



合肥大學  
HEFEI UNIVERSITY



# Programming with Python

## 11. Der Datentyp str

Thomas Weise (汤卫思)

[tweise@hfu.edu.cn](mailto:tweise@hfu.edu.cn)

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

应用优化研究所  
人工智能与大数据学院  
合肥大学  
中国安徽省合肥市

# 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-Code 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. Grundlegende Operationen
3. Die Funktion str und f-Strings
4. Strings in Andere Datentypen Konvertieren
5. Strings Escaping
6. Mehrzeilige Strings
7. Unicode
8. Zusammenfassung





# Einleitung





- Der vierte wichtige grundlegende Datentyp in Python sind Zeichenketten, Texte, auch genannt *Strings*.<sup>27</sup>



- Der vierte wichtige grundlegende Datentyp in Python sind Zeichenketten, Texte, auch genannt *Strings*.<sup>27</sup>
- Zeichenketten sind beliebig lange Sequenzen von Text-Zeichen.



- Der vierte wichtige grundlegende Datentyp in Python sind Zeichenketten, Texte, auch genannt *Strings*.<sup>27</sup>
- Zeichenketten sind beliebig lange Sequenzen von Text-Zeichen.
- In Python sind sie durch den Datentyp `str` repräsentiert.



- Der vierte wichtige grundlegende Datentyp in Python sind Zeichenketten, Texte, auch genannt *Strings*.<sup>27</sup>
- Zeichenketten sind beliebig lange Sequenzen von Text-Zeichen.
- In Python sind sie durch den Datentyp `str` repräsentiert.
- Wir haben sie bereits oftmals implizit oder explizit genutzt, z.B. in unserem ersten Programm, das einfach `"Hello World"` ausgedruckt hat.



- Der vierte wichtige grundlegende Datentyp in Python sind Zeichenketten, Texte, auch genannt *Strings*.<sup>27</sup>
- Zeichenketten sind beliebig lange Sequenzen von Text-Zeichen.
- In Python sind sie durch den Datentyp `str` repräsentiert.
- Wir haben sie bereits oftmals implizit oder explizit genutzt, z.B. in unserem ersten Programm, das einfach `"Hello World"` ausgedruckt hat.
- `"Hello World"` is so ein Text-String.



# Grundlegende Operationen



# Strings Definieren, Verketteten, und Indizieren

- Schauen wir uns die grundlegenden Operationen für Strings an.



# Strings Definieren, Verketteten, und Indizieren



- Wir öffnen ein Terminal (Unter Ubuntu Linux durch Drücken von `Ctrl` + `Alt` + `T`, unter Microsoft Windows durch Druck auf `Windows` + `R`, dann Schreiben von `cmd`, dann Druck auf `↵`.)

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$
```

# Strings Definieren, Verketteten, und Indizieren



- Wir schreiben `python3` und drücken `Enter`.

A terminal window with a dark background and light text. The title bar shows the user 'tweise@weise-laptop' and the current directory '~'. The terminal prompt is 'tweise@weise-laptop:~\$' and the command 'python3' has been entered, with a cursor at the end of the line. The window includes standard Linux window controls (minimize, maximize, close) and a search icon.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3
```

# Strings Definieren, Verketteten, und Indizieren



- Der Python-Interpreter startet.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

# Strings Definieren, Verketteten, und Indizieren



- Es gibt zwei grundlegende Methoden, Strings zu definieren.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"
```

# Strings Definieren, Verketteten, und Indizieren



- Es gibt zwei grundlegende Methoden, Strings zu definieren. Entweder mit einfachen oder doppelten Anführungszeichen.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"
```

# Strings Definieren, Verketteten, und Indizieren



- Es gibt zwei grundlegende Methoden, Strings zu definieren. Entweder mit einfachen oder doppelten Anführungszeichen. Probieren wir es zuerst mit den doppelten Anführungszeichen und schreiben `"Hello World!"`.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"
```

# Strings Definieren, Verketteten, und Indizieren



- Es gibt zwei grundlegende Methoden, Strings zu definieren. Probieren wir es zuerst mit den doppelten Anführungszeichen und schreiben `"Hello World!"`. Beachten Sie, dass die Anführungszeichen selbst nicht zum String gehören, sondern ihn nur begrenzen.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"
```

# Strings Definieren, Verketteten, und Indizieren



- Es gibt zwei grundlegende Methoden, Strings zu definieren. Probieren wir es zuerst mit den doppelten Anführungszeichen und schreiben `"Hello World!"`. Beachten Sie, dass die Anführungszeichen selbst nicht zum String gehören, sondern ihn nur begrenzen. Der Text wird uns wieder ausgegeben.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Nun probieren wir es mit einfachen Anführungszeichen und schreiben `'Hello World!'`.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'
```

# Strings Definieren, Verketteten, und Indizieren



- Nun probieren wir es mit einfachen Anführungszeichen und schreiben `'Hello World!'`. Und das wird uns auch wieder ausgegeben.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> □
```

# Strings Definieren, Verketteten, und Indizieren



- Es gibt zwei grundlegende Methoden, Strings zu definieren. Probieren wir es zuerst mit den doppelten Anführungszeichen und schreiben `"Hello World!"`. Beachten Sie, dass die Anführungszeichen selbst nicht zum String gehören, sondern ihn nur begrenzen. Der Text wird uns wieder ausgegeben.
- Nun probieren wir es mit einfachen Anführungszeichen und schreiben `'Hello World!'`. Und das wird uns auch wieder ausgegeben.

## Gute Praxis

Beim Definieren von String-Literalen sollte die Variante mit doppelten Anführungszeichen (`"..."`) bevorzugt werden. (Der *Style Guide for Python Code*<sup>32</sup> gibt keine Empfehlung, aber vielleicht für Konsistenz mit den *Docstring Conventions*<sup>13</sup>.)

# Strings Definieren, Verketteten, und Indizieren



- Strings können mit `+` aneinander angehängt (verkettet) werden.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"
```

# Strings Definieren, Verketteten, und Indizieren



- Strings können mit `+` aneinander angehängt (verkettet) werden. Sie ergeben dann einen einzigen String, der aus allen Teilstrings in der richtigen Reihenfolge besteht.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> □
```

# Strings Definieren, Verketteten, und Indizieren



- Die Funktion `len(x)` liefert uns die Länge des Strings `x`, also die Anzahl der Zeichen in der Zeichenkette.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")
```

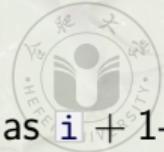
# Strings Definieren, Verketteten, und Indizieren



- Die Funktion `len(x)` liefert uns die Länge des Strings `x`, also die Anzahl der Zeichen in der Zeichenkette. "Hello" besteht aus fünf Zeichen.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> □
```

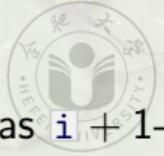
# Strings Definieren, Verketteten, und Indizieren



- Wir können auch einzelne Zeichen aus einem String herausholen. `x[i]` liefert das `i + 1`-te Zeichen.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch einzelne Zeichen aus einem String herausholen. `x[i]` liefert das `i + 1`-te Zeichen. Das erste Zeichen ist also an Index 0, bei "Hello" ist das "H". Dieses Zeichen ist natürlich wieder ein `str` (nur mit Länge 1).

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das zweite Zeichen bekommen wir durch `[1]`.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]
```

# Strings Definieren, Verketteten, und Indizieren



- Das zweite Zeichen bekommen wir durch `[1]`. Bei "Hello" ist das "e".

```
tweise@weise-laptop: ~  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das dritte Zeichen bekommen wir durch `[2]`.

```
tweise@weise-laptop: ~  
>>> "Hello World!"  
'Hello World!'  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]
```

# Strings Definieren, Verketteten, und Indizieren



- Das dritte Zeichen bekommen wir durch `[2]`. Bei "Hello" ist das "l".

```
tweise@weise-laptop: ~  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das vierte Zeichen bekommen wir durch `[3]`.

```
tweise@weise-laptop: ~  
>>> 'Hello World!'  
'Hello World!'  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]
```

# Strings Definieren, Verketteten, und Indizieren



- Das vierte Zeichen bekommen wir durch `[3]`. Bei "Hello" ist das "l".

```
tweise@weise-laptop: ~  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das fünfte Zeichen bekommen wir durch `[4]`.

```
tweise@weise-laptop: ~  
>>> "Hello" + ' ' + "World"  
'Hello World'  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]□
```

# Strings Definieren, Verketteten, und Indizieren



- Das fünfte Zeichen bekommen wir durch `[4]`. Bei "Hello" ist das "o".

```
tweise@weise-laptop: ~  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- "Hello" hat fünf Zeichen. Was passiert, wenn wir versuchen, auf das sechste Zeichen via `[5]` zuzugreifen?

```
tweise@weise-laptop: ~  
>>> len("Hello")  
5  
>>> "Hello"[0]  
'H'  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]
```

# Strings Definieren, Verketteten, und Indizieren



- "Hello" hat fünf Zeichen. Was passiert, wenn wir versuchen, auf das sechste Zeichen via `[5]` zuzugreifen? Dann gibt es eine Fehlermeldung. Das geht nämlich nicht.

```
tweise@weise-laptop: ~  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können einen String auch “von hinten” indizieren. `x[-1]` liefert das letzte Zeichen des Strings `x`.

```
tweise@weise-laptop: ~  
>>> "Hello"[1]  
'e'  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können einen String auch “von hinten” indizieren. `x[-1]` liefert das letzte Zeichen des Strings `x`. Bei “Hello” ist das “o”.

```
tweise@weise-laptop: ~  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das vorletzte Zeichen bekommen wir durch Index `[-2]`.

```
tweise@weise-laptop: ~  
>>> "Hello"[2]  
'l'  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]
```

# Strings Definieren, Verketteten, und Indizieren



- Das vorletzte Zeichen bekommen wir durch Index `[-2]`. Bei "Hello" ist das "l".

```
tweise@weise-laptop: ~  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das vor-vorletzte Zeichen bekommen wir durch Index `[-3]`.

```
tweise@weise-laptop: ~  
>>> "Hello"[3]  
'l'  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]
```

# Strings Definieren, Verketteten, und Indizieren



- Das vor-vorletzte Zeichen bekommen wir durch Index `[-3]`. Bei "Hello" ist das "l".

```
tweise@weise-laptop: ~  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das vierte Zeichen von hinten bekommen wir durch Index `[-4]`.

```
tweise@weise-laptop: ~  
>>> "Hello"[4]  
'o'  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]
```

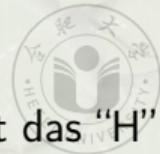
# Strings Definieren, Verketteten, und Indizieren



- Das vierte Zeichen von hinten bekommen wir durch Index `[-4]`. Bei "Hello" ist das "e".

```
tweise@weise-laptop: ~  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Das fünfte Zeichen von hinten bekommen wir durch Index `[-5]`. Bei "Hello" ist das "H".

```
tweise@weise-laptop: ~  
>>> "Hello"[5]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]
```

# Strings Definieren, Verketteten, und Indizieren



- Das fünfte Zeichen von hinten bekommen wir durch Index `[-5]`. Bei "Hello" ist das "H".

```
tweise@weise-laptop: ~  
File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- “Hello” hat nur fünf Zeichen. Was passiert, wenn wir versuchen, auf das sechste Zeichen von hinten zuzugreifen?

```
tweise@weise-laptop: ~  
File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[-1]  
'o'  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]
```

# Strings Definieren, Verketteten, und Indizieren



- "Hello" hat nur fünf Zeichen. Was passiert, wenn wir versuchen, auf das sechste Zeichen von hinten zuzugreifen? Dann gibt es wieder eine Fehlermeldung, weil das nämlich nicht geht.

```
tweise@weise-laptop: ~  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch ganze Substrings (Unterzeichenketten) extrahieren, indem wir den Index `i` des ersten Zeichens und den Index `j` nach dem letzten zu extrahierenden Zeichen angeben als `[i:j]`.

```
tweise@weise-laptop: ~  
>>> "Hello"[-2]  
'l'  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch ganze Substrings (Unterzeichenketten) extrahieren, indem wir den Index `i` des ersten Zeichens und den Index `j` nach dem letzten zu extrahierenden Zeichen angeben als `[i:j]`. `[0:3]` ergibt also die Zeichen an den Indizes 0, 1, 2. Bei "Hello" ist das `"Hel"`.

```
tweise@weise-laptop: ~  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- `"Hello"[1:3]` ist der Substring, der beim zweiten Zeichen anfängt (Index 1) und **vor** dem vierten Zeichen (Index 3) aufhört.

```
tweise@weise-laptop: ~  
>>> "Hello"[-3]  
'l'  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]
```

# Strings Definieren, Verketteten, und Indizieren



- `"Hello"[1:3]` ist der Substring, der beim zweiten Zeichen anfängt (Index 1) und **vor** dem vierten Zeichen (Index 3) aufhört. Also `"el"`.

```
tweise@weise-laptop: ~  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Lassen wir einfach den zweiten Index weg, dann werden alle Zeichen bis zum Ende des Strings zurückgegeben.

```
tweise@weise-laptop: ~  
>>> "Hello"[-4]  
'e'  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]
```

# Strings Definieren, Verketteten, und Indizieren



- Lassen wir einfach den zweiten Index weg, dann werden alle Zeichen bis zum Ende des Strings zurückgegeben. `"Hello"[2:]` ergibt also die Zeichen an den Indices 2, 3, 4, und 5, also `"llo"`.

```
tweise@weise-laptop: ~  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können genausogut auch negative Indizes verwenden, die dann wieder von hinten Zählen.

```
tweise@weise-laptop: ~  
>>> "Hello"[-5]  
'H'  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> "Hello"[1:-2]
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können genauso gut auch negative Indizes verwenden, die dann wieder von hinten Zählen. `"Hello"[1:-2]` started an Index 1 und hört vor dem zweiten Zeichen von hinten (Index 3) auf. Die Indizes im Originalstring gehen von 0 bis 5, wir haben also die Zeichen an den Indizes von 1 bis 2, demnach also `"el"`.

```
tweise@weise-laptop: ~  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> "Hello"[1:-2]  
'el'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch den ersten Index weglassen, dann fängt der zurückgelieferte String am Anfang der Zeichenkette an und geht bis **vor** den zweiten Index.

```
tweise@weise-laptop: ~  
>>> "Hello"[-6]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> "Hello"[1:-2]  
'el'  
>>> "Hello"[:-2]
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch den ersten Index weglassen, dann fängt der zurückgelieferte String am Anfang der Zeichenkette an und geht bis **vor** den zweiten Index. `"Hello"[:-2]` fängt am Anfang an und hört **vor** dem vorletzten Zeichen auf, ist also `"Hel"`.

```
tweise@weise-laptop: ~  
File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> "Hello"[1:-2]  
'el'  
>>> "Hello"[:-2]  
'Hel'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch drei Werte angeben, `[i:j:k]`, wobei `i` wieder der Index des ersten Zeichens und `j` der Index **nach** dem letzten Zeichen ist. `k` ist die Schrittweite. `"Hello World!"[1:8:2]` liefert **jedes zweite** Zeichen beginnend an Index 1 und endet **vor** Index 8, also die Zeichen an den Indices 1, 3, 5, und 7. Das sind `"e l o"`.

