合肥大學
HEFEI UNIVERSITY

Programming with Python

8. Der Datentyp float

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

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

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-11-04

Programming with Python

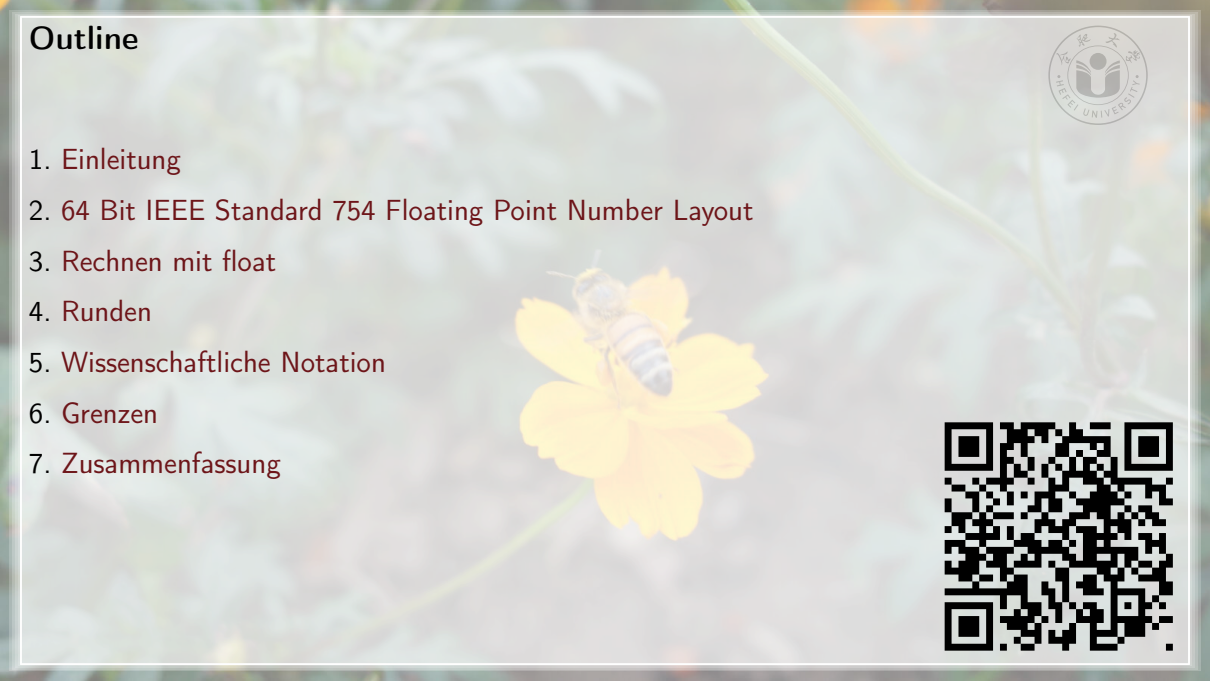


Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

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



Outline



1. Einleitung
2. 64 Bit IEEE Standard 754 Floating Point Number Layout
3. Rechnen mit float
4. Runden
5. Wissenschaftliche Notation
6. Grenzen
7. Zusammenfassung

Einleitung

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.
- Und gerne hätten wir dieses Feature auch für solche Zahlen!

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.
- Und gerne hätten wir dieses Feature auch für solche Zahlen!
- Das geht aber nicht.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.
- Und gerne hätten wir dieses Feature auch für solche Zahlen!
- Das geht aber nicht.
- Sie erinnern sich sicherlich an die Zahlen $\pi \approx 3.141\,592\,653\,590\ldots$ und $e \approx 2.718\,281\,828\,459\ldots$ aus dem Schulunterricht.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.
- Und gerne hätten wir dieses Feature auch für solche Zahlen!
- Das geht aber nicht.
- Sie erinnern sich sicherlich an die Zahlen $\pi \approx 3.141\,592\,653\,590\ldots$ und $e \approx 2.718\,281\,828\,459\ldots$ aus dem Schulunterricht.
- Sie sind beide irrational [37,51,63], was bedeutet, dass ihre Nachkommastellen niemals enden und auch keinem einfachen Schema folgen.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.
- Und gerne hätten wir dieses Feature auch für solche Zahlen!
- Das geht aber nicht.
- Sie erinnern sich sicherlich an die Zahlen $\pi \approx 3.141\,592\,653\,590\ldots$ und $e \approx 2.718\,281\,828\,459\ldots$ aus dem Schulunterricht.
- Sie sind beide irrational [37,51,63], was bedeutet, dass ihre Nachkommastellen niemals enden und auch keinem einfachen Schema folgen.
- Sie sind unendlich lang und wir würden unendlich viel Speicher brauchen, um sie zu exakt darzustellen.

Zu viele reelle Zahlen



- In der letzten Einheit haben wir über den Ganzzahldatentyp `int` gelernt.
- Eines der schönen Features der Sprache Python 3 ist, dass Ganzzahlen fast unbegrenzt groß seien dürfen.
- Es gibt nur einen einzigen Datentyp für Ganzzahlen, `int`, und der kann jeden Ganzzahlwert speichern, so fern der in den Arbeitsspeicher unseres Computers passt.
- Nun müssen wir aber auch mit reellen Zahlen aus der Menge $\mathbb{R}$ rechnen können.
- Und gerne hätten wir dieses Feature auch für solche Zahlen!
- Das geht aber nicht.
- Sie erinnern sich sicherlich an die Zahlen $\pi \approx 3.141\,592\,653\,590\ldots$ und $e \approx 2.718\,281\,828\,459\ldots$ aus dem Schulunterricht.
- Sie sind beide irrational [37,51,63], was bedeutet, dass ihre Nachkommastellen niemals enden und auch keinem einfachen Schema folgen.
- Sie sind unendlich lang und wir würden unendlich viel Speicher brauchen, um sie zu exakt darzustellen.
- Daher können wir das nicht und Python kann es auch nicht.

64 Bit IEEE Standard 754 Floating Point Number Layout



Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?

Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir die Zahlen nicht beliebig genau speichern können, selbst in so häufig vorkommenden Fällen wie π und e?

Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir die Zahlen nicht beliebig genau speichern können, selbst in so häufig vorkommenden Fällen wie π und e?
- Wie gehen wir damit um, dass es reelle Zahlen gibt so groß wie 10^{300} und so klein wie 10^{-300}?

Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir die Zahlen nicht beliebig genau speichern können, selbst in so häufig vorkommenden Fällen wie π und e?
- Wie gehen wir damit um, dass es reelle Zahlen gibt so groß wie 10^{300} und so klein wie 10^{-300}?
- In Python gibt es dafür den Datentyp `float`.

Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir die Zahlen nicht beliebig genau speichern können, selbst in so häufig vorkommenden Fällen wie π und e?
- Wie gehen wir damit um, dass es reelle Zahlen gibt so groß wie 10^{300} und so klein wie 10^{-300}?
- In Python gibt es dafür den Datentyp `float`.
- `float` hat die selbe interne Struktur wie der Datentyp `double` in C[64] – er folgt dem *64 Bit IEEE Standard 754 Floating Point Number Layout*[39,45,48].

Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir die Zahlen nicht beliebig genau speichern können, selbst in so häufig vorkommenden Fällen wie π und e?
- Wie gehen wir damit um, dass es reelle Zahlen gibt so groß wie 10^{300} und so klein wie 10^{-300}?
- In Python gibt es dafür den Datentyp `float`.
- `float` hat die selbe interne Struktur wie der Datentyp `double` in C[64] – er folgt dem *64 Bit IEEE Standard 754 Floating Point Number Layout*[39,45,48].
- Das Ziel ist es, dass wir eben genau Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.

Wie funktioniert das?



- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir die Zahlen nicht beliebig genau speichern können, selbst in so häufig vorkommenden Fällen wie π und e?
- Wie gehen wir damit um, dass es reelle Zahlen gibt so groß wie 10^{300} und so klein wie 10^{-300}?
- In Python gibt es dafür den Datentyp `float`.
- `float` hat die selbe interne Struktur wie der Datentyp `double` in C[64] – er folgt dem *64 Bit IEEE Standard 754 Floating Point Number Layout*[39,45,48].
- Das Ziel ist es, dass wir eben genau Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können, aber das wir auch akzeptieren, dass wir $10^{300} + 10^{-300}$ nicht exakt erfassen können.

64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.



64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- Die ersten 52 Bit – der so genannte Signifikand oder die Mantisse – speichern die Ziffern der Zahl.



