.

$\boldsymbol{\Theta}(g(x))$ If $f(x) = \boldsymbol{\Theta}(g(x))$, then $f(x) = \mathcal{O}(g(x))$ and $f(x) = \boldsymbol{\Omega}(g(x))$[46,47]. In other words, $\boldsymbol{\Theta}(g(x))$ describes an exact order of function growth.

$\mathbb{R}$ the set of the real numbers.

$\mathbb{R}^+$ the set of the positive real numbers, i.e., $\mathbb{R}^+ = \{x \in \mathbb{R} : x > 0\}$.

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