```
tweise@weise-laptop: ~  
File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> "Hello"[1:-2]  
'el'  
>>> "Hello"[:-2]  
'Hel'  
>>> "Hello World!"[1:8:2]
```

# Strings Definieren, Verketteten, und Indizieren



- Wir können auch drei Werte angeben, `[i:j:k]`, wobei `i` wieder der Index des ersten Zeichens und `j` der Index **nach** dem letzten Zeichen ist. `k` ist die Schrittweite. `"Hello World!"[1:8:2]` liefert **jedes zweite** Zeichen beginnend an Index 1 und endet **vor** Index 8, also die Zeichen an den Indices 1, 3, 5, und 7. Das sind `"e l o"`.

```
tweise@weise-laptop: ~  
>>> "Hello"[0:3]  
'Hel'  
>>> "Hello"[1:3]  
'el'  
>>> "Hello"[2:]  
'llo'  
>>> "Hello"[1:-2]  
'el'  
>>> "Hello"[:-2]  
'Hel'  
>>> "Hello World!"[1:8:2]  
'e l o'  
>>> 
```

# Strings Definieren, Verketteten, und Indizieren



- Schauen wir uns die grundlegenden Operationen für Strings an.
- Damit haben wir also erstmal ein Grundverständnis, was Strings sind, wie wir sie verketteten, ihre Länge bestimmen, und wieder auseinanderfummeln können.

# Weitere Grundlegende String Operationen



- Schauen wir uns die ein paar weitere grundlegenden Operationen für Strings an.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

# Weitere Grundlegende String Operationen



- Mit dem Operator `a in b` prüfen wir, ob die Zeichenkette `a` irgendwo im String `b` enthalten ist.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"
```

# Weitere Grundlegende String Operationen



- Mit dem Operator `a in b` prüfen wir, ob die Zeichenkette `a` irgendwo im String `b` enthalten ist. Das ist hier der Fall: `"World"` ist tatsächlich im String `"Hello World!"` enthalten.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> 
```

# Weitere Grundlegende String Operationen



- Ist "Earth" irgendwo in "Hello World!" enthalten?

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"
```

# Weitere Grundlegende String Operationen



- Ist "Earth" irgendwo in "Hello World!" enthalten? Nein.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> 
```

# Weitere Grundlegende String Operationen



- Die Funktion `a.find(b)` sucht den Index, an dem die Zeichenkette `b` in `a` beginnt.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")
```

# Weitere Grundlegende String Operationen



- Die Funktion `a.find(b)` sucht den Index, an dem die Zeichenkette `b` in `a` beginnt.  
`"World"` beginnt an Index 6 in `"Hello World!"`.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> □
```

# Weitere Grundlegende String Operationen



- String-Funktionen und Vergleiche sind "case-sensitive": Groß- und Kleinbuchstaben werden als unterschiedlich betrachtet.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")
```

# Weitere Grundlegende String Operationen



- String-Funktionen und Vergleiche sind "case-sensitive": Groß- und Kleinbuchstaben werden als unterschiedlich betrachtet. Somit gilt `"W" != "w"`.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")
```

## Weitere Grundlegende String Operationen



- String-Funktionen und Vergleiche sind "case-sensitive": Groß- und Kleinbuchstaben werden als unterschiedlich betrachtet. Somit gilt `"W" != "w"`. Somit kann `"world"` nicht in `"Hello World!"` gefunden werden.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> □
```

## Weitere Grundlegende String Operationen



- Groß- und Kleinbuchstaben werden als unterschiedlich betrachtet. Somit gilt `"W" != "w"`. Somit kann `"world"` nicht in `"Hello World!"` gefunden werden. Somit liefert die Function `-1` zurück.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> 
```

## Weitere Grundlegende String Operationen



- Groß- und Kleinbuchstaben werden als unterschiedlich betrachtet. Somit kann `"world"` nicht in `"Hello World!"` gefunden werden. Somit liefert die Function `-1` zurück. Beachte also: Niemals das Ergebnis von `find` direkt zum Indizieren nehmen, denn `-1` steht für "letztes Zeichen"...

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> 
```

# Weitere Grundlegende String Operationen



- Wo befindet sich `"l"` in `"Hello World!"`?

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")
```

## Weitere Grundlegende String Operationen



- Wo befindet sich "l" in "Hello World! "? An Index 2.

```
tweise@weise-laptop: ~  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> □
```

## Weitere Grundlegende String Operationen



- Wir können auch den Index angeben, ab dem gesucht werden soll: Wo befindet sich `"l"` in `"Hello World!"` wenn wir ab Index 3 suchen?

```
tweise@weise-laptop: ~  
>>> "World" in "Hello World!"  
True  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)
```

## Weitere Grundlegende String Operationen



- Wir können auch den Index angeben, ab dem gesucht werden soll: Wo befindet sich `"l"` in `"Hello World!"` wenn wir ab Index 3 suchen? An Index 3.

```
tweise@weise-laptop: ~  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> □
```

## Weitere Grundlegende String Operationen



- Wo befindet sich "l" in "Hello World!" wenn wir ab Index 4 suchen?

```
tweise@weise-laptop: ~  
>>> "Earth" in "Hello World!"  
False  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)
```

## Weitere Grundlegende String Operationen



- Wo befindet sich "l" in "Hello World!" wenn wir ab Index 4 suchen? An Index 9.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> □
```

## Weitere Grundlegende String Operationen



- Wo befindet sich "l" in "Hello World!" wenn wir ab Index 10 suchen?

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("World")  
6  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)
```

## Weitere Grundlegende String Operationen



- Wo befindet sich "l" in "Hello World!" wenn wir ab Index 10 suchen? Dann finden wir kein "l" mehr und -1 wird zurückgegeben.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> 
```

# Weitere Grundlegende String Operationen



- `rfind` sucht von hinten/rechts nach vorne.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("world")  
-1  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")
```

## Weitere Grundlegende String Operationen



- `rfind` sucht von hinten/rechts nach vorne. Von rechts aus gehen findet sich das erste Auftreten von `"l"` in `"Hello World!"` an Index 9.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> □
```

## Weitere Grundlegende String Operationen



- Sowohl bei `find` als auch bei `rfind` können wir keinen Index, den Startindex, oder den Start- und den (exklusiven) End-Index für die Suche angeben. Wir suchen nun in der Zeichenkette von Index 2 bis **vor** Index 9 von rechts.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l")  
2  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)
```

## Weitere Grundlegende String Operationen



- Sowohl bei `find` als auch bei `rfind` können wir keinen Index, den Startindex, oder den Start- und den (exklusiven) End-Index für die Suche angeben. Wir suchen nun in der Zeichenkette von Index 2 bis `vor` Index 9 von rechts. Und finden `"l"` an Index 3.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> □
```

## Weitere Grundlegende String Operationen



- Wir suchen nun in der Zeichenkette von Index 0 bis **vor** Index 3 von rechts.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l", 3)  
3  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)
```

## Weitere Grundlegende String Operationen



- Wir suchen nun in der Zeichenkette von Index 0 bis vor Index 3 von rechts. Und finden "l" an Index 2.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> □
```

## Weitere Grundlegende String Operationen



- Wir suchen nun in der Zeichenkette von Index 0 bis **vor** Index 2 von rechts.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l", 4)  
9  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)
```

## Weitere Grundlegende String Operationen



- Wir suchen nun in der Zeichenkette von Index 0 bis **vor** Index 2 von rechts. Und finden **"1"** gar nicht mehr, bekommen also **-1** zurück.

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.replace(b, c)` ersetzt alle Auftreten von `b` in `a` mit `c` und gibt das Ergebnis als neuen String zurück

```
tweise@weise-laptop: ~  
>>> "Hello World!".find("l", 10)  
-1  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")
```

## Weitere Grundlegende String Operationen



- `a.replace(b, c)` ersetzt alle Auftreten von `b` in `a` mit `c` und gibt das Ergebnis als neuen String zurück Ersetzen wir alle `"Hello"` in `"Hello World!"` mit `"Hi"`, so bekommen wir `"Hi World!"`.

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> □
```

# Weitere Grundlegende String Operationen



- Ersetzen wir alle "Hello" in "Hello World! Hello!" mit "Hi"...

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l")  
9  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")
```

## Weitere Grundlegende String Operationen



- Ersetzen wir alle "Hello" in "Hello World! Hello!" mit "Hi", so bekommen wir "Hi World! Hi!".

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> 
```

# Weitere Grundlegende String Operationen



- Ersetzen wir alle "Hello" in "Hello World!" mit "Hello! Hello!"...

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l", 2, 9)  
3  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")
```

## Weitere Grundlegende String Operationen



- Ersetzen wir alle "Hello" in "Hello World!" mit "Hello! Hello!", so bekommen wir "Hello! Hello! World!". Das Ersetzen funktioniert also nicht rekursiv, ersetzt also nicht in bereits ersetzten Strings.

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.strip()` entfernt alle so-genannten “whitespace”-Zeichen (Leerzeichen, Newlines, Tabs) am Anfang und Ende eines Strings und gibt das Ergebnis als neuen String zurück

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l", 0, 3)  
2  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()
```

## Weitere Grundlegende String Operationen



- `a.strip()` entfernt alle so-genannten “whitespace”-Zeichen (Leerzeichen, Newlines, Tabs) am Anfang und Ende eines Strings und gibt das Ergebnis als neuen String zurück  
`" Hello World! "` wird so zu `"Hello World!"`.

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.lstrip()` entfernt alle so-geannten “whitespace”-Zeichen (Leerzeichen, Newlines, Tabs) am Anfang eines Strings und gibt das Ergebnis als neuen String zurück

```
tweise@weise-laptop: ~  
>>> "Hello World!".rfind("l", 0, 2)  
-1  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".lstrip()
```

## Weitere Grundlegende String Operationen



- `a.lstrip()` entfernt alle so-geannten “whitespace”-Zeichen (Leerzeichen, Newlines, Tabs) am Anfang eines Strings und gibt das Ergebnis als neuen String zurück  
`" Hello World! "` wird so zu `"Hello World! "`.

```
tweise@weise-laptop: ~  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".lstrip()  
'Hello World! '  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.rstrip()` entfernt alle so-geannten “whitespace”-Zeichen (Leerzeichen, Newlines, Tabs) am Ende eines Strings und gibt das Ergebnis als neuen String zurück

```
tweise@weise-laptop: ~  
>>> "Hello World!".replace("Hello", "Hi")  
'Hi World!'  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".rstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()
```

## Weitere Grundlegende String Operationen



- `a.rstrip()` entfernt alle so-genannten “whitespace”-Zeichen (Leerzeichen, Newlines, Tabs) am Ende eines Strings und gibt das Ergebnis als neuen String zurück  
`" Hello World! "` wird so zu `" Hello World!"`.

```
tweise@weise-laptop: ~  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".rstrip()  
'Hello World! '  
>>> " Hello World! ".lstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.lower()` wandelt alle Großbuchstaben in `a` in Kleinbuchstaben um und gibt das Ergebnis als neuen String zurück.

```
tweise@weise-laptop: ~  
>>> "Hello World! Hello!".replace("Hello", "Hi")  
'Hi World! Hi!'  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".rstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()
```

## Weitere Grundlegende String Operationen



- `a.lower()` wandelt alle Großbuchstaben in `a` in Kleinbuchstaben um und gibt das Ergebnis als neuen String zurück. `"Hello World!"` wird so zu `"hello world!"`.

```
tweise@weise-laptop: ~  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".lstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.upper()` wandelt alle Kleinbuchstaben in `a` in Großbuchstaben um und gibt das Ergebnis als neuen String zurück.

```
tweise@weise-laptop: ~  
>>> "Hello World!".replace("Hello", "Hello! Hello!")  
'Hello! Hello! World!'  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".rstrip()  
'Hello World! '  
>>> " Hello World! ".lstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
^
```

## Weitere Grundlegende String Operationen



- `a.upper()` wandelt alle Kleinbuchstaben in `a` in Großbuchstaben um und gibt das Ergebnis als neuen String zurück. `"Hello World!"` wird so zu `"HELLO WORLD"`.

```
tweise@weise-laptop: ~  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".lstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> 
```

## Weitere Grundlegende String Operationen



- `a.startswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` anfängt.

```
tweise@weise-laptop: ~  
>>> " Hello World! ".strip()  
'Hello World!'  
>>> " Hello World! ".rstrip()  
'Hello World! '  
>>> " Hello World! ".lstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")
```

## Weitere Grundlegende String Operationen



- `a.startswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` anfängt. Und weil es case-sensitive ist, ergibt das hier `False`.

```
tweise@weise-laptop: ~  
>>> " Hello World! ".lstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.startswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` anfängt.

```
tweise@weise-laptop: ~  
>>> " Hello World! ".lstrip()  
'Hello World! '  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> "Hello World!".startswith("Hello")
```

## Weitere Grundlegende String Operationen



- `a.startswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` anfängt. Aber jetzt stimmt es.

```
tweise@weise-laptop: ~  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> "Hello World!".startswith("Hello")  
True  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.endswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` endet.

```
tweise@weise-laptop: ~  
>>> " Hello World! ".rstrip()  
' Hello World!'  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> "Hello World!".startswith("Hello")  
True  
>>> "Hello World!".endswith("Hello")
```

## Weitere Grundlegende String Operationen



- `a.endswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` endet. Und das ist hier natürlich nicht der Fall.

```
tweise@weise-laptop: ~  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> "Hello World!".startswith("Hello")  
True  
>>> "Hello World!".endswith("Hello")  
False  
>>> □
```

## Weitere Grundlegende String Operationen



- `a.endswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` endet.

```
tweise@weise-laptop: ~  
>>> "Hello World!".lower()  
'hello world!'  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> "Hello World!".startswith("Hello")  
True  
>>> "Hello World!".endswith("Hello")  
False  
>>> "Hello World!".endswith("World!")
```

## Weitere Grundlegende String Operationen



- `a.endswith(b)` gibt `True` zurück wenn und nur wenn `a` mit `b` endet. Aber das stimmt.