64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- Die ersten 52 Bit – der so genannte Signifikand oder die Mantisse – speichern die Ziffern der Zahl.
- Mit 52 Bits können wir $52 \log_2 10 \approx 15$ bis 16 dezimale Ziffern darstellen.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- Die ersten 52 Bit – der so genannte Signifikand oder die Mantisse – speichern die Ziffern der Zahl.
- Mit 52 Bits können wir $52 \log_2 10 \approx 15$ bis 16 dezimale Ziffern darstellen.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- Die ersten 52 Bit – der so genannte Signifikand oder die Mantisse – speichern die Ziffern der Zahl.
- Mit 52 Bits können wir $52 \log_2 10 \approx 15$ bis 16 dezimale Ziffern darstellen.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Würden wir nur diese 52 Bits nehmen, dann könnten wir Zahlen von 0 bis $2^{52} - 1$ mit einer Auflösung von 1 darstellen.



64 Bit IEEE Standard 754 Floating Point Number Layout



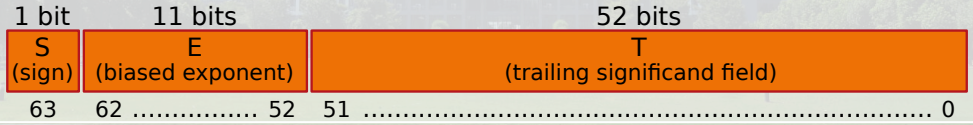
- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- Die ersten 52 Bit – der so genannte Signifikand oder die Mantisse – speichern die Ziffern der Zahl.
- Mit 52 Bits können wir $52 \log_2 10 \approx 15$ bis 16 dezimale Ziffern darstellen.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Würden wir nur diese 52 Bits nehmen, dann könnten wir Zahlen von 0 bis $2^{52} - 1$ mit einer Auflösung von 1 darstellen.
- Natürlich könnten wir auch eine andere Auflösung auswählen, sagen wir 0.001.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- Mit 52 Bits können wir $52 \log_2 10 \approx 15$ bis 16 dezimale Ziffern darstellen.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Würden wir nur diese 52 Bits nehmen, dann könnten wir Zahlen von 0 bis $2^{52} - 1$ mit einer Auflösung von 1 darstellen.
- Natürlich könnten wir auch eine andere Auflösung auswählen, sagen wir 0.001. Dann könnten wir Zahlen von 0 bis $0.001 * (2^{52} - 1)$ darstellen, und die kleinste Zahl größer als 0 wäre 0.001 anstelle von 1.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Würden wir nur diese 52 Bits nehmen, dann könnten wir Zahlen von 0 bis $2^{52} - 1$ mit einer Auflösung von 1 darstellen.
- Natürlich könnten wir auch eine andere Auflösung auswählen, sagen wir 0.001. Dann könnten wir Zahlen von 0 bis $0.001 * (2^{52} - 1)$ darstellen, und die kleinste Zahl größer als 0 wäre 0.001 anstelle von 1.
- Egal welche Auflösung wir auswählen würden, sie wäre gut in manchen Situationen und schlecht in anderen.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Würden wir nur diese 52 Bits nehmen, dann könnten wir Zahlen von 0 bis $2^{52} - 1$ mit einer Auflösung von 1 darstellen.
- Natürlich könnten wir auch eine andere Auflösung auswählen, sagen wir 0.001. Dann könnten wir Zahlen von 0 bis $0.001 * (2^{52} - 1)$ darstellen, und die kleinste Zahl größer als 0 wäre 0.001 anstelle von 1.
- Egal welche Auflösung wir auswählen würden, sie wäre gut in manchen Situationen und schlecht in anderen.
- Deshalb kommt der zweite Teil der Fließkommarepräsentation ins Spiel: Die 11 Bits des Exponenten speichern die Auflösung.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Natürlich könnten wir auch eine andere Auflösung auswählen, sagen wir 0.001. Dann könnten wir Zahlen von 0 bis $0.001 * (2^{52} - 1)$ darstellen, und die kleinste Zahl größer als 0 wäre 0.001 anstelle von 1.
- Egal welche Auflösung wir auswählen würden, sie wäre gut in manchen Situationen und schlecht in anderen.
- Deshalb kommt der zweite Teil der Fließkommarepräsentation ins Spiel: Die 11 Bits des Exponenten speichern die Auflösung.
- Der Exponent repräsentiert eine Potenz von 2, mit der der Signifikand multipliziert wird.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Egal welche Auflösung wir auswählen würden, sie wäre gut in manchen Situationen und schlecht in anderen.
- Deshalb kommt der zweite Teil der Fließkommarepräsentation ins Spiel: Die 11 Bits des Exponenten speichern die Auflösung.
- Der Exponent repräsentiert eine Potenz von 2, mit der der Signifikand multipliziert wird.
- Damit wir sowohl große als auch kleine Zahlen darstellen können, muss der Exponent sowohl positiv als auch negativ seien können.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Deshalb kommt der zweite Teil der Fließkommarepräsentation ins Spiel: Die 11 Bits des Exponenten speichern die Auflösung.
- Der Exponent repräsentiert eine Potenz von 2, mit der der Signifikand multipliziert wird.
- Damit wir sowohl große als auch kleine Zahlen darstellen können, muss der Exponent sowohl positiv als auch negativ seien können.
- Daher wird von dem Exponent immer ein konstanter Bias von 1023 abgezogen.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Der Exponent repräsentiert eine Potenz von 2, mit der der Signifikand multipliziert wird.
- Damit wir sowohl große als auch kleine Zahlen darstellen können, muss der Exponent sowohl positiv als auch negativ seien können.
- Daher wird von dem Exponent immer ein konstanter Bias von 1023 abgezogen.
- Ist z. B. 1050 im Exponenten-Feld gespeichert, dann ist der eigentliche Exponent $1050 - 1023 = 27$ und der Signifikand wird mit $2^{27} = 134\,217\,728$ multipliziert.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Damit wir sowohl große als auch kleine Zahlen darstellen können, muss der Exponent sowohl positiv als auch negativ seien können.
- Daher wird von dem Exponent immer ein konstanter Bias von 1023 abgezogen.
- Ist z. B. 1050 im Exponenten-Feld gespeichert, dann ist der eigentliche Exponent $1050 - 1023 = 27$ und der Signifikand wird mit $2^{27} = 134\,217\,728$ multipliziert.
- Das letzte Bit, das Sign-Bit, speichert das Vorzeichen der Zahl.



64 Bit IEEE Standard 754 Floating Point Number Layout



- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[45,48] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Daher wird von dem Exponent immer ein konstanter Bias von 1023 abgezogen.
- Ist z. B. 1050 im Exponenten-Feld gespeichert, dann ist der eigentliche Exponent $1050 - 1023 = 27$ und der Signifikand wird mit $2^{27} = 134\,217\,728$ multipliziert.
- Das letzte Bit, das Sign-Bit, speichert das Vorzeichen der Zahl.
- Wir können somit Zahlen von $2.225\,073\,858\,507\,201\,4 * 10^{-308}$ bis $1.797\,693\,134\,8623\,157 * 10^{308}$ mit einer Auflösung von etwa 15 Ziffern speichern, zusammen mit 0, und dem selben Wertebereich im Negativen, und einige Sonderwerte.



Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.
  3. Sie können Ganzzahlen und gebrochene Zahlen darstellen.

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.
  3. Sie können Ganzzahlen und gebrochene Zahlen darstellen.
  4. Ihre Genauigkeit ist jedoch auf 15 bis 16 Ziffern begrenzt.

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.
  3. Sie können Ganzzahlen und gebrochene Zahlen darstellen.
  4. Ihre Genauigkeit ist jedoch auf 15 bis 16 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Ergebnis immer noch 10^{16}, weil nur 15 Ziffern „gespeichert“ werden und die 1 einfach „herunterfällt.“

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.
  3. Sie können Ganzzahlen und gebrochene Zahlen darstellen.
  4. Ihre Genauigkeit ist jedoch auf 15 bis 16 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Ergebnis immer noch 10^{16}, weil nur 15 Ziffern „gespeichert“ werden und die 1 einfach „herunterfällt.“
- Zahlen können mit `floats` nicht beliebig genau gespeichert werden[38].

Das Wichtige



- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.
  3. Sie können Ganzzahlen und gebrochene Zahlen darstellen.
  4. Ihre Genauigkeit ist jedoch auf 15 bis 16 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Ergebnis immer noch 10^{16}, weil nur 15 Ziffern „gespeichert“ werden und die 1 einfach „herunterfällt.“
- Zahlen können mit `floats` nicht beliebig genau gespeichert werden[38].
- Ansonsten sind `floats` aber schon ziemlich cool.

Das Wichtige



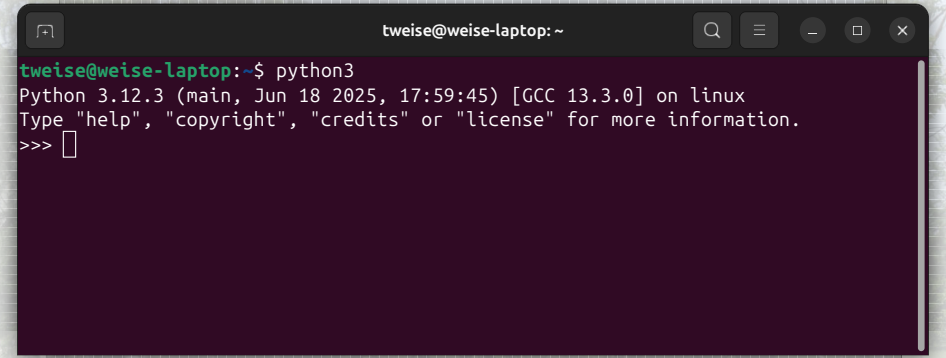
- Zum Glück muss man all das nicht wirklich wissen, wenn man nur normal vor sich hin programmiert.
- Die wichtigen Informationen, die Sie sich merken müssen, sind jedoch:
  1. Fließkommazahlen (`floats`) können eine große Vielzahl verschiedener Werte repräsentieren.
  2. Ihr Wertebereich ist groß, aber begrenzt.
  3. Sie können Ganzzahlen und gebrochene Zahlen darstellen.
  4. Ihre Genauigkeit ist jedoch auf 15 bis 16 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Ergebnis immer noch 10^{16}, weil nur 15 Ziffern „gespeichert“ werden und die 1 einfach „herunterfällt.“
- Zahlen können mit `floats` nicht beliebig genau gespeichert werden[38].
- Ansonsten sind `floats` aber schon ziemlich cool.
- Hinweis: Neben Python's `floats` gibt es viele andere Fließkommaformate mit verschiedenen Auflösungen[39,48].

Rechnen mit float



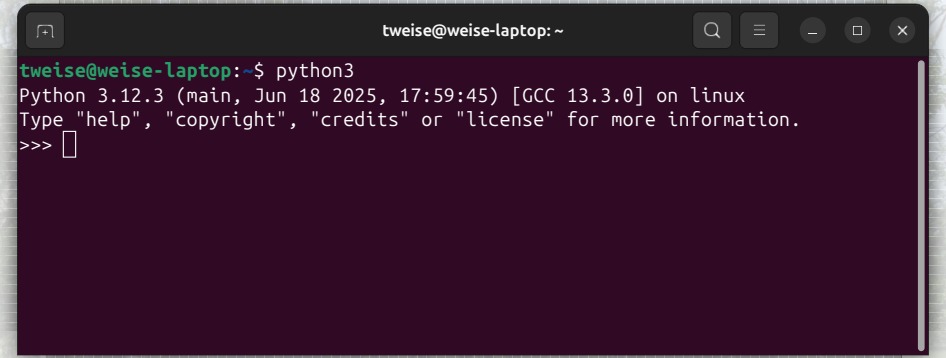
Rechnen mit float

- Lassen Sie uns nun mit `floats` rechnen.



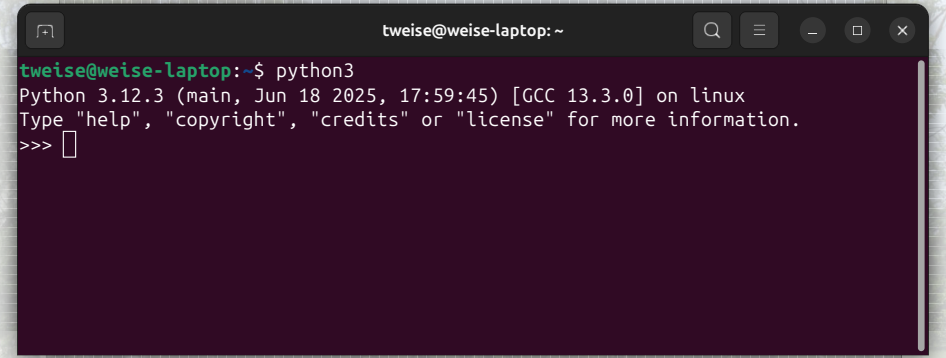
Rechnen mit float

- Lassen Sie uns nun mit `floats` rechnen.
- Fließkommazahlen werden immer mit dem Dezimalpunkt aufgeschrieben.



Rechnen mit float

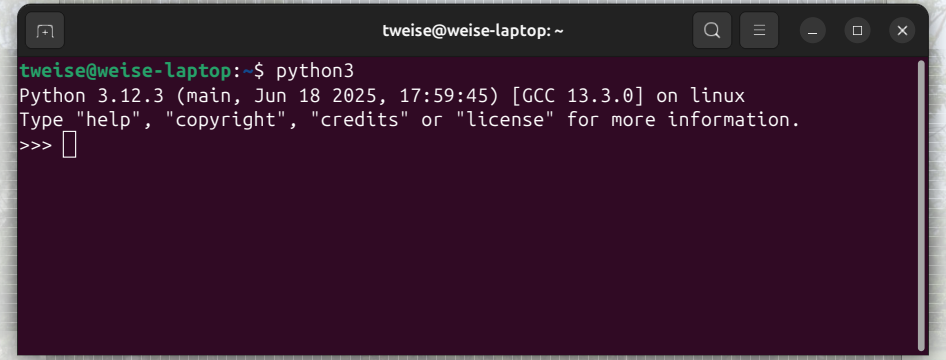
- Lassen Sie uns nun mit `floats` rechnen.
- Fließkommazahlen werden immer mit dem Dezimalpunkt aufgeschrieben. So ist `3.0` ein `float`, `3` aber ein `int`.



Rechnen mit float

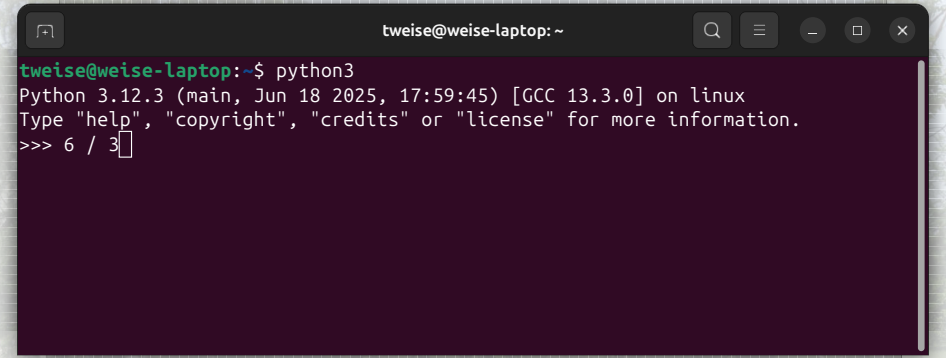


- Fließkommazahlen werden immer mit dem Dezimalpunkt aufgeschrieben. So ist `3.0` ein `float`, `3` aber ein `int`.
- Beachten Sie: Kommt in einer Rechnung mit `ints` irgendwo ein `float` vor, dann ist das Ergebnis immer ein `float`.



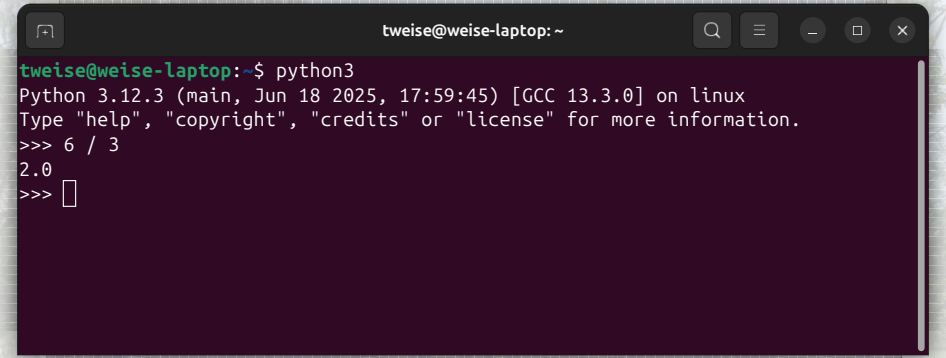
Rechnen mit float

- Die Fließkommadivision `6 / 3`...



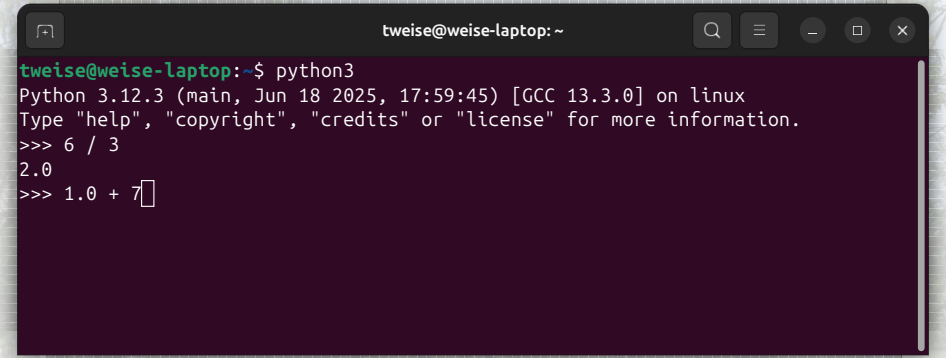
Rechnen mit float

- Die Fließkommadivision `6 / 3` ergibt den `float` `2.0`.