```
tweise@weise-laptop: ~  
>>> "Hello World!".upper()  
'HELLO WORLD!'  
>>> "Hello World!".startswith("hello")  
False  
>>> "Hello World!".startswith("Hello")  
True  
>>> "Hello World!".endswith("Hello")  
False  
>>> "Hello World!".endswith("World!")  
True  
>>> □
```

## Weitere Grundlegende String Operationen



- Schauen wir uns die ein paar weitere grundlegenden Operationen für Strings an.
- So, nun haben wir schon ziemlich viele String-Funktionen gelernt.



# Die Funktion str und f-Strings



## Die Funktion str

- Für Werte `a` der meisten Datentypen liefert die Funktion `str(a)` einen entsprechenden String zurück.





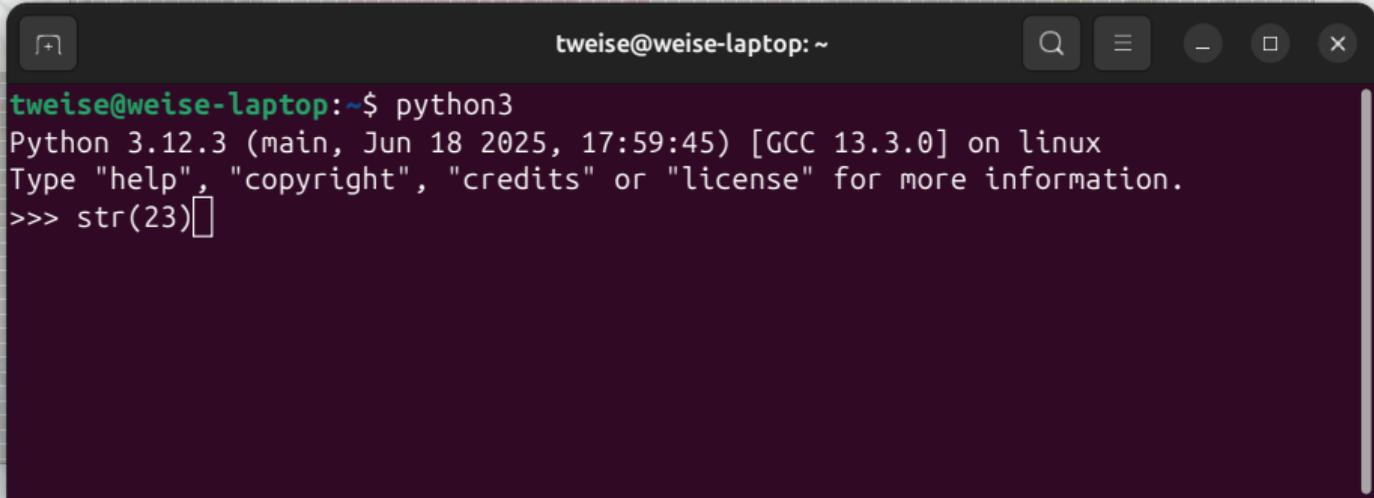
# Die Funktion str

- Für Werte `a` der meisten Datentypen liefert die Funktion `str(a)` einen entsprechenden String zurück.
- Probieren wir das mal aus.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

# Die Funktion str

- Wenden wir die Funktion `str` auf den `int`-Wert `23` an...

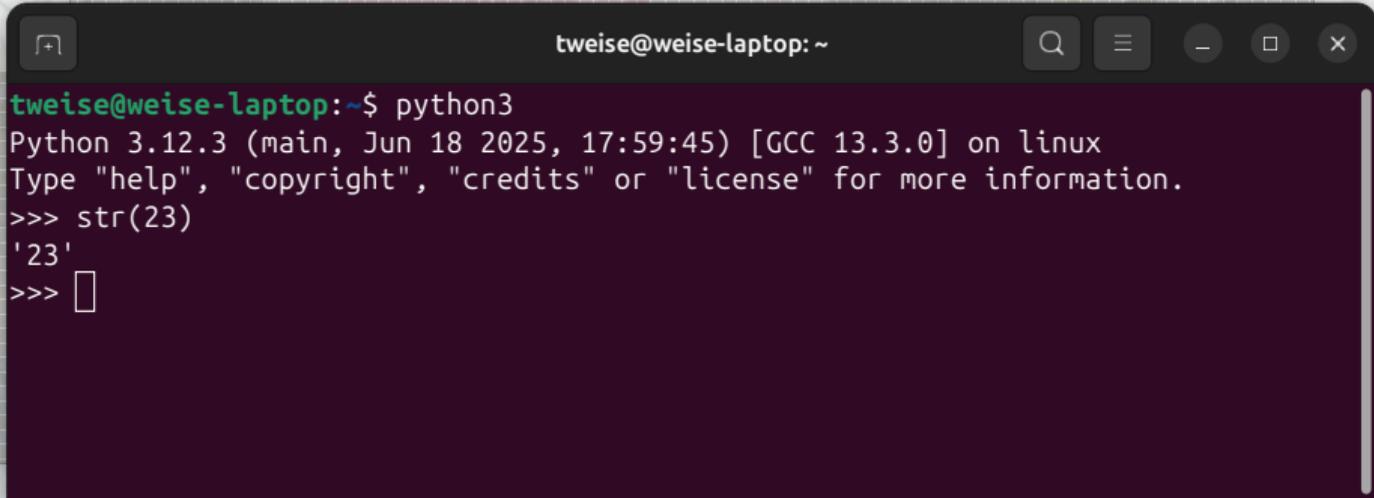


```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)
```



# Die Funktion str

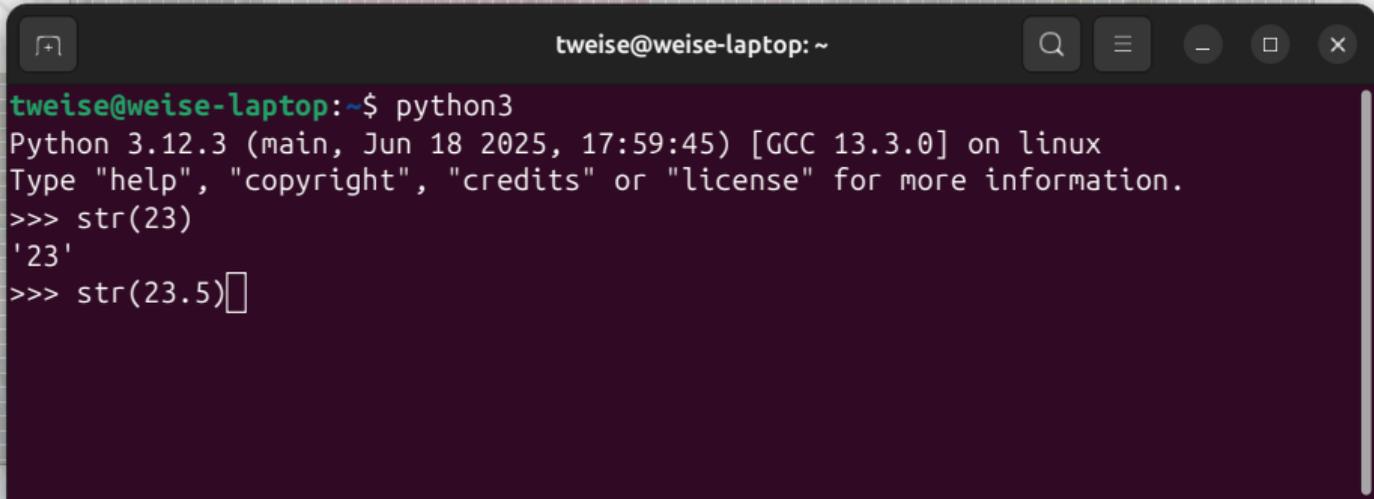
- Wenden wir die Funktion `str` auf den `int`-Wert `23` an so liefert sie den `str`-Wert `"23"`.



```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> 
```

# Die Funktion str

- Wenden wir die Funktion `str` auf den `float`-Wert `23.5` an...



```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> str(23.5)
```





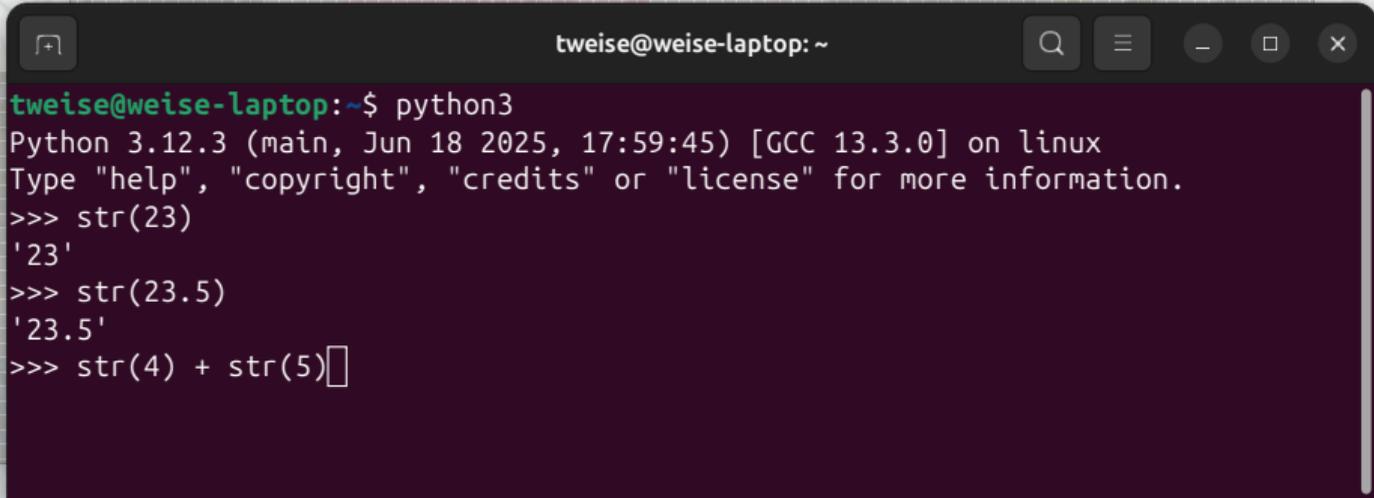
# Die Funktion str

- Wenden wir die Funktion `str` auf den `float`-Wert `23.5` an so liefert sie den `str`-Wert `"23.5"`.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> str(23.5)  
'23.5'  
>>> 
```

# Die Funktion str

- Wenn der Ausdruck `str(4) + str(5)` ausgerechnet wird...

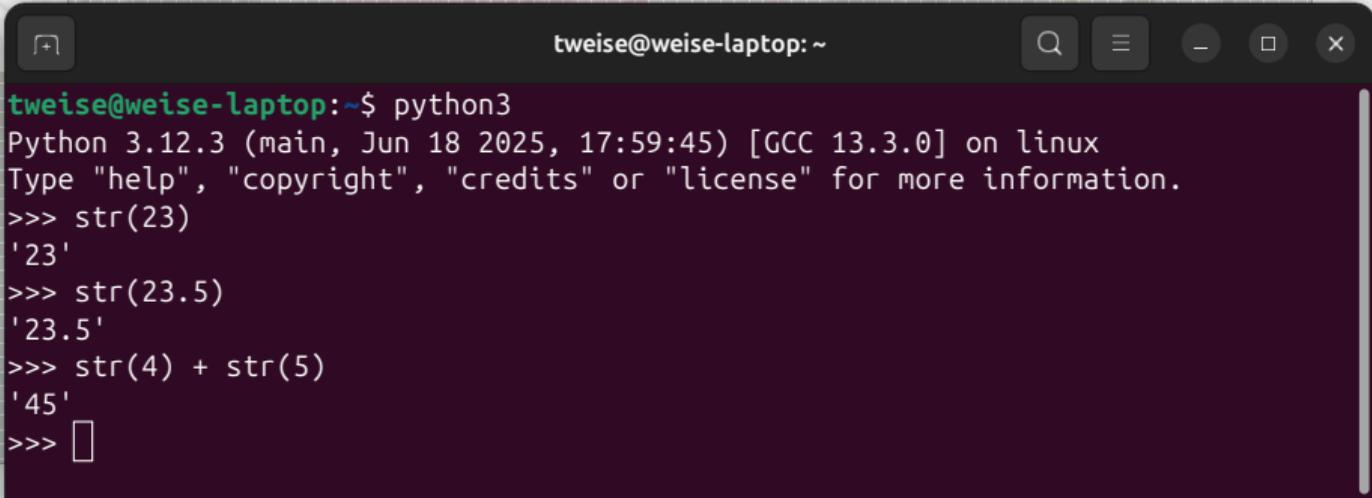


```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> str(23.5)  
'23.5'  
>>> str(4) + str(5)
```



# Die Funktion str

- Wenn der Ausdruck `str(4) + str(5)` ausgerechnet wird, dann werden zuerst die beiden `str`-Funktionen ausgerechnet, was dann `"4" + "5"` ergibt. Dies ergibt dann wiederum `"45"`.



```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> str(23.5)  
'23.5'  
>>> str(4) + str(5)  
'45'  
>>> □
```

# Die Funktion str

- Wir können nahezu beliebige Aufrufe von `str(...)` mit anderen Strings und `+` kombinieren.



```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> str(23.5)  
'23.5'  
>>> str(4) + str(5)  
'45'  
>>> str(1 < 5) + " is True and " + str(1 + 5) + " = 6."□
```

# Die Funktion str

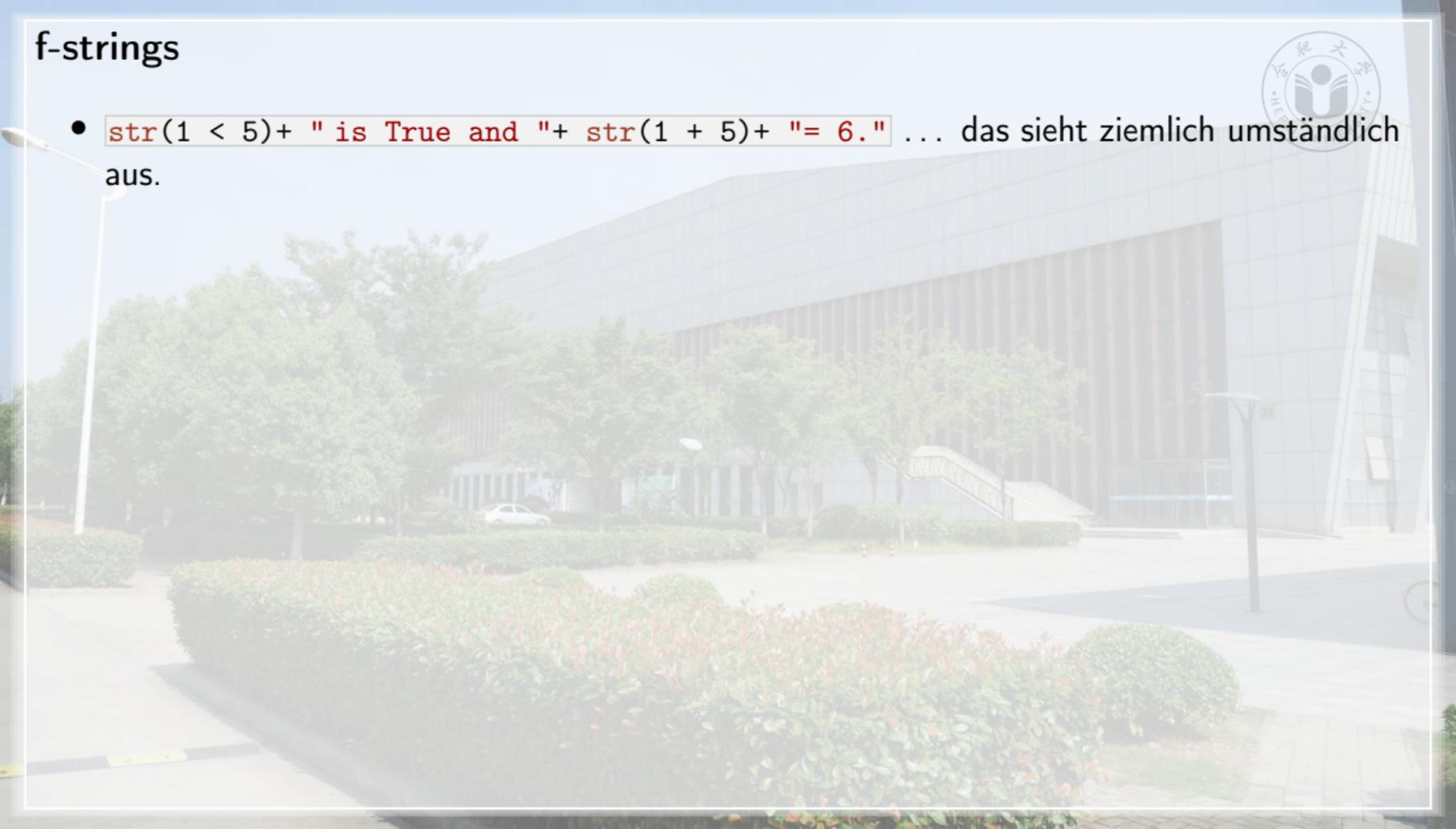


- Wir können nahezu beliebige Aufrufe von `str(...)` mit anderen Strings und `+` kombinieren. Beispielsweise ist `str(1 < 5)` das selbe wie `str(True)`, was `"True"` ergibt. `str(1 + 5)` ist das selbe wie `str(6)`, was dann `"6"` ergibt. Alle Strings zusammen ergeben dann...