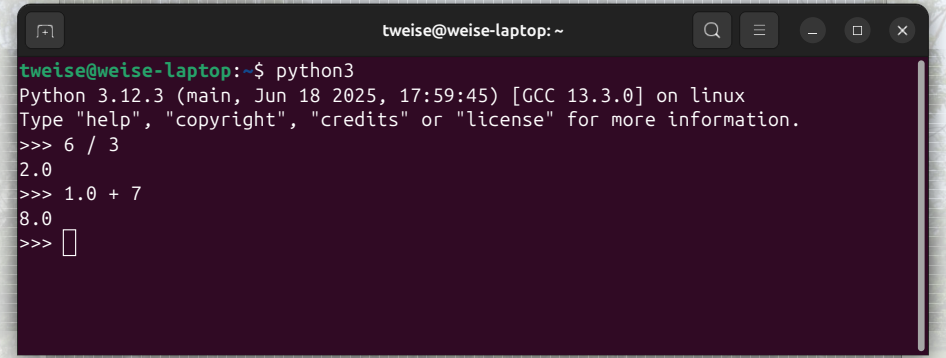
Rechnen mit float

- Addieren wir den `float` `1.0` zum `int` `7`...



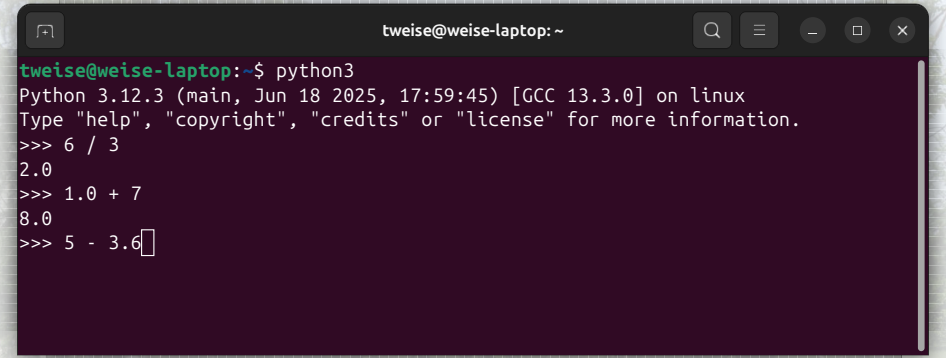
Rechnen mit float

- Addieren wir den `float` `1.0` zum `int` `7` ergibt das den `float` `8.0`.



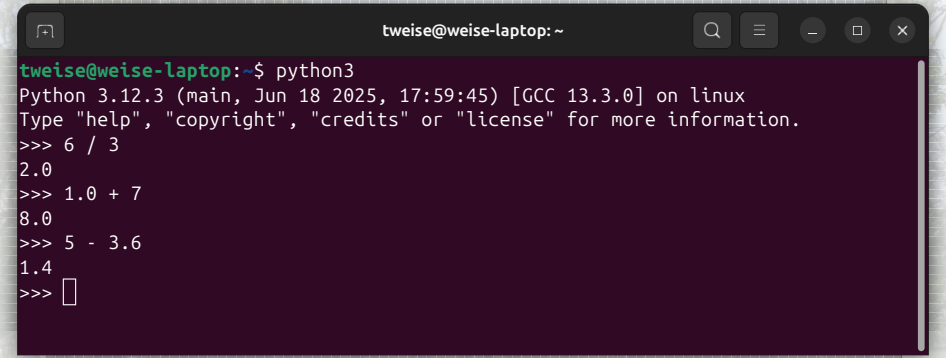
Rechnen mit float

- Subtrahieren wir den `float` `3.6` vom `int` `5`...



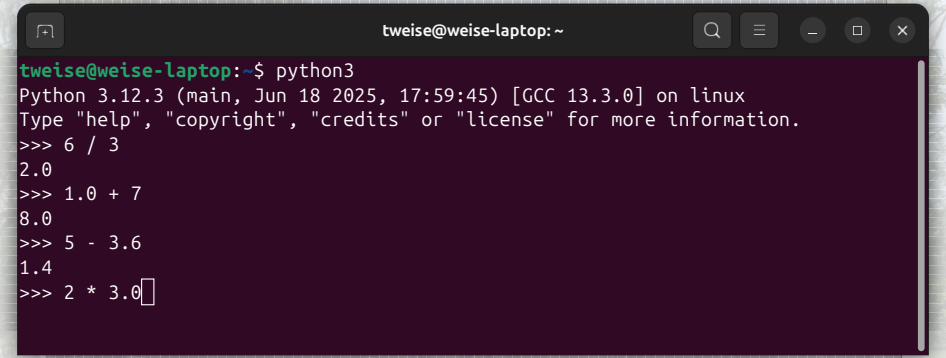
Rechnen mit float

- Subtrahieren wir den `float` `3.6` vom `int` `5` ergibt das den `float` `1.4`.



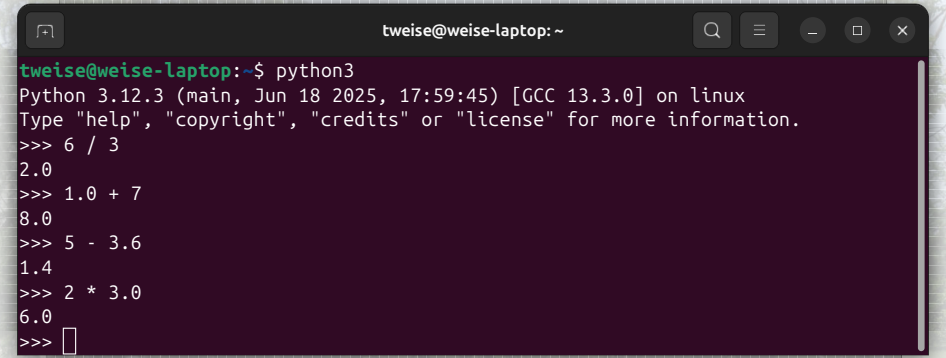
Rechnen mit float

- Multiplizieren wir den `int` `2` mit dem `float` `3`...



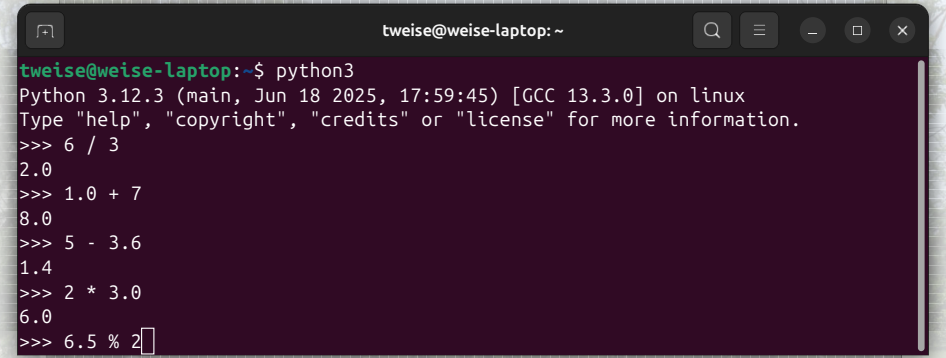
Rechnen mit float

- Multiplizieren wir den `int` `2` mit dem `float` `3` ergibt das den `float` `6.0`.



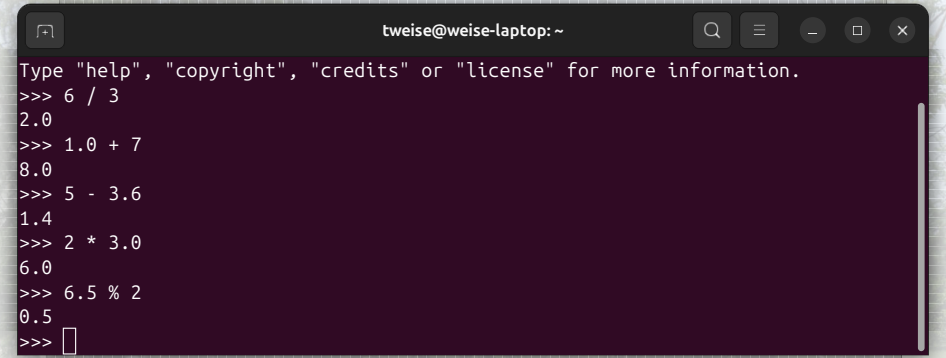
Rechnen mit float

- Berechnen wir den Rest der Division des `float` `6.5` durch den `int` `2`...



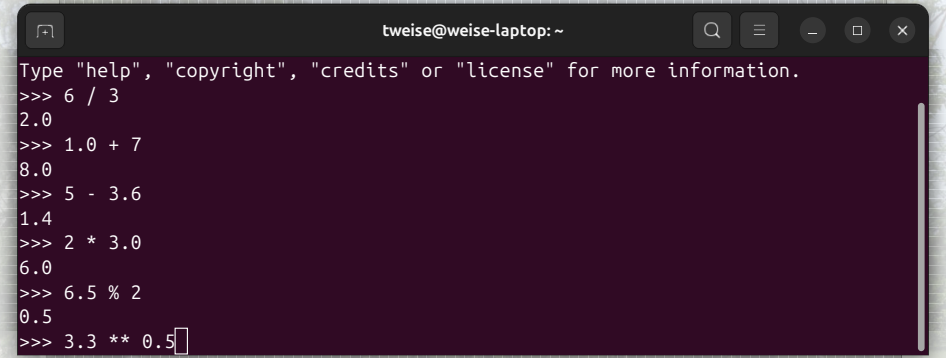
Rechnen mit float

- Berechnen wir den Rest der Division des `float` `6.5` durch den `int` `2` ergibt das den `float` `0.5`.



Rechnen mit float

- Berechnen wir $3.3^{0.5} \equiv \sqrt{3.3}\ldots$



Rechnen mit float

- Berechnen wir $3.3^{0.5} \equiv \sqrt{3.3}$ ergibt das Näherungswert `1.816590212458495`, der auf etwa 15 Stellen genau ist.



Rechnen mit float

- Auch große Berechnungen mit Klammern funktionieren einwandfrei.



Rechnen mit float

- Auch große Berechnungen mit Klammern funktionieren einwandfrei.



Rechnen mit float

- Wollen wir mathematische Funktionen und Konstanten nutzen, müssen wir diese aus dem Modul `math` importieren.



Rechnen mit float

- Wollen wir mathematische Funktionen und Konstanten nutzen, müssen wir diese aus dem Modul `math` importieren.
- Wir lernen später, was das genau bedeutet.



Rechnen mit float

- Wir lernen später, was das genau bedeutet.
- Für jetzt, nehmen wir die Syntax `from math import XXX` an, wenn wir `XXX` nutzen wollen.



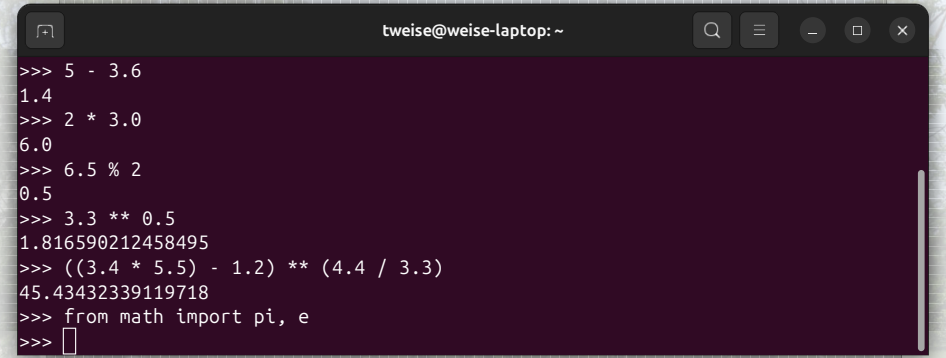
Rechnen mit float

- Für jetzt, nehmen wir die Syntax `from math import XXX` an, wenn wir `XXX` nutzen wollen.
- Wollen wir die Konstanten π und e nutzen, schreiben wir also `from math import pi, e`.



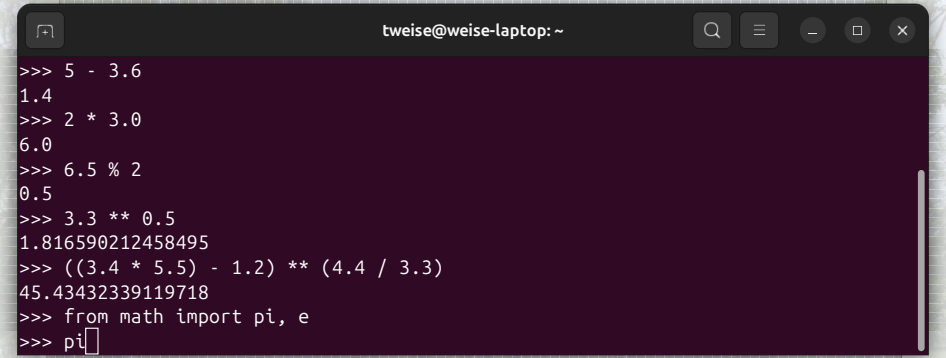
Rechnen mit float

- Wollen wir die Konstanten π und e nutzen, schreiben wir also `from math import pi, e`.



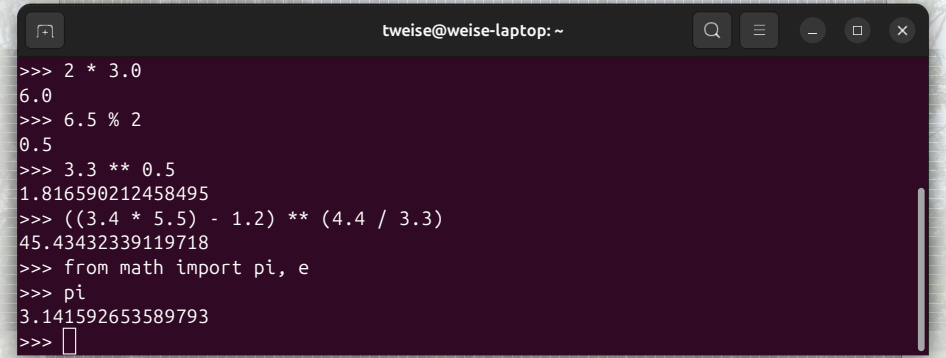
Rechnen mit float

- Die Konstante π, in Python `math.pi`...



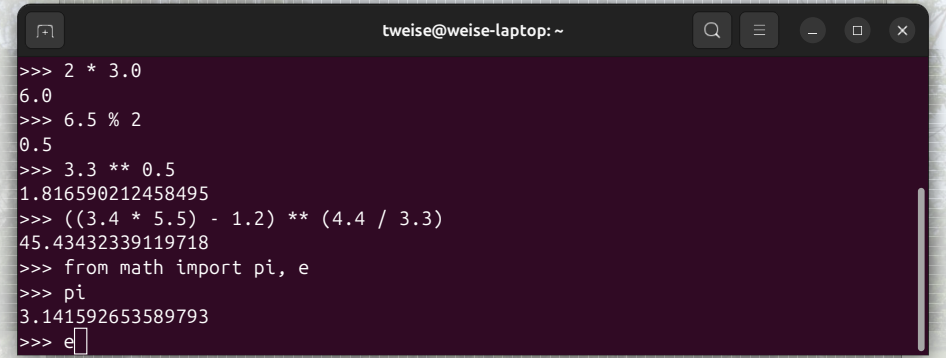
Rechnen mit float

- Die Konstante π, in Python `math.pi` hat den Näherungswert `3.141592653589793`, der auf etwa 15 Stellen genau ist.



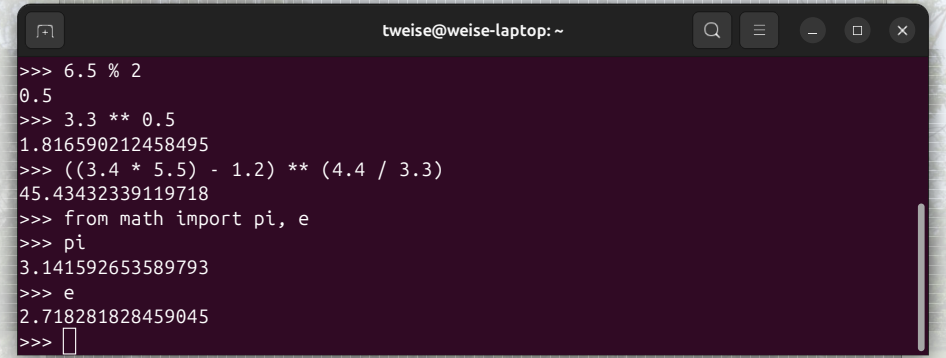
Rechnen mit float

- Die Konstante e, in Python `math.e`...



Rechnen mit float

- Die Konstante e, in Python `math.e` hat den Näherungswert `2.718281828459045`, der auf etwa 15 Stellen genau ist.



Rechnen mit float

- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.

Rechnen mit float

- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht.

Rechnen mit float

- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht. Wir müssen immer irgendwo abbrechen.

Rechnen mit float

- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht. Wir müssen immer irgendwo abbrechen.
- Wir könnten z. B. 0.14285714285714285 schreiben ... aber das ist nicht das Gleiche wie $\frac{1}{7}$.