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> str(23)  
'23'  
>>> str(23.5)  
'23.5'  
>>> str(4) + str(5)  
'45'  
>>> str(1 < 5) + " is True and " + str(1 + 5) + " = 6."  
'True is True and 6 = 6.'  
>>> 
```

# f-strings

- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.



# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können.

# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.

# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.

# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax.

# f-strings



- `str(1 < 5)+ " is True and "+ str(1 + 5)+ "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).

# f-strings



- `str(1 < 5)+ " is True and "+ str(1 + 5)+ "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).
  2. Sie können beliebige Ausdrücke enthalten, die jeweils mit `{...}` begrenzt sind.

# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).
  2. Sie können beliebige Ausdrücke enthalten, die jeweils mit `{...}` begrenzt sind.
  3. Zusätzlich können Markierungen für die Formatierung der Ausdruck-Ergebnisse angegeben werden.

# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).
  2. Sie können beliebige Ausdrücke enthalten, die jeweils mit `{...}` begrenzt sind.
  3. Zusätzlich können Markierungen für die Formatierung der Ausdruck-Ergebnisse angegeben werden.
  4. Während der sogenannten String-Interpolation, welche automatisch passiert, werden diese Ausdrücke dann ausgerechnet und ihre Ergebnisse an ihrer Stelle in den Text eingefügt.

# f-strings



- `str(1 < 5) + " is True and " + str(1 + 5) + "= 6."` ... das sieht ziemlich umständlich aus.
- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).
  2. Sie können beliebige Ausdrücke enthalten, die jeweils mit `{...}` begrenzt sind.
  3. Zusätzlich können Markierungen für die Formatierung der Ausdruck-Ergebnisse angegeben werden.
  4. Während der sogenannten String-Interpolation, welche automatisch passiert, werden diese Ausdrücke dann ausgerechnet und ihre Ergebnisse an ihrer Stelle in den Text eingefügt.
- So wird aus `f"a{6-1}b"` dann `"a5b"`.

# f-strings

- Probieren wir das mal aus.



```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

# f-strings

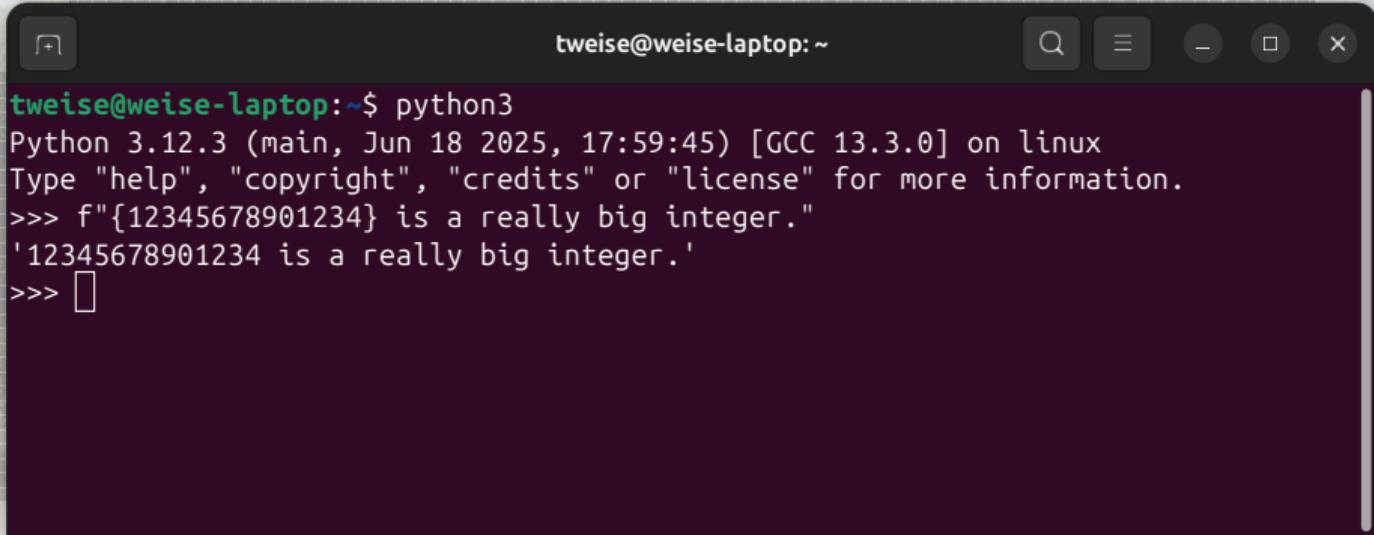


- Eine `int`-Zahl ist ein Ausdruck. Schreiben wir also mal einen f-String, der eine Zahl in einen Text umwandelt und mit anderem Text kombiniert.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."□
```

# f-strings

- Eine `int`-Zahl ist ein Ausdruck. Schreiben wir also mal einen f-String, der eine Zahl in einen Text umwandelt und mit anderem Text kombiniert. Das funktioniert problemlos.



```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> □
```

# f-strings



- f-Strings erlauben es uns auch, die Ergebnisse von Ausdrücken zu formatieren. Schreiben wir z.B. `:` hinten an den Ausdruck (hier: der Zahl) ran, dann wird der Ausdruck als Zahl behandelt und es werden Tausender-Trennzeichen (hier: Kommas) bei der Umwandlung in Text eingefügt.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,}."[]
```

# f-strings



- f-Strings erlauben es uns auch, die Ergebnisse von Ausdrücken zu formatieren. Schreiben wir z.B. `:`, hinten an den Ausdruck (hier: der Zahl) ran, dann wird der Ausdruck als Zahl behandelt und es werden Tausender-Trennzeichen (hier: Kommas) bei der Umwandlung in Text eingefügt. Aus der Zahl `12345678901234` wird der Text `12,345,678,901,234`.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,}."  
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> □
```

# f-strings

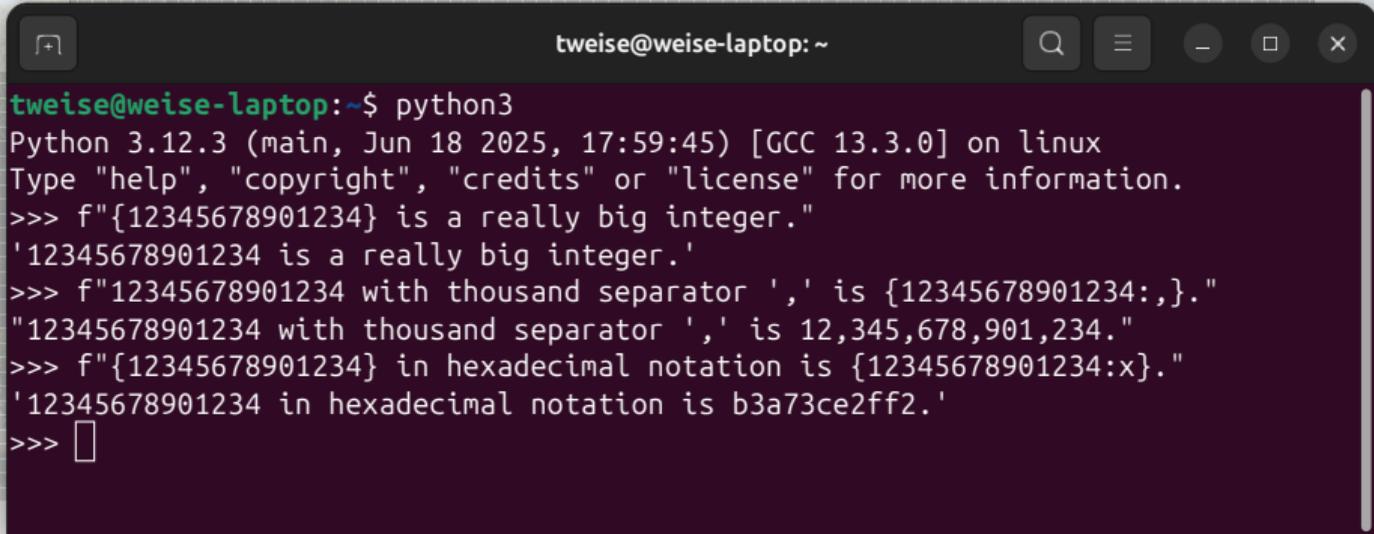


- Wir haben in den Slides zum Datentyp `int` verschiedene Zahlensysteme erwähnt. Fügen wir `:x` an den Ausdruck an, so wird sein Ergebnis als Hexadezimalzahl dargestellt.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,.}"  
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."□
```

# f-strings

- Wir haben in den Slides zum Datentyp `int` verschiedene Zahlensysteme erwähnt. Fügen wir `:x` an den Ausdruck an, so wird sein Ergebnis als Hexadezimalzahl dargestellt. So wird aus der (Dezimal-)Zahl `12345678901234` die Hexadezimalzahl `b3a73ce2ff2`.



```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:},)."   
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> □
```

# f-strings



- Wollen wir zusätzlich noch das `0x`-Präfix mit davor haben, schreiben wir `:#x` anstatt von `:x`.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:},)."   
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}." □
```

# f-strings



- Wollen wir zusätzlich noch das `0x`-Präfix mit davor haben, schreiben wir `:#x` anstatt von `:x`. So wird aus der (Dezimal-)Zahl `12345678901234` die Hexadezimalzahl `0xb3a73ce2ff2`.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,.}"  
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> 
```

# f-strings



- Fügen wir `:b` an den Ausdruck an, so wird sein Ergebnis als Binärzahl dargestellt.

```
tweise@weise-laptop: ~  
twaise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,.}."  
'12345678901234 with thousand separator ',' is 12,345,678,901,234.'  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."□
```

# f-strings

- Fügen wir `:b` an den Ausdruck an, so wird sein Ergebnis als Binärzahl dargestellt. So wird aus der (Dezimal-)Zahl `1234567890` die Binärzahl `1001001100101100000001011010010`.

```
tweise@weise-laptop: ~  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,.}."  
'12345678901234 with thousand separator ',' is 12,345,678,901,234.'  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> 
```

# f-strings



- Fügen wir `:#b` an den Ausdruck an, so wird sein Ergebnis als Binärzahl mit dem Präfix `0b` dargestellt.

```
tweise@weise-laptop: ~  
Type "help", "copyright", "credits" or "license" for more information.  
>>> f"{12345678901234} is a really big integer."  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:.,}."  
'12345678901234 with thousand separator ',' is 12,345,678,901,234.'  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."□
```

# f-strings



- Fügen wir `:#b` an den Ausdruck an, so wird sein Ergebnis als Binärzahl mit dem Präfix `0b` dargestellt. So wird aus der (Dezimal-)Zahl `1234567890` die Binärzahl `0b1001001100101100000001011010010`.

```
tweise@weise-laptop: ~  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:},."  
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> □
```

# f-strings



- Nun sind `int`-Zahlen zwar Ausdrücke, aber nicht besonders spannend. Probieren wir es diesmal mit einer echten kleinen Berechnung.

```
tweise@weise-laptop: ~  
'12345678901234 is a really big integer.'  
>>> f"12345678901234 with thousand separator ',' is {12345678901234:},)."   
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x})."   
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x})."   
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b})."   
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b})."   
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"
```

# f-strings



- Nun sind `int`-Zahlen zwar Ausdrücke, aber nicht besonders spannend. Probieren wir es diesmal mit einer echten kleinen Berechnung. `f"{5 + 4}"` wird tatsächlich zu `"9"`.

```
tweise@weise-laptop: ~  
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> □
```

# f-strings



- Wir können in f-Strings auf **alle** Werte zugreifen, auf die wir aktuell Zugriff haben. Importieren wir also interessehalber mal wieder  $\pi$  from `math`-Modul.

```
tweise@weise-laptop: ~  
"12345678901234 with thousand separator ',' is 12,345,678,901,234."  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi
```

# f-strings



- Wir können in f-Strings auf **alle** Werte zugreifen, auf die wir aktuell Zugriff haben. Importieren wir also interessehalber mal wieder  $\pi$  from `math`-Modul. Was natürlich klappt.

```
tweise@weise-laptop: ~  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> □
```

# f-strings



- Nun können wir auch `pi` in f-Strings verwenden.

```
tweise@weise-laptop: ~  
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."  
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}." 
```

# f-strings



- Nun können wir auch `pi` in f-Strings verwenden. Und sein Wert wird eingefügt.

```
tweise@weise-laptop: ~  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> □
```

# f-strings



- Wenn wir alle Stellen der `float`-Konstante  $\pi$  drucken, dann ist das ziemlich viel. Wollen wir nur `NNN` Nachkommastellen einer Zahl angezeigt bekommen, so fügen wir einfach `:.NNNf` an den Ausdruck an.

```
tweise@weise-laptop: ~  
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."  
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}." □
```

# f-strings



- Wenn wir alle Stellen der `float`-Konstante  $\pi$  drucken, dann ist das ziemlich viel. Wollen wir nur `NNN` Nachkommastellen einer Zahl angezeigt bekommen, so fügen wir einfach `:.NNnf` an den Ausdruck an. Das `:.2f` bewirkt, dass der `float`-Wert `pi` auf zwei Nachkommastellen gerundet wird.

```
tweise@weise-laptop: ~  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> □
```

# f-strings



- Wir können eine Zahl in Prozent konvertieren, in dem wir `:%` an den Ausdruck anhängen. Wollen wir den Prozentwert mit `NNN` Nachkommastellen, so hängen wir stattdessen `:.NNN%` an.

```
tweise@weise-laptop: ~  
>>> f"{1234567890} in binary notation is {1234567890:b}."  
'1234567890 in binary notation is 1001001100101100000001011010010.'  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}." □
```

# f-strings



- Wir können eine Zahl in Prozent konvertieren, in dem wir `:%` an den Ausdruck anhängen. Wollen wir den Prozentwert mit `NNN` Nachkommastellen, so hängen wir stattdessen `:.NNN%` an.  $\frac{1}{321} \approx 0.00311526$ , was  $0.311526\%$  entspricht. Die Ausgabe stimmt also.

```
tweise@weise-laptop: ~  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> □
```

# f-strings



- Wir können auch Tausender-Trennzeichen und Nachkommastellenangaben kombinieren.

```
tweise@weise-laptop: ~  
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."  
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."
```

# f-strings



- Wir können auch Tausender-Trennzeichen und Nachkommastellenangaben kombinieren. `1.2345533e4` entspricht  $1.2345533 * 10^4$ , was wiederum 12345.533 ist. Auf eine Nachkommastelle genau ergibt das 12345.5.

```
tweise@weise-laptop: ~  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> □
```

# f-strings



- Importieren wir die Sinus-Funktion aus dem Modul `math`.

```
tweise@weise-laptop: ~  
>>> f"{5} + {4} = {5 + 4}"  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin
```

# f-strings



- Importieren wir die Sinus-Funktion aus dem Modul `math`. Gar kein Problem.

```
tweise@weise-laptop: ~  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> □
```

# f-strings



- Und schon können wir sie in f-Strings verwenden, deren Ausgabe wir wieder beliebig formatieren können.

```
tweise@weise-laptop: ~  
'5 + 4 = 9'  
>>> from math import pi  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}." □
```

# f-strings



- Und schon können wir sie in f-Strings verwenden, deren Ausgabe wir wieder beliebig formatieren können.  $\sin \frac{\pi}{4} = \frac{\sqrt{2}}{2} \approx 0.7071067811865476$ .

```
tweise@weise-laptop: ~  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> □
```

# f-strings



- Das Format `:.e` druckt eine Zahl in der wissenschaftlichen Notation. `:.DDDg` benutzt ebenfalls die wissenschaftliche Notation, druckt jedoch nur `DDD` Ziffern (insgesamt, nicht Nachkommastellen).

```
tweise@weise-laptop: ~  
>>> f"pi is approximately {pi}."  
'pi is approximately 3.141592653589793.'  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."
```

# f-strings



- Das Format `:.e` druckt eine Zahl in der wissenschaftlichen Notation. `:.DDDg` benutzt ebenfalls die wissenschaftliche Notation, druckt jedoch nur `DDD` Ziffern (insgesamt, nicht Nachkommastellen). Aus `:.3g` folgert, dass nur drei Ziffern (`1.24`) gedruckt werden.

```
tweise@weise-laptop: ~  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> □
```

# f-strings



- Wenn die geschwungenen Klammern `{` und `}` Ausdrücke in den Zeichenketten begrenzen ... was machen wir dann, wenn wir mal eine geschwungene Klammer in einem String haben wollen?

```
tweise@weise-laptop: ~  
>>> f"pi rounded to two decimals is {pi:.2f}."  
'pi rounded to two decimals is 3.14.'  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."□
```

# f-strings



- Wenn die geschwungenen Klammern `{` und `}` Ausdrücke in den Zeichenketten begrenzen ... was machen wir dann, wenn wir mal eine geschwungene Klammer in einem String haben wollen? Einfach *zwei* geschwungene Klammern schreiben.

```
tweise@weise-laptop: ~  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> 
```

# f-strings



- Ein wirklich cooles Feature ist, dass wir sowohl einen Ausdruck `A` als auch dessen Ergebnis `A` ausdrucken können, in dem wir `f"{A = }"` schreiben.

```
tweise@weise-laptop: ~  
>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."  
'1/321 as percentage with 2 decimals is 0.31%.'  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> f"{5 + 4 = }"
```

# f-strings



- Ein wirklich cooles Feature ist, dass wir sowohl einen Ausdruck `A` als auch dessen Ergebnis `A` ausdrücken können, in dem wir `f"{A = }"` schreiben. Das wird dann in `"A = E"` umgewandelt.

```
tweise@weise-laptop: ~  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> f"{5 + 4 = }"  
'5 + 4 = 9'  
>>> □
```

# f-strings



- Das funktioniert auch mit komplizierten Ausdrücken.

```
tweise@weise-laptop: ~  
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1f}."  
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> f"{5 + 4 = }"  
'5 + 4 = 9'  
>>> f"{23 * sin(2 - 5) = }" 
```

# f-strings



- Das funktioniert auch mit komplizierten Ausdrücken. Problemlos.

```
tweise@weise-laptop: ~  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> f"{5 + 4 = }"  
'5 + 4 = 9'  
>>> f"{23 * sin(2 - 5) = }"  
'23 * sin(2 - 5) = -3.245760185376946'  
>>> □
```

# f-strings



- Kommen wir nun zu unserem Anfangsbeispiel, dem komplizierten String `str(1 < 5)+ " is True and "+ str(1 + 5)+ "= 6."`, zurück. Als f-String sieht das so aus: `f"{1 < 5}is True and {1 + 5}= 6."` Beide Varianten ergeben `"True is True and 6 = 6."`.

```
tweise@weise-laptop: ~  
>>> from math import sin  
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> f"{5 + 4 = }"  
'5 + 4 = 9'  
>>> f"{23 * sin(2 - 5) = }"  
'23 * sin(2 - 5) = -3.245760185376946'  
>>> f"{1 < 5} is True and {1 + 5} = 6."□
```

# f-strings



- Kommen wir nun zu unserem Anfangsbeispiel, dem komplizierten String `str(1 < 5)+ " is True and "+ str(1 + 5)+ "= 6."`, zurück. Als f-String sieht das so aus: `f"{1 < 5}is True and {1 + 5}= 6."` Beide Varianten ergeben `"True is True and 6 = 6."`.

```
tweise@weise-laptop: ~  
'sin(0.25pi) is approximately 0.70711.'  
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3g}."  
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'  
>>> f"Single braces without expression: {{ and }}."  
'Single braces without expression: { and }.'  
>>> f"{5 + 4 = }"  
'5 + 4 = 9'  
>>> f"{23 * sin(2 - 5) = }"  
'23 * sin(2 - 5) = -3.245760185376946'  
>>> f"{1 < 5} is True and {1 + 5} = 6."  
'True is True and 6 = 6.'  
>>> □
```

# f-strings



- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).
  2. Sie können beliebige Ausdrücke enthalten, die jeweils mit `{...}` begrenzt sind.
  3. Zusätzlich können Markierungen für die Formatierung der Ausdruck-Ergebnisse angegeben werden.
  4. Während der sogenannten String-Interpolation, welche automatisch passiert, werden diese Ausdrücke dann ausgerechnet und ihre Ergebnisse an ihrer Stelle in den Text eingefügt.
- Das waren natürlich nur einige Beispiele.

# f-strings



- In der Tat gibt es eine Möglichkeit, schön formatierte Zeichenketten zu erstellen, die Ergebnisse von Berechnungen beinhalten können: f-Strings.
- f-Strings erlauben es uns, die Ergebnisse von Ausdrücken in Zeichenketten einzufügen<sup>5,11,12,14,22,26</sup>.
- Sie haben eine einfache Syntax:
  1. Sie sind mit `f"..."` begrenzt (beachten Sie das kleine `f` vor dem einleitenden Anführungszeichen!).
  2. Sie können beliebige Ausdrücke enthalten, die jeweils mit `{...}` begrenzt sind.
  3. Zusätzlich können Markierungen für die Formatierung der Ausdruck-Ergebnisse angegeben werden.
  4. Während der sogenannten String-Interpolation, welche automatisch passiert, werden diese Ausdrücke dann ausgerechnet und ihre Ergebnisse an ihrer Stelle in den Text eingefügt.
- Das waren natürlich nur einige Beispiele.
- Mehr Informationen gibt es unter *Python f-Strings*<sup>5</sup>, "Formatted String Literals"<sup>11</sup>, und *Literal String Interpolation*<sup>26</sup>.



# Strings in Andere Datentypen Konvertieren



# Strings in Andere Datentypen Konvertieren

- Oft wollen wir Daten in unser Program einlesen oder eingeben.



# Strings in Andere Datentypen Konvertieren



- Oft wollen wir Daten in unser Program einlesen oder eingeben.
- Die Daten sind of als Text gespeichert oder werden als Text in ein Terminal eingegeben.

# Strings in Andere Datentypen Konvertieren



- Oft wollen wir Daten in unser Program einlesen oder eingeben.
- Die Daten sind oft als Text gespeichert oder werden als Text in ein Terminal eingegeben.
- Wir brauchen sie dann aber oft als andere Datentypen, z.B. als Zahlen.

# Strings in Andere Datentypen Konvertieren



- Oft wollen wir Daten in unser Program einlesen oder eingeben.
- Die Daten sind oft als Text gespeichert oder werden als Text in ein Terminal eingegeben.
- Wir brauchen sie dann aber oft als andere Datentypen, z.B. als Zahlen.
- Viele Datentypen in Python bieten eine Funktion an, die eine Zeichenkette in sie umwandelt.

# Strings in Andere Datentypen Konvertieren



- Oft wollen wir Daten in unser Program einlesen oder eingeben.
- Die Daten sind oft als Text gespeichert oder werden als Text in ein Terminal eingegeben.
- Wir brauchen sie dann aber oft als andere Datentypen, z.B. als Zahlen.
- Viele Datentypen in Python bieten eine Funktion an, die eine Zeichenkette in sie umwandelt.
- `int("4")` erzeugt z.B. den `int`-Wert `4`.

# Strings in Andere Datentypen Konvertieren



- Oft wollen wir Daten in unser Program einlesen oder eingeben.
- Die Daten sind oft als Text gespeichert oder werden als Text in ein Terminal eingegeben.
- Wir brauchen sie dann aber oft als andere Datentypen, z.B. als Zahlen.
- Viele Datentypen in Python bieten eine Funktion an, die eine Zeichenkette in sie umwandelt.
- `int("4")` erzeugt z.B. den `int`-Wert `4`.
- Schauen wir uns das mal an.

# Strings in Andere Datentypen Konvertieren



- Schauen wir uns das mal an.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `int(a)` wandelt einen Wert `a` eines anderen Datentypen in einen `int`-Wert um. `a` könnte z.B. ein String sein.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `int(a)` wandelt einen Wert `a` eines anderen Datentypen in einen `int`-Wert um. `a` könnte z.B. ein String sein. `int("1111")` ergibt daher den Integer `1111`.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion hat einen optionalen zweiten Parameter, der die Basis des zu benutzenden Zahlensystems angibt. Die bekannten Präfixe werden automatisch erkannt.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16) □
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion hat einen optionalen zweiten Parameter, der die Basis des zu benutzenden Zahlensystems angibt. Die bekannten Präfixe werden automatisch erkannt.

`int("0x1111", 16)` wandelt eine Hexdezimalzahl `0x1111` um, es gilt  $1 * 1 + 1 * 16 + 1 * 16^2 + 1 * 16^3 = 1 + 16 + 256 + 4096 = 4369$  und das ergibt daher den Integer `4369`.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> □
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion hat einen optionalen zweiten Parameter, der die Basis des zu benutzenden Zahlensystems angibt. Die bekannten Präfixe werden automatisch erkannt.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion hat einen optionalen zweiten Parameter, der die Basis des zu benutzenden Zahlensystems angibt. Die bekannten Präfixe werden automatisch erkannt.

`int("0b1111", 2)` wandelt eine Binärzahl `0b1111` um, es gilt

$1 * 1 + 1 * 2 + 1 * 2^2 + 1 * 2^3 = 1 + 2 + 4 + 8 = 15$  und das ergibt daher den Integer `15`.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> □
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `float` wandelt einen Wert `a` eines anderen Datentypen in einen `float`-Wert um. `a` könnte z.B. ein String sein.

```
tweise@weise-laptop: ~  
twiese@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `float` wandelt einen Wert `a` eines anderen Datentypen in einen `float`-Wert um. `a` könnte z.B. ein String sein. `float` erkennt die wissenschaftliche Notation und `float("2.233e4")` wird daher `22330.0`.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Natürlich geht das auch ohne wissenschaftliche Notation. `float("0.1123")` wird zu `0.1123`.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")
```

# Strings in Andere Datentypen Konvertieren



- Natürlich geht das auch ohne wissenschaftliche Notation. `float("0.1123")` wird zu `0.1123`.

```
tweise@weise-laptop: ~  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Es werden auch die Konstanten für “zu groß für `float`” und “keine Zahl” erkannt.

`float("inf")` wird zu `math.inf`.

```
tweise@weise-laptop: ~  
>>> int("1111")  
1111  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")
```

# Strings in Andere Datentypen Konvertieren



- Es werden auch die Konstanten für “zu groß für `float`” und “keine Zahl” erkannt.

`float("inf")` wird zu `math.inf`.

```
tweise@weise-laptop: ~  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Es werden auch die Konstanten für “zu groß für `float`” und “keine Zahl” erkannt.

```
tweise@weise-laptop: ~  
>>> int("0x1111", 16)  
4369  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> float("nan")
```

# Strings in Andere Datentypen Konvertieren



- Es werden auch die Konstanten für “zu groß für `float`” und “keine Zahl” erkannt. Und `float("nan")` wird zu `math.nan`.

```
tweise@weise-laptop: ~  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> float("nan")  
nan  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `bool` wandelt einen Wert `a` eines anderen Datentypen in einen `bool`-Wert um. `a` könnte z.B. ein String sein.

```
tweise@weise-laptop: ~  
>>> int("0b1111", 2)  
15  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> float("nan")  
nan  
>>> bool("True")
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `bool` wandelt einen Wert `a` eines anderen Datentypen in einen `bool`-Wert um. `a` könnte z.B. ein String sein. Aus `bool("True")` wird daher `True`.

```
tweise@weise-laptop: ~  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> float("nan")  
nan  
>>> bool("True")  
True  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `bool` wandelt einen Wert `a` eines anderen Datentypen in einen `bool`-Wert um. `a` könnte z.B. ein String sein.

```
tweise@weise-laptop: ~  
>>> float("2.233e4")  
22330.0  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> float("nan")  
nan  
>>> bool("True")  
True  
>>> bool("False")
```

# Strings in Andere Datentypen Konvertieren



- Die Funktion `bool` wandelt einen Wert `a` eines anderen Datentypen in einen `bool`-Wert um. `a` könnte z.B. ein String sein. Und `bool("False")` wird zu `False`.

```
tweise@weise-laptop: ~  
>>> float("0.1123")  
0.1123  
>>> float("inf")  
inf  
>>> float("nan")  
nan  
>>> bool("True")  
True  
>>> bool("False")  
True  
>>> 
```

# Strings in Andere Datentypen Konvertieren



- Oft wollen wir Daten in unser Program einlesen oder eingeben.
- Die Daten sind oft als Text gespeichert oder werden als Text in ein Terminal eingegeben.
- Wir brauchen sie dann aber oft als andere Datentypen, z.B. als Zahlen.
- Viele Datentypen in Python bieten eine Funktion an, die eine Zeichenkette in sie umwandelt.
- Das war relativ einfach.



## Strings Escaping



# Strings Escaping

- Zeichenketten werden durch Anführungszeichen begrenzt.



# Strings Escaping



- Zeichenketten werden durch Anführungszeichen begrenzt.
- Was machen wir, wenn wir Anführungszeichen **in** einer Zeichenkette haben wollen?

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

# Strings Escaping



- Wenn wir unsere Strings durch doppelte Anführungszeichen (") begrenzen, dann können problemlos einfache Anführungszeichen (') einfügen.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")
```

# Strings Escaping



- Wenn wir unsere Strings durch doppelte Anführungszeichen (") begrenzen, dann können problemlos einfache Anführungszeichen (') einfügen. Beachten Sie, dass wenn wir Strings mit `print` ausgeben, die begrenzenden Anführungszeichen nicht in der Ausgabe enthalten sind.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> 
```

# Strings Escaping



- Wenn wir doppelte Anführungszeichen (") brauchen, dann begrenzen wir unsere Strings eben mit einfachen Anführungszeichen (').

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: ")  
A single quotation mark: '  
>>> print('A double quotation mark: "')
```

# Strings Escaping



- Wenn wir doppelte Anführungszeichen (") brauchen, dann begrenzen wir unsere Strings eben mit einfachen Anführungszeichen ('). Beachten Sie, dass wenn wir Strings mit `print` ausgeben, die begrenzenden Anführungszeichen nicht in der Ausgabe enthalten sind.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: ")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> 
```

# Strings Escaping



- Aber was machen wir, wenn wir **beide** Arten von Anführungszeichen brauchen?

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: ")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> □
```

## Strings Escaping



- Aber was machen wir, wenn wir **beide** Arten von Anführungszeichen brauchen? Dann können wir nicht auf “andere” Anführungszeichen zur Begrenzung der Strings ausweichen.