Rechnen mit float

- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht. Wir müssen immer irgendwo abbrechen.
- Wir könnten z. B. 0.14285714285714285 schreiben ... aber das ist nicht das Gleiche wie $\frac{1}{7}$.
- Im binären Format sehen wir das Problem schon bei Zahlen wie $\frac{1}{10} = 0.1$[38].

Rechnen mit float

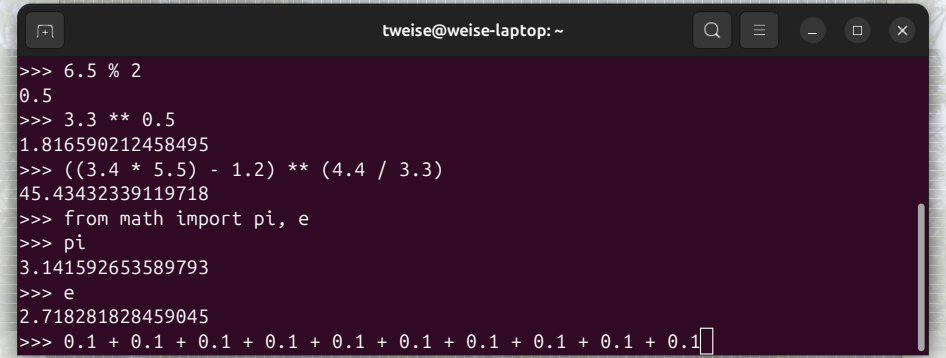
- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht. Wir müssen immer irgendwo abbrechen.
- Wir könnten z. B. 0.14285714285714285 schreiben ... aber das ist nicht das Gleiche wie $\frac{1}{7}$.
- Im binären Format sehen wir das Problem schon bei Zahlen wie $\frac{1}{10} = 0.1^{38}$.
- Wir könnten schreiben $\frac{1}{10} \approx \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{16}} + \ldots$, aber wir würden niemals genau auf 0.1 kommen.

Rechnen mit float

- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht. Wir müssen immer irgendwo abbrechen.
- Wir könnten z. B. 0.14285714285714285 schreiben ... aber das ist nicht das Gleiche wie $\frac{1}{7}$.
- Im binären Format sehen wir das Problem schon bei Zahlen wie $\frac{1}{10} = 0.1^{38}$.
- Wir könnten schreiben $\frac{1}{10} \approx \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{16}} + \ldots$, aber wir würden niemals genau auf 0.1 kommen.
- Das bedeutet, das 0.1 nicht im genau im `float`-Format dargestellt werden kann.

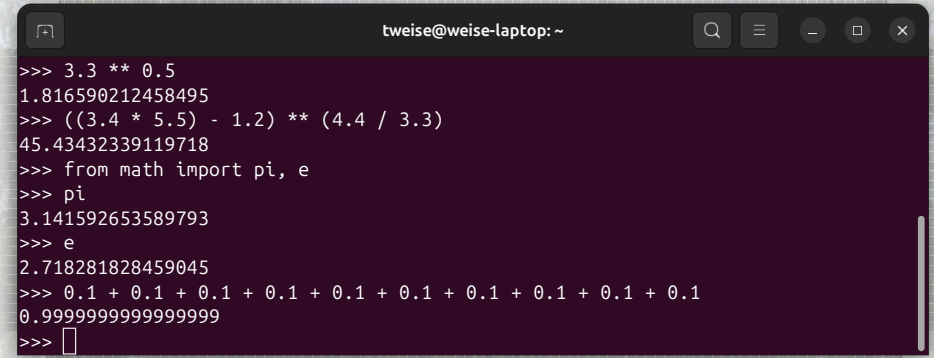
Rechnen mit float

- Aber was bedeutet wiederum das? Addieren wir „zehn `0.1`“ zusammen...



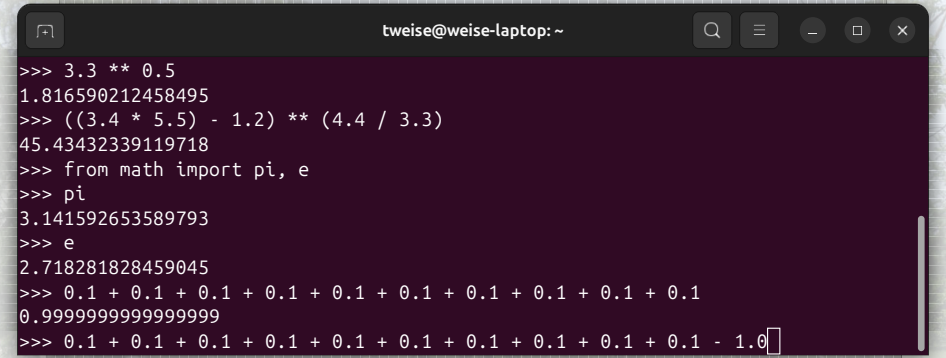
Rechnen mit float

- Aber was bedeutet wiederum das? Addieren wir „zehn `0.1`“ zusammen, dann liegt das Ergebnis ganz knapp neben 1!



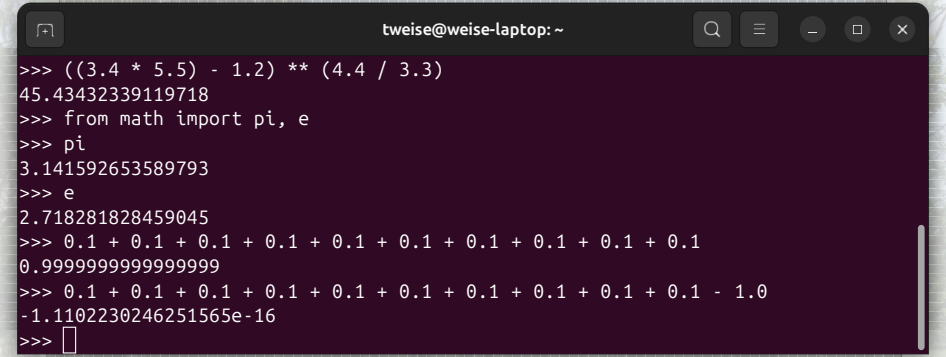
Rechnen mit float

- Ziehen wir `1.0` von „zehn `0.1`“ ab



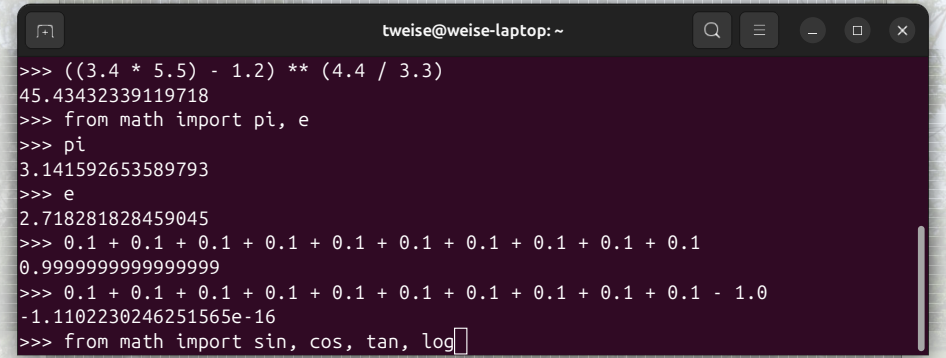
Rechnen mit float

- Ziehen wir `1.0` von „zehn `0.1`“ ab, dann ist das Ergebnis eben nur fast 0!



Rechnen mit float

- Anderes Thema: Python hat natürlich auch trigonometrische und logarithmische Funktionen.



Rechnen mit float

- Anderes Thema: Python hat natürlich auch trigonometrische und logarithmische Funktionen, die wir natürlich wieder aus `math` importieren.



Rechnen mit float

- Aus der Schule wissen wir, dass $\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$ und das darum $\sin^2\left(\frac{\pi}{4}\right) = \frac{1}{2}$, also berechnen wir `sin(0.25 * pi)** 2`...



Rechnen mit float

- Aus der Schule wissen wir, dass $\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$ und das darum $\sin^2\left(\frac{\pi}{4}\right) = \frac{1}{2}$, also berechnen wir `sin(0.25 * pi)** 2` und das Ergebnis kommt auf etwa 15 Stellen genau hin...



Rechnen mit float

- Aus der Schule wissen wir, dass $\cos\left(\frac{\pi}{3}\right) = \frac{1}{2}$, also berechnen wir `cos(pi / 3)` ...



Rechnen mit float

- Aus der Schule wissen wir, dass $\cos\left(\frac{\pi}{3}\right) = \frac{1}{2}$, also berechnen wir `cos(pi / 3)` und das Ergebnis kommt auf etwa 15 Stellen genau hin...



Rechnen mit float

- Aus der Schule wissen wir, dass $\tan\left(\frac{\pi}{4}\right) = 1$, also berechnen wir `tan(pi / 4)`...



Rechnen mit float

- Aus der Schule wissen wir, dass $\tan\left(\frac{\pi}{4}\right) = 1$, also berechnen wir `tan(pi / 4)` und das Ergebnis kommt auf etwa 15 Stellen genau hin...



Rechnen mit float

- Aus der Schule wissen wir, dass $\log\left(e^{10}\right) = 10$, also berechnen wir `log(e ** 10)`...



Rechnen mit float

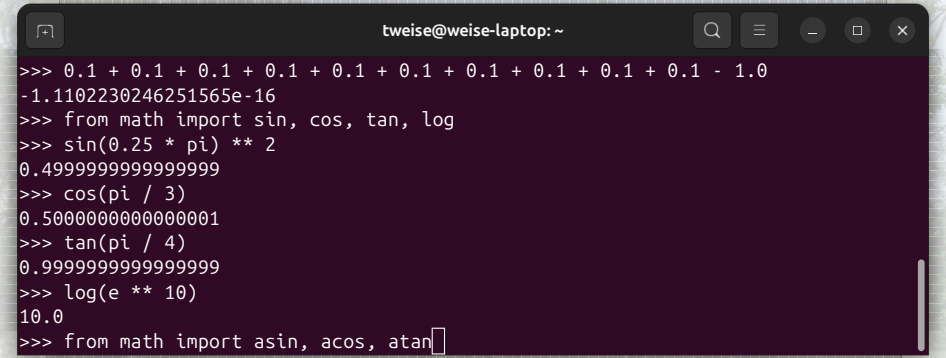


- Aus der Schule wissen wir, dass $\log\left(e^{10}\right) = 10$, also berechnen wir `log(e ** 10)` und das Ergebnis stimmt genau.

```
tweise@weise-laptop: ~
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 - 1.0
-1.1102230246251565e-16
>>> from math import sin, cos, tan, log
>>> sin(0.25 * pi) ** 2
0.4999999999999999
>>> cos(pi / 3)
0.5000000000000001
>>> tan(pi / 4)
0.9999999999999999
>>> log(e ** 10)
10.0
>>>
```

Rechnen mit float

- Python hat natürlich auch die Umkehrfunktionen der trigonometrischen Funktionen.



Rechnen mit float

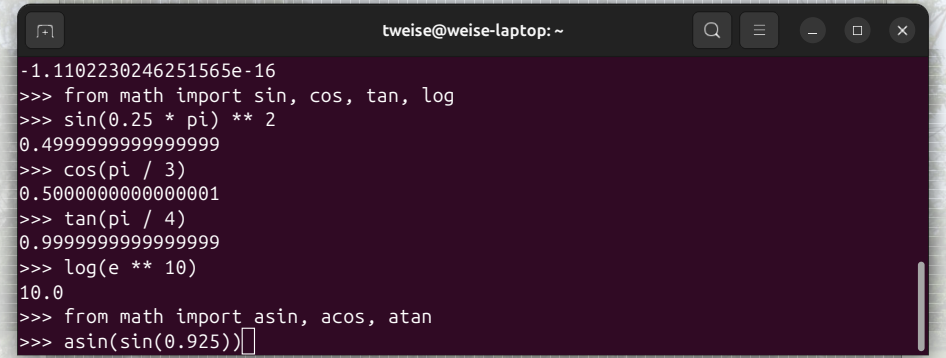


- Python hat natürlich auch die Umkehrfunktionen der trigonometrischen Funktionen, die wir natürlich wieder aus `math` importieren.

```
tweise@weise-laptop: ~
-1.1102230246251565e-16
>>> from math import sin, cos, tan, log
>>> sin(0.25 * pi) ** 2
0.4999999999999999
>>> cos(pi / 3)
0.5000000000000001
>>> tan(pi / 4)
0.9999999999999999
>>> log(e ** 10)
10.0
>>> from math import asin, acos, atan
>>>
```

Rechnen mit float

- Aus der Schule wissen wir, dass $\arcsin \sin x = x \; \forall x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, also berechnen wir `asin(sin(0.925))`...



Rechnen mit float

- Aus der Schule wissen wir, dass $\arcsin \sin x = x \; \forall x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, also berechnen wir `asin(sin(0.925))` und das Ergebnis stimmt genau.



Rechnen mit float

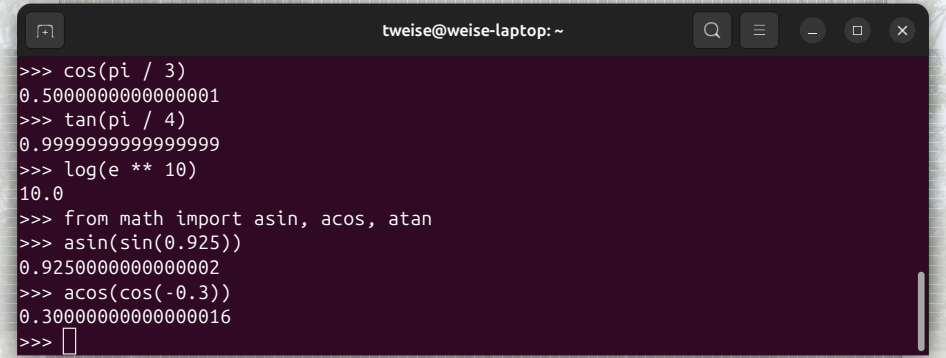


- Aus der Schule wissen wir, dass $\arccos \cos x = x \; \forall x \in [0, \pi]$, also berechnen wir `acos(cos(-0.3))`, wobei -0.3 außerhalb des Intervalls liegt...

```
tweise@weise-laptop: ~
>>> sin(0.25 * pi) ** 2
0.4999999999999999
>>> cos(pi / 3)
0.5000000000000001
>>> tan(pi / 4)
0.9999999999999999
>>> log(e ** 10)
10.0
>>> from math import asin, acos, atan
>>> asin(sin(0.925))
0.9250000000000002
>>> acos(cos(-0.3))
```

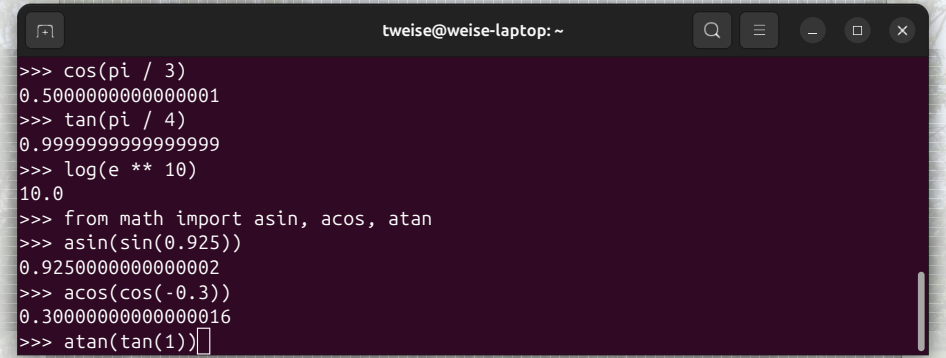
Rechnen mit float

- Aus der Schule wissen wir, dass $\arccos \cos x = x \; \forall x \in [0, \pi]$, also berechnen wir `acos(cos(-0.3))`, wobei -0.3 außerhalb des Intervalls liegt; und das Ergebnis kommt auf etwa 15 Stellen genau hin...



Rechnen mit float

- Aus der Schule wissen wir, dass $\arctan \tan x = x \; \forall x \in (-\frac{\pi}{2}, \frac{\pi}{2})$, also berechnen wir `atan(tan(1))`...



Rechnen mit float

- Aus der Schule wissen wir, dass $\arctan \tan x = x \; \forall x \in (-\frac{\pi}{2}, \frac{\pi}{2})$, also berechnen wir `atan(tan(1))` und das Ergebnis stimmt genau.



Rechnen mit float



- Damit haben wir erstmal einige grundlegende Rechenarten abgearbeitet.

Fließkommazahlen sind ungenau



Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[6,71].

Fließkommazahlen sind ungenau



Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[6,71].

- Wegen der begrenzten Präzision kann es sein, dass Sie zwei Zahlen addieren $a + b = c$ aber dann feststellen dass $c - a \neq b$.

Fließkommazahlen sind ungenau



Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[6,71].

- Wegen der begrenzten Präzision kann es sein, dass Sie zwei Zahlen addieren $a + b = c$ aber dann feststellen dass $c - a \neq b$.
- Das ist offensichtlich, wenn wir sehr kleine Zahlen zu sehr großen Zahlen addieren.

Fließkommazahlen sind ungenau



Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[6,71].

- Wegen der begrenzten Präzision kann es sein, dass Sie zwei Zahlen addieren $a + b = c$ aber dann feststellen dass $c - a \neq b$.
- Das ist offensichtlich, wenn wir sehr kleine Zahlen zu sehr großen Zahlen addieren: Wir haben nur 15 Ziffern, also wenn wir $10^{20} + 1$ rechnen, dann kommt mit Fließkommaarithmentik wieder 10^{20} heraus[71].