## Strings Escaping



- Aber was machen wir, wenn wir **beide** Arten von Anführungszeichen brauchen? Dann können wir nicht auf “andere” Anführungszeichen zur Begrenzung der Strings ausweichen.
- Dann brauchen wir sogenannte Escape-Sequenzen<sup>7</sup>.

# Strings Escaping



- Wir können dem doppelten Anführungszeichen einfach in Backslash (“\”) voranstellen.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")
```

# Strings Escaping



- Wir können dem doppelten Anführungszeichen einfach in Backslash (“\”) voranstellen. Dann wird es nicht mehr als String-Begrenzer interpretiert und verwandelt sich in ein normales doppeltes Anführungszeichen.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> □
```

# Strings Escaping



- So können wir bequem doppelte Anführungszeichen in Strings haben, die mit doppelten Anführungszeichen begrenzt sind.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\\"Hello World!\")
```

# Strings Escaping



- So können wir bequem doppelte Anführungszeichen in Strings haben, die mit doppelten Anführungszeichen begrenzt sind. Die Backslashes verschwinden dann in der Ausgabe.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\\"Hello World!\\"")  
"Hello World!"  
>>> 
```

# Strings Escaping



- So können wir bequem einfache Anführungszeichen in Strings haben, die mit einfache Anführungszeichen begrenzt sind. Die Backslashes verschwinden dann in der Ausgabe.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\"Hello World!\")  
"Hello World!"  
>>> print('\Hello World\')
```

# Strings Escaping



- So können wir bequem einfache Anführungszeichen in Strings haben, die mit einfache Anführungszeichen begrenzt sind. Die Backslashes verschwinden dann in der Ausgabe.

```
tweise@weise-laptop: ~  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\"Hello World!\")")  
"Hello World!"  
>>> print('\ 'Hello World\')  
'Hello World'  
>>> □
```

# Strings Escaping



- Aber was machen wir, wenn wir ein Backslash (“\”) in einem String haben wollen?

```
tweise@weise-laptop: ~  
>>> print("A single quotation mark: '")  
A single quotation mark: '  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\\"Hello World!\\"")  
"Hello World!"  
>>> print('\\"Hello World\\""')  
'Hello World'  
>>> print("\\")
```

# Strings Escaping



- Aber was machen wir, wenn wir ein Backslash (“\”) in einem String haben wollen? Einfach noch ein Backslash davor schreiben. Aus dem doppelten Backslash wird dann ein einfaches Backslash.

```
tweise@weise-laptop: ~  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\"Hello World!\")  
"Hello World!"  
>>> print('\ 'Hello World\ ' ')  
'Hello World'  
>>> print("\\")  
\  
>>> □
```

# Strings Escaping



- Natürlich können wir beliebige viele Escape-Sequenzen in unseren Strings verwenden.

```
tweise@weise-laptop: ~  
>>> print('A double quotation mark: "')  
A double quotation mark: "  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\"Hello World!\")  
"Hello World!"  
>>> print('\\"Hello World\\')  
'Hello World'  
>>> print("\\")  
\  
>>> print("\\\"'\\\\\\\"")
```

# Strings Escaping



- Natürlich können wir beliebige viele Escape-Sequenzen in unseren Strings verwenden. Der String `"\"'\\\"\\\""` beginnt und endet mit `"`. Das erste Zeichen ist ein durch `\` escapedes `"`. Darauf folgt ein escapedes einfaches Anführungszeichen, ein escapedes Backslash, ein zweites escapedes doppeltes Anführungszeichen, und ein drittes escapedes doppeltes Anführungszeichen.

```
tweise@weise-laptop: ~  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\"Hello World!\")  
"Hello World!"  
>>> print('\\"Hello World\\')  
'Hello World'  
>>> print("\\")  
\  
>>> print("\"\"'\\\"\\\"")  
"\"'\""  
>>> 
```

# Strings Escaping



- Die Sequenz `\n` steht für “Newline”, also den Anfang einer neuen Zeile.

```
tweise@weise-laptop: ~  
>>> print("A single (') and a double (\") quotation mark.")  
A single (') and a double (") quotation mark.  
>>> print("\"Hello World!\")  
"Hello World!"  
>>> print('\`Hello World\`')  
'Hello World'  
>>> print("\\")  
\  
>>> print("\`\\\"")  
"\""  
>>> print("Hello\nWorld!")
```

# Strings Escaping



- Die Sequenz `\n` steht für “Newline”, also den Anfang einer neuen Zeile. Somit können wir mehrzeilige Strings in einer einzigen Zeile schreiben.

```
tweise@weise-laptop: ~  
"Hello World!"  
>>> print('\ 'Hello World\ ')  
'Hello World'  
>>> print("\\")  
\  
>>> print("\ \"'\\\ \"'")  
" \"'  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> 
```

# Strings Escaping



- Unter Windows kann man auch die Sequenz `\r\n` verwenden.

```
tweise@weise-laptop: ~  
"Hello World!"  
>>> print('\ 'Hello World\ ')  
'Hello World'  
>>> print("\\")  
\  
>>> print("\ \"\\\"")  
" \""  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")
```

# Strings Escaping



- Unter Windows kann man auch die Sequenz `\r\n` verwenden. Muss man aber nicht, Python gibt auch mit `\n` den richtigen Text aus.

```
tweise@weise-laptop: ~  
>>> print("\\")  
\  
>>> print("\"'\\\\"")  
"'\\""  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> 
```

# Strings Escaping



- Die Sequenz `\t` steht für “Tabulator”, welcher oft equivalent zu mehreren (oft 4) Leerzeichen ausgegeben wird. Das braucht man selten.

```
tweise@weise-laptop: ~  
>>> print("\\")  
\  
>>> print("\"'\\\"'")  
"'\t"  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> print("The horizontal tab is like a bigger space: '\t'.")
```

# Strings Escaping



- Die Sequenz `\t` steht für “Tabulator”, welcher oft equivalent zu mehreren (oft 4) Leerzeichen ausgegeben wird. Das braucht man selten.

```
tweise@weise-laptop: ~  
>>> print("\"'\\\\"")  
"'\\""  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> print("The horizontal tab is like a bigger space: '\t'.")  
The horizontal tab is like a bigger space: '    '  
>>> 
```

# Strings Escaping



- Wollen wir einen String über mehrere Zeilen hinweg schreiben, dann können wir ans Ende der Zeile einfach ein Backslash (“\”) schreiben.

```
tweise@weise-laptop: ~  
>>> print("\"'\\\\"")  
\"'  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> print("The horizontal tab is like a bigger space: '\t'.")  
The horizontal tab is like a bigger space: '    '  
>>> print("Hello \u25a1")
```

# Strings Escaping



- Wollen wir einen String über mehrere Zeilen hinweg schreiben, dann können wir ans Ende der Zeile einfach ein Backslash (“\”) schreiben. Wenn wir dann `Enter` drücken, geht es in der nächsten Zeile einfach weiter, die im Interpreter dann mit drei Punkten beginnt.

```
tweise@weise-laptop: ~  
""\n"  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> print("The horizontal tab is like a bigger space: '\t'.")  
The horizontal tab is like a bigger space: '    '.  
>>> print("Hello \  
... █
```

# Strings Escaping



- Wollen wir einen String über mehrere Zeilen hinweg schreiben, dann können wir ans Ende der Zeile einfach ein Backslash (“\”) schreiben. Wenn wir dann `Enter` drücken, geht es in der nächsten Zeile einfach weiter, die im Interpreter dann mit drei Punkten beginnt. So können wir den Text weiterschreiben.

```
tweise@weise-laptop: ~  
""\n"  
>>> print("Hello\nWorld!")  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> print("The horizontal tab is like a bigger space: '\t'.")  
The horizontal tab is like a bigger space: '    '.  
>>> print("Hello \  
... World!")
```

# Strings Escaping



- Wollen wir einen String über mehrere Zeilen hinweg schreiben, dann können wir ans Ende der Zeile einfach ein Backslash (“\”) schreiben. Wenn wir dann `Enter` drücken, geht es in der nächsten Zeile einfach weiter, die im Interpreter dann mit drei Punkten beginnt. So können wir den Text weiterschreiben. Der String beinhaltet dann keinen Zeilenumbruch.

```
tweise@weise-laptop: ~  
Hello  
World!  
>>> print("Hello\r\nWorld!")  
Hello  
World!  
>>> print("The horizontal tab is like a bigger space: '\t'.")  
The horizontal tab is like a bigger space: '    '.  
>>> print("Hello \  
... World!")  
Hello World!  
>>> 
```

# Strings Escaping



- Zeichenketten werden durch Anführungszeichen begrenzt.
- Was machen wir, wenn wir Anführungszeichen **in** einer Zeichenkette haben wollen?
- Aber was machen wir, wenn wir **beide** Arten von Anführungszeichen brauchen? Dann können wir nicht auf "andere" Anführungszeichen zur Begrenzung der Strings ausweichen.
- Dann brauchen wir sogenannte Escape-Sequenzen<sup>7</sup>.
- Wir können dem doppelten Anführungszeichen einfach ein Backslash ("\\") voranstellen. Dann wird es nicht mehr als String-Begrenzer interpretiert und verwandelt sich in ein normales doppeltes Anführungszeichen.
- Escape-Sequenzen sind also nützliche Werkzeuge, um Zeichen in Strings zu schreiben, die wir sonst nicht schreiben könnten.

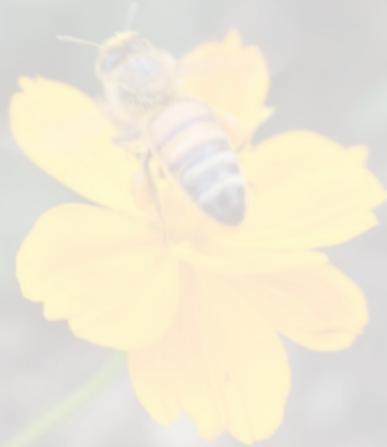


## Mehrzeilige Strings



## Mehrzeilige Strings

- Wir können Strings erstellen, die über mehrere Zeilen gehen, in dem wir die Escape-Sequenz `\n` einfügen.



## Mehrzeilige Strings



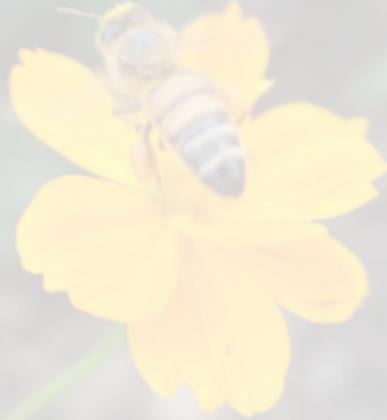
- Wir können Strings erstellen, die über mehrere Zeilen gehen, in dem wir die Escape-Sequenz `\n` einfügen.
- Wenn die Strings lang werden, dann kann das aber zu häßlichen sehr langen Kodezeilen führen.



# Mehrzeilige Strings



- Wir können Strings erstellen, die über mehrere Zeilen gehen, in dem wir die Escape-Sequenz `\n` einfügen.
- Wenn die Strings lang werden, dann kann das aber zu häßlichen sehr langen Kodezeilen führen.
- Für lange mehrzeilige Strings gibt es die Syntax mit dreifachen doppelten Anführungszeichen: `"""..."""`.



# Mehrzeilige Strings



- Für lange mehrzeilige Strings gibt es die Syntax mit dreifachen doppelten Anführungszeichen: `"""..."""`.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```



# Mehrzeilige Strings



- Wir beginnen solche Strings immer mit `"""` anstelle von `"`. Machen wir einen Zeilenumbruch in so einem String...

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
...  
""")
```

# Mehrzeilige Strings



- Wir beginnen solche Strings immer mit `"""` anstelle von `"`. Machen wir einen Zeilenumbruch in so einem String, dann können wir wieder auf der nächsten Zeile weiterschreiben...

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
""")
```

# Mehrzeilige Strings



- Wir beginnen solche Strings immer mit `"""` anstelle von `"`. Machen wir einen Zeilenumbruch in so einem String, dann können wir wieder auf der nächsten Zeile weiterschreiben, die im Interpreter wieder mit drei Punkten anfängt.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... """)
```

# Mehrzeilige Strings



- Wir beginnen solche Strings immer mit `"""` anstelle von `"`. Machen wir einen Zeilenumbruch in so einem String, dann können wir wieder auf der nächsten Zeile weiterschreiben, die im Interpreter wieder mit drei Punkten anfängt. Wir können so beliebig viele Zeilenumbrüche schreiben.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... This linebreak is then part of the string.""")
```

# Mehrzeilige Strings



- Wir beginnen solche Strings immer mit `"""` anstelle von `"`. Machen wir einen Zeilenumbruch in so einem String, dann können wir wieder auf der nächsten Zeile weiterschreiben, die im Interpreter wieder mit drei Punkten anfängt. Wir können so beliebig viele Zeilenumbrüche schreiben. Diese sind dann Teil des Strings (anders als bei `\n` an Zeilenende!).

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... This linebreak is then part of the string.""")  
This is a multi-line string.  
I can hit enter to begin a new line.  
This linebreak is then part of the string.  
>>> 
```

# Mehrzeilige Strings



- Diese mehrzeilige Syntax lässt sich mit f-Strings kombinieren.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... This linebreak is then part of the string.""")  
This is a multi-line string.  
I can hit enter to begin a new line.  
This linebreak is then part of the string.  
>>> print(f"""This is a multi-line f-string.□
```



# Mehrzeilige Strings



- Diese mehrzeilige Syntax lässt sich mit f-Strings kombinieren. Solche mehrzeiligen f-Strings sind von `f"""..."""` begrenzt. Sie können beliebige Ausdrücke enthalten und normal interpoliert.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... This linebreak is then part of the string.""")  
This is a multi-line string.  
I can hit enter to begin a new line.  
This linebreak is then part of the string.  
>>> print(f"""This is a multi-line f-string.  
... It can contain expressions like {(5 ** 7) / 3123:.2f}""")
```

# Mehrzeilige Strings



- Diese mehrzeilige Syntax lässt sich mit f-Strings kombinieren. Solche mehrzeiligen f-Strings sind von `f"""..."""` begrenzt. Sie können beliebige Ausdrücke enthalten und normal interpoliert.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... This linebreak is then part of the string.""")  
This is a multi-line string.  
I can hit enter to begin a new line.  
This linebreak is then part of the string.  
>>> print(f"""This is a multi-line f-string.  
... It can contain expressions like {(5 ** 7) / 3123:.2f}  
... """)
```

# Mehrzeilige Strings



- Diese mehrzeilige Syntax lässt sich mit f-Strings kombinieren. Solche mehrzeiligen f-Strings sind von `f"""..."""` begrenzt. Sie können beliebige Ausdrücke enthalten und normal interpoliert.

```
tweise@weise-laptop: ~  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("""This is a multi-line string.  
... I can hit enter to begin a new line.  
... This linebreak is then part of the string.""")  
This is a multi-line string.  
I can hit enter to begin a new line.  
This linebreak is then part of the string.  
>>> print(f"""This is a multi-line f-string.  
... It can contain expressions like {(5 ** 7) / 3123:.2f}  
... or {(5 < 9) and ((5.5 / 3) < (2 * 11))}.""")
```

# Mehrzeilige Strings

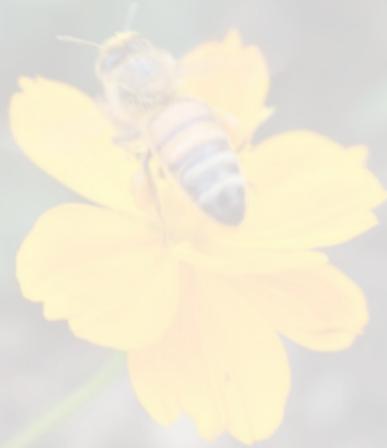


- Diese mehrzeilige Syntax lässt sich mit f-Strings kombinieren. Solche mehrzeiligen f-Strings sind von `f"""..."""` begrenzt. Sie können beliebige Ausdrücke enthalten und normal interpoliert.

```
tweise@weise-laptop: ~  
... This linebreak is then part of the string."""  
This is a multi-line string.  
I can hit enter to begin a new line.  
This linebreak is then part of the string.  
>>> print(f"""This is a multi-line f-string.  
... It can contain expressions like {(5 ** 7) / 3123:.2f}  
... or {(5 < 9) and ((5.5 / 3) < (2 * 11))}.""")  
This is a multi-line f-string.  
It can contain expressions like 25.02  
or True.  
>>> 
```

## Mehrzeilige Strings

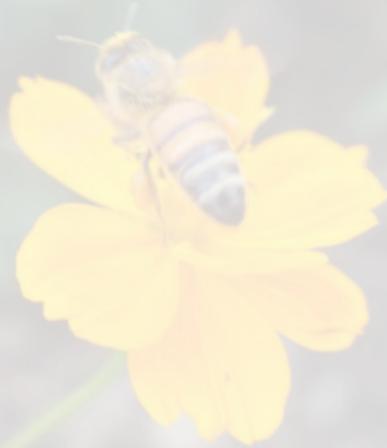
- Somit haben wir nun auch eine Methode, große Textstücke als Strings in unsere Programme einzufügen.



## Mehrzeilige Strings



- Somit haben wir nun auch eine Methode, große Textstücke als Strings in unsere Programme einzufügen.
- Das wird später bei der Dokumentation von Kode nützlich.



# Mehrzeilige Strings



- Somit haben wir nun auch eine Methode, große Textstücke als Strings in unsere Programme einzufügen.
- Das wird später bei der Dokumentation von Code nützlich.

## Gute Praxis

Wenn wir mehrzeilige Strings definieren, dann sollte immer doppelte Anführungszeichen verwendet werden und nicht einfache (natürlich immer in dreifacher Ausfertigung).<sup>13,32</sup>



# Unicode



## Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.
- Während Python “beliebig-große” `ints` anbietet, die aus entsprechend vielen Bytes bestehen, nutzen wir in anderen Programmiersprachen oftmals jeweils 8 Bytes, um einen Integer darzustellen.

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.
- Während Python “beliebig-große” `ints` anbietet, die aus entsprechend vielen Bytes bestehen, nutzen wir in anderen Programmiersprachen oftmals jeweils 8 Bytes, um einen Integer darzustellen.
- Der Datentyp `float` ist in Python ebenfalls 8 Bytes groß – wie diese interpretiert werden haben Sie schon gelernt.

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.
- Während Python “beliebig-große” `ints` anbietet, die aus entsprechend vielen Bytes bestehen, nutzen wir in anderen Programmiersprachen oftmals jeweils 8 Bytes, um einen Integer darzustellen.
- Der Datentyp `float` ist in Python ebenfalls 8 Bytes groß – wie diese interpretiert werden haben Sie schon gelernt.
- Aber wie funktioniert das mit Text?

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.
- Während Python “beliebig-große” `ints` anbietet, die aus entsprechend vielen Bytes bestehen, nutzen wir in anderen Programmiersprachen oftmals jeweils 8 Bytes, um einen Integer darzustellen.
- Der Datentyp `float` ist in Python ebenfalls 8 Bytes groß – wie diese interpretiert werden haben Sie schon gelernt.
- Aber wie funktioniert das mit Text?
- In dem jedes Zeichen eines Textes als eine Zahl dargestellt wird.

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.
- Während Python “beliebig-große” `ints` anbietet, die aus entsprechend vielen Bytes bestehen, nutzen wir in anderen Programmiersprachen oftmals jeweils 8 Bytes, um einen Integer darzustellen.
- Der Datentyp `float` ist in Python ebenfalls 8 Bytes groß – wie diese interpretiert werden haben Sie schon gelernt.
- Aber wie funktioniert das mit Text?
- In dem jedes Zeichen eines Textes als eine Zahl dargestellt wird.
- Ein `str` ist dann nichts als eine Sequenz von Zahlen.

# Texte als Zahlen



- Der Speicher unserer Computer speichert Gruppen von Bits in fester Größe, sagen wir, Bytes zu jeweils 8 Bit.
- Diese Bytes werden oft als Ganzzahlen betrachtet.
- Während Python “beliebig-große” `ints` anbietet, die aus entsprechend vielen Bytes bestehen, nutzen wir in anderen Programmiersprachen oftmals jeweils 8 Bytes, um einen Integer darzustellen.
- Der Datentyp `float` ist in Python ebenfalls 8 Bytes groß – wie diese interpretiert werden haben Sie schon gelernt.
- Aber wie funktioniert das mit Text?
- In dem jedes Zeichen eines Textes als eine Zahl dargestellt wird.
- Ein `str` ist dann nichts als eine Sequenz von Zahlen.
- Das System weiß, welches Zeichen zu welcher Zahl gehört.

# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.



# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.
- Die bekannteste solche Zuordnung ist ASCII<sup>1,31</sup>, welche sieben Bits pro Zeichen verwendete.



# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.
- Die bekannteste solche Zuordnung ist ASCII<sup>1,31</sup>, welche sieben Bits pro Zeichen verwendete.
- Darum konnte ASCII auch nur Lateinische Buchstaben, Satzzeichen, Ziffern, und ein Kontrollzeichen (wie `\t` und `\n...`) darstellen.

# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.
- Die bekannteste solche Zuordnung ist ASCII<sup>1,31</sup>, welche sieben Bits pro Zeichen verwendete.
- Darum konnte ASCII auch nur Lateinische Buchstaben, Satzzeichen, Ziffern, und ein Kontrollzeichen (wie `\t` und `\n...`) darstellen.
- Ihnen ist vielleicht aufgefallen, dass verschiedene Sprachen verschiedene Zeichen verwenden ... und zwar mehr als insgesamt  $2^7 = 128$ .

# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.
- Die bekannteste solche Zuordnung ist ASCII<sup>1,31</sup>, welche sieben Bits pro Zeichen verwendete.
- Darum konnte ASCII auch nur Lateinische Buchstaben, Satzzeichen, Ziffern, und ein Kontrollzeichen (wie `\t` und `\n...`) darstellen.
- Ihnen ist vielleicht aufgefallen, dass verschiedene Sprachen verschiedene Zeichen verwenden ... und zwar mehr als insgesamt  $2^7 = 128$ .
- Deshalb gibt es verschiedene solche Zuordnungen.

# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.
- Die bekannteste solche Zuordnung ist ASCII<sup>1,31</sup>, welche sieben Bits pro Zeichen verwendete.
- Darum konnte ASCII auch nur Lateinische Buchstaben, Satzzeichen, Ziffern, und ein Kontrollzeichen (wie `\t` und `\n...`) darstellen.
- Ihnen ist vielleicht aufgefallen, dass verschiedene Sprachen verschiedene Zeichen verwenden ... und zwar mehr als insgesamt  $2^7 = 128$ .
- Deshalb gibt es verschiedene solche Zuordnungen.
- In China existieren verschiedene, auf Chinesische Zeichen spezialisierte, Zuordnungen, z.B. das historische GB 2312<sup>36</sup>, GBK<sup>38</sup>, und das neuere GB 18030<sup>37</sup>.

# Zeichen als Zahlen



- Historisch gesehen gibt es viele “Zeichen-zu-Zahlen” Mappings.
- Die bekannteste solche Zuordnung ist ASCII<sup>1,31</sup>, welche sieben Bits pro Zeichen verwendete.
- Darum konnte ASCII auch nur Lateinische Buchstaben, Satzzeichen, Ziffern, und ein Kontrollzeichen (wie `\t` und `\n...`) darstellen.
- Ihnen ist vielleicht aufgefallen, dass verschiedene Sprachen verschiedene Zeichen verwenden ... und zwar mehr als insgesamt  $2^7 = 128$ .
- Deshalb gibt es verschiedene solche Zuordnungen.
- In China existieren verschiedene, auf Chinesische Zeichen spezialisierte, Zuordnungen, z.B. das historische GB 2312<sup>36</sup>, GBK<sup>38</sup>, und das neuere GB 18030<sup>37</sup>.
- Die große Mehrheit der Systeme heute (und Python) verwendet einen gemeinsamen Standard für **alle** Sprachen: Unicode<sup>18,28,30</sup>.

# Unicode Beispiel

- Jedem Zeichen ist eine Zahl zugeordnet. Gucken wir uns das mal an.



# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		、	。	、	々	々	〇	<	>	<	>	「	」	『	』	
301		【	】	〒	=	(	)	【	】	(	)	[	]	~	"	"
4f6	你	仙	佢	佣	佢	金	佻	佻	佻	佩	佢	佢	佢	佢	佢	佢
4f7	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢
597	巽	巽	巽	女	奴	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶
598	改	灼	如	妃	妾	虹	妆	妇	妈	媵	妊	妹	妍	妍	妍	妍

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		、	。	、	々	々	〇	<	>	<	>	「	」	『	』	
301		【	】	〒	≡	(	)	【	】	(	)	【	】	~	"	"
4f6	你	伧	佢	佣	佻	金	佻	佻	佻	佩	佻	佻	佻	佻	佻	佻
4f7	佰	金	佻	佳	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	巽	巽	巽	女	奴	奶	奸	奸	她	效	效	效	好	妯	妯	
598	改	灼	如	妃	妄	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你

\u

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	↗	○	<	>	《	》	「	」	『	』	
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	佣	佢	金	佻	佻	佻	佩	佻	佻	佻	佻	佻	佻
4f7	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	嬖	嬖	嬖	女	奴	奶	奸	奸	她	姦	姦	好	姦	姦	姦	姦
598	改	灼	如	妃	妾	姦	妆	妇	妈	姦	姦	姦	姦	姦	姦	姦

你

\u

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	.	/	⊙	々	ノ	○	<	>	《	》	「	」	『	』
301		【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"
4f6		<b>你</b>	你	你	你	你	你	你	你	你	你	你	你	你	你	你
4f7		你	你	你	你	你	你	你	你	你	你	你	你	你	你	你
597		你	你	你	你	你	你	你	你	你	你	你	你	你	你	你
598		你	你	你	你	你	你	你	你	你	你	你	你	你	你	你

**你**

\u

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	佣	佻	金	佑	侉	侗	佩	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	巽	女	奴	奶	奸	奸	她	效	效	效	好	妯	妯	
598	改	灼	如	妃	妾	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你

\u4f6

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	侗	侩	金	佻	侉	佩	侗	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	巽	女	奴	奶	奸	奸	她	效	效	效	好	妯	妯	
598	改	灼	如	妃	妾	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你

\u4f6

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	"
4f6	你	仙	佢	佣	佢	金	佻	佻	佻	佩	佻	佻	佻	佻	佻	佻
4f7	佰	金	佻	佳	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	巽	巽	巽	女	奴	奶	奸	奸	她	效	效	效	好	妯	妯	
598	改	灼	如	妃	妾	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你

\u4f60

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	侗	侩	金	佑	侉	侗	佩	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	巽	女	奴	奶	奸	奸	她	效	效	效	好	妯	妯	
598	改	灼	如	妃	妾	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你 好

\u4f60 \u

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	侗	侩	金	佑	侉	侗	佩	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	巽	女	奴	奶	奸	奸	她	效	效	效	好	妯	妯	
598	改	灼	如	妃	妾	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你 好

\u4f60 \u

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	佣	佢	金	佻	佻	佻	佩	佻	佻	佻	佻	佻	佻
4f7	佰	金	佻	佳	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	龠	龠	龠	女	奴	奶	奸	奸	她	效	姣	姣	姣	姣	姣	姣
598	改	灼	如	妃	妄	虹	妆	妇	妈	媵	妊	妹	妍	妍	妍	妍

你 好

\u4f60 \u

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300	,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』		
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	„
4f6	你	仙	佢	佣	佢	金	佑	佻	佻	佩	佻	佻	佻	佻	佻	佻
4f7	佰	金	佻	佳	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	龠	龠	龠	女	奴	奶	奸	奸	她	效	姁	姁	姁	好	妣	妣
598	改	灼	如	妃	妄	虹	妆	妇	妈	妣	妊	妹	妍	妍	妍	妍

你 好

\u4f60

\u597d

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	.	/	⊙	々	〇	<	>	《	》	「	」	『	』	
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	"
4f6	你	仙	佢	佣	佻	金	佑	侉	侗	侗	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	禪	女	奴	妲	奶	奸	奸	她	效	姣	姣	好	妲	妲
598	改	灼	如	妃	妾	虹	妆	妇	妈	妲	妊	妹	姘	妍	妍	妍

你 好

\u4f60 \u597d

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	.	/	⊙	々	ノ	〇	<	>	<	>	「	」	『	』
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	"
4f6	你	仙	佢	佣	佻	金	佑	侉	侗	佩	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	禪	女	奴	奶	奶	奸	奸	她	效	效	效	好	妯	妯
598	改	灼	如	妃	妄	虹	妆	妇	妈	妯	妊	妹	妍	妍	妍	妍

你 好

\u4f60

\u597d

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	.	/	⊙	々	ノ	〇	<	>	<	>	「	」	『	』
301	【	】	〒	≡	(	)	【	】	(	)	[	]	~	"	"	"
4f6	你	仙	佢	佣	佻	金	佑	侉	侗	佩	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	龠	龠	龠	女	奴	妲	奶	奸	奸	她	姦	姦	姦	好	妲	妲
598	改	灼	如	妃	妾	虹	妆	妇	妈	媵	妊	妹	媵	妍	妍	妍

你 好 。

\u4f60 \u597d \u002e

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	。	、	⊙	々	ノ	〇	<	>	<	>	「	」	『	』
301	【	】	〒	＝	（	）	【	】	（	）	【	】	～	“	”	„
4f6	你	仙	佢	佣	佢	金	佑	佻	佻	佩	佻	佻	佻	佻	佻	佻
4f7	佰	金	佻	佳	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	巽	巽	巽	女	奴	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶
598	改	灼	如	妃	妄	虹	妆	妇	妈	媵	妊	妹	妍	妍	妍	妍

你 好 。

\u4f60 \u597d \u002c

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~		
300		、	。	、	、	々	々	、	〇	<	>	<	>	「	」	『	』
301	【	】	〒	=	(	)	【	】	(	)	【	】	~	"	"	"	"
4f6	你	仙	佢	佣	佢	金	佑	佢	佢	佩	佢	佢	佬	佬	佢	佢	佢
4f7	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢	佢
597	吳	美	禱	女	奴	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶
598	改	灼	如	妃	妄	虹	妝	婦	媽	媼	妊	妹	媿	妍	妍	妍	妍

你 好 。

\u4f60 \u597d \u002e

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~		
300		、	。	、	、	々	々	、	〇	<	>	<	>	「	」	『	』
301	【	】	〒	＝	（	）	【	】	（	）	【	】	～	“	”	„	
4f6	你	仙	佢	佣	佻	金	佑	侉	侗	佩	侗	侗	侗	侗	侗	侗	侗
4f7	佰	金	侗	佳	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗	侗
597	巽	巽	禱	女	奴	妲	奶	奸	奸	她	效	姝	姝	姝	好	妲	妲
598	改	灼	如	妃	妾	虹	妆	妇	妈	妲	妊	妹	姘	妍	妍	妍	妍

你 好 。

\u4f60 \u597d \u300d

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	。	、	⊙	々	ノ	〇	<	>	《	》	「	」	『	』
301	【	】	〒	=	(	)	【	】	(	)	[	]	~	"	"	"
4f6	你	仙	佢	佣	佢	金	佑	佻	佻	佩	佻	佻	佬	佬	佻	佻
4f7	佰	金	佬	佳	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻	佻
597	巽	巽	禱	女	奴	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶
598	改	灼	如	妃	妄	虹	妆	妇	妈	媪	妊	妹	妍	妍	妍	妍

你 好 。

\u4f60 \u597d \u3002

# Unicode Beispiel



\u	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
300		,	。	、	⊙	々	ノ	〇	<	>	《	》	「	」	『	』
301	【	】	〒	=	(	)	【	】	(	)	[	]	~	"	"	"
4f6	你	仙	佢	佣	佢	金	佑	侬	侬	佩	侬	侬	侬	侬	侬	侬
4f7	佰	金	侬	佳	侬	侬	侬	侬	侬	侬	侬	侬	侬	侬	侬	侬
597	巽	巽	禪	女	奴	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶	奶
598	改	灼	如	妃	妾	虹	妆	妇	妈	媪	妊	妹	姘	妍	妍	妍

你

好

。

\u4f60

\u597d

\u3002

# Unicode Beispiel



- Jedem Zeichen ist eine Zahl zugeordnet.
- “你” entspricht der Hexadezimalzahl 4f60, “好” ist durch die Hexadezimalzahl 597d repräsentiert, und der Voll-weite Satzpunkt “。” entspricht der Hexadezimalzahl 3002.

# Unicode Beispiel



- Jedem Zeichen ist eine Zahl zugeordnet.
- “你” entspricht der Hexadezimalzahl 4f60, “好” ist durch die Hexadezimalzahl 597d repräsentiert, und der Voll-weite Satzpunkt “。” entspricht der Hexadezimalzahl 3002.
- Alle Zeichen können als so-genanntes Unicode-Escape, also `\u` gefolgt von der entsprechenden Hexadezimalzahl geschrieben werden.

# Unicode Beispiel



- Jedem Zeichen ist eine Zahl zugeordnet.
- “你” entspricht der Hexadezimalzahl 4f60, “好” ist durch die Hexadezimalzahl 597d repräsentiert, und der Voll-weite Satzpunkt “。” entspricht der Hexadezimalzahl 3002.
- Alle Zeichen können als so-genanntes Unicode-Escape, also `\u` gefolgt von der entsprechenden Hexadezimalzahl geschrieben werden.
- Gucken wir uns das mal an.

# Unicode Beispiel

- Gucken wir uns das mal an.



```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

# Unicode Beispiel



- Wir verwenden die Unicode-Escapes `\u4f60`, `\u597d`, und `\u3002` für die drei Zeichen “你好”.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("\u4f60\u597d\u3002")
```

# Unicode Beispiel



- Wir verwenden die Unicode-Escapes `\u4f60`, `\u597d`, und `\u3002` für die drei Zeichen “你好。”. Und tatsächlich wird der korrekte Text gedruckt.

```
tweise@weise-laptop: ~  
tweise@weise-laptop:~$ python3  
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("\u4f60\u597d\u3002")  
你好。  
>>> █
```

# Unicode Beispiel



- Jedem Zeichen ist eine Zahl zugeordnet.
- “你” entspricht der Hexadezimalzahl 4f60, “好” ist durch die Hexadezimalzahl 597d repräsentiert, und der Voll-weite Satzpunkt “。” entspricht der Hexadezimalzahl 3002.
- Alle Zeichen können als so-geanntes Unicode-Escape, also `\u` gefolgt von der entsprechenden Hexadezimalzahl geschrieben werden.
- Wir verwenden die Unicode-Escapes `\u4f60`, `\u597d`, und `\u3002` für die drei Zeichen “你好。”. Und tatsächlich wird der korrekte Text gedruckt.
- Damit haben Sie nun also auch eine Vorstellung, wie Text im Speicher des Computers dargestellt wird.



# Zusammenfassung



# Zusammenfassung

- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.



# Zusammenfassung



- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.

# Zusammenfassung



- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.
- Wir wandeln sie dann in andere Datentypen, with `int` oder `float` um, um mit ihnen zu arbeiten.

# Zusammenfassung



- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.
- Wir wandeln sie dann in andere Datentypen, with `int` oder `float` um, um mit ihnen zu arbeiten.
- Danach geben wir die Ergebnisse unserer Berechnungen wieder als Text aus. Dabei können wir z.B. f-Strings nutzen, um andere Datentypen elegant als Strings zu formatieren.

# Zusammenfassung



- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.
- Wir wandeln sie dann in andere Datentypen, with `int` oder `float` um, um mit ihnen zu arbeiten.
- Danach geben wir die Ergebnisse unserer Berechnungen wieder als Text aus. Dabei können wir z.B. f-Strings nutzen, um andere Datentypen elegant als Strings zu formatieren.
- Manchmal wollen wir auch mit Strings direkt arbeiten, z.B. Teile von ihnen extrahieren oder Groß- in Kleinbuchstaben umwandeln.

# Zusammenfassung



- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.
- Wir wandeln sie dann in andere Datentypen, with `int` oder `float` um, um mit ihnen zu arbeiten.
- Danach geben wir die Ergebnisse unserer Berechnungen wieder als Text aus. Dabei können wir z.B. f-Strings nutzen, um andere Datentypen elegant als Strings zu formatieren.
- Manchmal wollen wir auch mit Strings direkt arbeiten, z.B. Teile von ihnen extrahieren oder Groß- in Kleinbuchstaben umwandeln.
- Wir können Sonderzeichen durch Escape-Sequenzen in Strings einführen und mehrzeilige Strings erstellen.

# Zusammenfassung



- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.
- Wir wandeln sie dann in andere Datentypen, with `int` oder `float` um, um mit ihnen zu arbeiten.
- Danach geben wir die Ergebnisse unserer Berechnungen wieder als Text aus. Dabei können wir z.B. f-Strings nutzen, um andere Datentypen elegant als Strings zu formatieren.
- Manchmal wollen wir auch mit Strings direkt arbeiten, z.B. Teile von ihnen extrahieren oder Groß- in Kleinbuchstaben umwandeln.
- Wir können Sonderzeichen durch Escape-Sequenzen in Strings einführen und mehrzeilige Strings erstellen.
- Intern werden die Zeichen einer Zeichenkette durch Zahlen repräsentiert, wobei die Zuordnung von Zahlen zu Zeichen meist auf Unicode basiert.

# Zusammenfassung



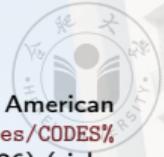
- Zeichenketten, Strings, also Instanzen der Klasse `str` sind wichtig.
- Oftmals bekommen wir Daten in Form von solchen Zeichenketten als Input für unsere Programme.
- Wir wandeln sie dann in andere Datentypen, with `int` oder `float` um, um mit ihnen zu arbeiten.
- Danach geben wir die Ergebnisse unserer Berechnungen wieder als Text aus. Dabei können wir z.B. f-Strings nutzen, um andere Datentypen elegant als Strings zu formatieren.
- Manchmal wollen wir auch mit Strings direkt arbeiten, z.B. Teile von ihnen extrahieren oder Groß- in Kleinbuchstaben umwandeln.
- Wir können Sonderzeichen durch Escape-Sequenzen in Strings einführen und mehrzeilige Strings erstellen.
- Intern werden die Zeichen einer Zeichenkette durch Zahlen repräsentiert, wobei die Zuordnung von Zahlen zu Zeichen meist auf Unicode basiert.
- Das war schon recht viel. Hat aber auch Spaß gemacht.



谢谢您们!  
Thank you!  
Vielen Dank!



# References I



- [1] *American Standard Code for Information Interchange (ASCII 1963)*. Techn. Ber. ASA X3.4-1963. New York, NY, USA: American Standards Association Incorporated, 17. Juni 1963. URL: <https://www.sensitiveresearch.com/Archive/CharCodeHist/Files/CODES%20standards%20documents%20ASCII%20Sean%20Leonard%200ct%202015/ASCII%2063,%20X3.4-1963.pdf> (besucht am 2024-07-26) (siehe S. 266–272).
- [2] 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. 315, 316).
- [3] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 315).
- [4] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 315).
- [5] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: <https://fstring.help> (besucht am 2024-07-25) (siehe S. 128–178, 315).
- [6] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 316).
- [7] "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](https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences) (besucht am 2025-08-05) (siehe S. 207–215, 236, 315).
- [8] 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. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: <https://www.researchgate.net/publication/301720080> (besucht am 2024-09-24). Translation of<sup>9</sup>. (Siehe S. 310).
- [9] 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<sup>8</sup> for a translation. (Siehe S. 310, 316).

# References II



- [10] Michael Filaseta. "The Transcendence of  $e$  and  $\pi$ ". 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. 316).
- [11] "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. 128–178, 315).
- [12] 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. 128–137, 177, 178, 315).
- [13] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: <https://peps.python.org/pep-0257> (besucht am 2024-07-27) (siehe S. 22, 238–256, 315).
- [14] 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. 128–137, 177, 178, 315).
- [15] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 315).
- [16] 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. 316).
- [17] 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. 316).

# References III



- [18] *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. 266–272, 316).
- [19] Arthur Jones, Kenneth R. Pearson und Sidney A. Morris. “Transcendence of  $e$  and  $\pi$ ”. 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. 316).
- [20] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 316).
- [21] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O’Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 316).
- [22] 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. 128–137, 177, 178, 315).
- [23] 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. 315).
- [24] Ivan Niven. “The Transcendence of  $\pi$ ”. *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. 316).
- [25] 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. 315).
- [26] 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. 128–178, 315).
- [27] “Text Sequence Type – `str`”. In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/stdtypes.html#textseq> (besucht am 2024-07-25) (siehe S. 5–9).

# References IV



- [28] *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. 266–272, 316).
- [29] 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. 315).
- [30] *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. 266–272, 316).
- [31] *USA Standard Code for Information Interchange (ASCII 1967)*. Techn. Ber. USAS X3.4-1967. New York, NY, USA: United States of America Standards Institute (USAS), 7. Juli 1967. URL: <https://www.sensitiveresearch.com/Archive/CharCodeHist/Files/CODES%20standards%20documents%20ASCII%20Sean%20Leonard%200ct%202015/ASCII%2068,%20X3.4-1967.pdf> (besucht am 2024-07-26) (siehe S. 266–272).
- [32] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: <https://peps.python.org/pep-0008> (besucht am 2024-07-27) (siehe S. 22, 238–256, 315).
- [33] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 315).
- [34] 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. 316).
- [35] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 315).
- [36] 信息交换用汉字编码字符集 (*Code of Chinese Graphic Character Set for Information Interchange – Primary Set*). 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) GB/T 2312-1980. China, Beijing (中国北京市): 中国国家标准化管理委员会 (Standardization Administration of the People's Republic of China, SAC), 1980–1. Mai 1981. URL: <https://openstd.samr.gov.cn/bz/gk/gb/newGbInfo?hcno=5664A728BD9D523DE3B99BC37AC7A2CC> (besucht am 2024-07-26) (siehe S. 266–272).

# References V



- [37] 信息技术中文编码字符集 (*Information Technology – Chinese Coded Character Set*). 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) GB 18030-2022. China, Beijing (中国北京市): 中华人民共和国工业和信息化部 (Ministry of Industry and Information Technology, MIIT): 国家市场监督管理总局 (State Administration for Market Regulation) und 中国国家标准化管理委员会 (Standardization Administration of the People's Republic of China, SAC), 19. Juli 2022–1. Aug. 2023. URL: <https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=A1931A578FE14957104988029B0833D3> (besucht am 2024-07-26) (siehe S. 266–272).
- [38] 汉字内码扩展规范 (*Chinese Internal Code Specification, National Standard Extended GBK*). 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) 1995 229. China, Beijing (中国北京市): 中华人民共和国全国信息技术标准化技术委员会 (National Information Technology Standardization Technical Committee of the People's Republic of China), 国家技术监督局 (National Bureau of Technical Supervision) und 中华人民共和国电子工业部 (Ministry of Electronics Industry of the People's Republic of China), 15. Dez. 1995 (siehe S. 266–272).

# 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<sup>4,23,35</sup>. Learn more at <https://www.gnu.org/software/bash>.

**docstring** Docstrings are special string constants in Python that contain documentation for modules or functions<sup>13</sup>. They must be delimited by `"""..."""`<sup>13,32</sup>.

**escape sequence** Escaping is the process of presenting “forbidden” characters or symbols in a sequence of characters or symbols. In Python, 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 (“\”)<sup>7</sup>. 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 `{`.

**f-string** let you include the results of expressions in strings<sup>5,11,12,14,22,26</sup>. 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

**Linux** is the leading open source operating system, i.e., a free alternative for Microsoft Windows<sup>2,15,25,29,33</sup>. We recommend using it for this course, for software development, and for research. Learn more at <https://www.linux.org>. Its variant Ubuntu is particularly easy to use and install.

**Microsoft Windows** is a commercial proprietary operating system<sup>3</sup>. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at <https://www.microsoft.com/windows>.

# Glossary (in English) II



**Python** The Python programming language<sup>17,20,21,34</sup>, i.e., what you will learn about in our book<sup>34</sup>. Learn more at <https://python.org>.

**(string) interpolation** In Python, string interpolation is the process where all the expressions in an f-string are evaluated and the final string is constructed. An example for string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`.

**terminal** A terminal is a text-based window where you can enter commands and execute them<sup>2,6</sup>. 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** `[Win]+[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<sup>6,16</sup>. 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<sup>18,28,30</sup>. 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.

$\pi$  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<sup>10,19,24</sup>, 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.

$e$  is Euler's number<sup>9</sup>, the base of the natural logarithm.  $e \in \mathbb{R}$  is an irrational and transcendental number<sup>10,19</sup>, 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.

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