Fließkommazahlen sind ungenau



Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[6,71].

- Wegen der begrenzten Präzision kann es sein, dass Sie zwei Zahlen addieren $a + b = c$ aber dann feststellen dass $c - a \neq b$.
- Das ist offensichtlich, wenn wir sehr kleine Zahlen zu sehr großen Zahlen addieren: Wir haben nur 15 Ziffern, also wenn wir $10^{20} + 1$ rechnen, dann kommt mit Fließkommaarithmentik wieder 10^{20} heraus[71].
- Ziffern können aber auch verloren gehen, wenn wir Zahlen der selben Größenordnung addieren, denn auch deren Summe kann breiter als das 15-Ziffern-Fenster sein.

Fließkommazahlen sind ungenau



Gute Praxis

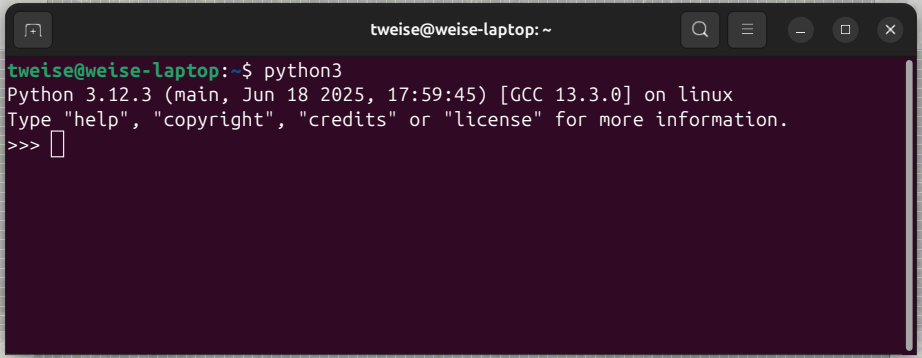
Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[6,71].

- Wegen der begrenzten Präzision kann es sein, dass Sie zwei Zahlen addieren $a + b = c$ aber dann feststellen dass $c - a \neq b$.
- Das ist offensichtlich, wenn wir sehr kleine Zahlen zu sehr großen Zahlen addieren: Wir haben nur 15 Ziffern, also wenn wir $10^{20} + 1$ rechnen, dann kommt mit Fließkommaarithmentik wieder 10^{20} heraus[71].
- Ziffern können aber auch verloren gehen, wenn wir Zahlen der selben Größenordnung addieren, denn auch deren Summe kann breiter als das 15-Ziffern-Fenster sein. Dann fällt die niedrigwertigste Ziffer einfach weg[6]...

Runden

Runden

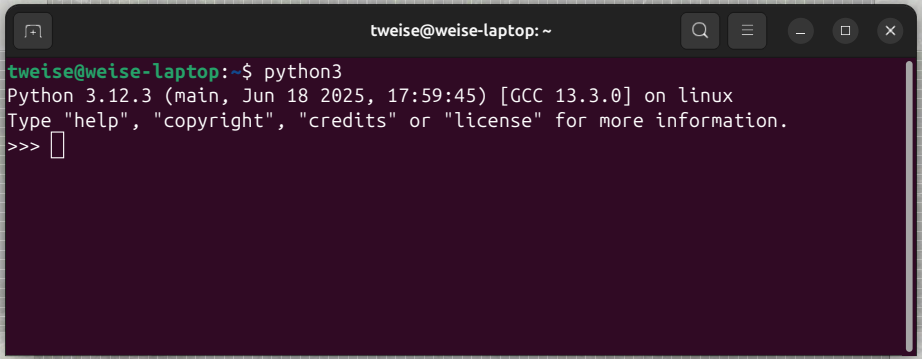
- Der Weg von `floats` zurück zu `ints` geht über runden.



Runden

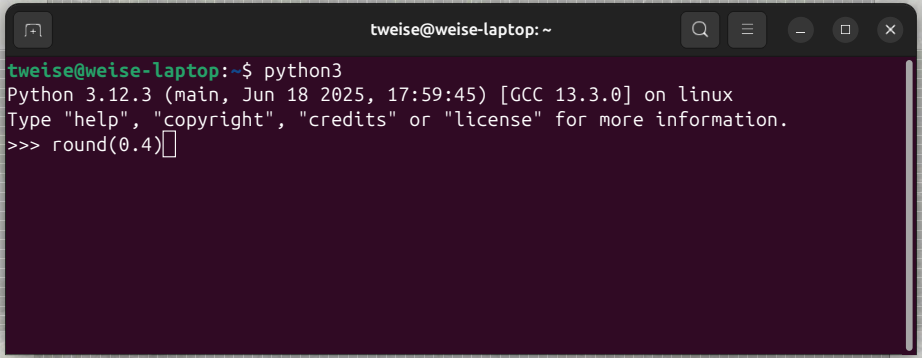


- Der Weg von `floats` zurück zu `ints` geht über runden.
- Python bietet uns mehrere verschiedene Funktionen zum Runden an.



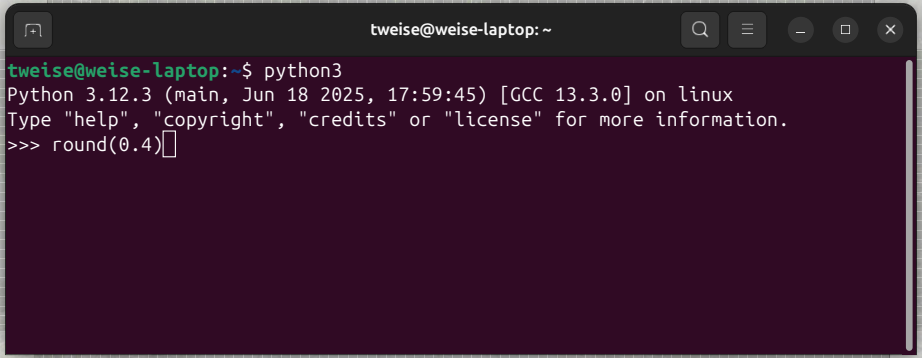
Runden

- Die Funktion `round(x)` führt sogenanntes „banker's rounding“ durch.



Runden

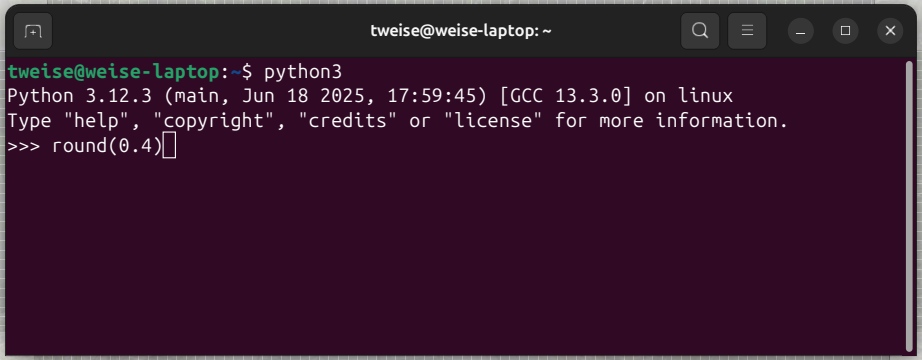
- Die Funktion `round(x)` führt sogenanntes „banker's rounding“ durch.
- Dabei wird `x` zur nächsten Ganzzahl gerundet.



Runden



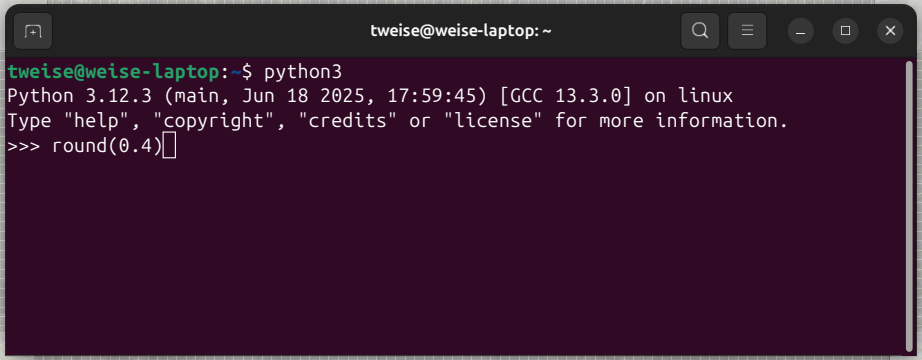
- Die Funktion `round(x)` führt sogenanntes „banker's rounding“ durch.
- Dabei wird `x` zur nächsten Ganzzahl gerundet.
- Sind zwei Ganzzahlen gleichweit entfernt, also wenn `x` in der Form `z.5` ist, dann wird zur der Zahl gerundet, die gerade ist.[15]



Runden



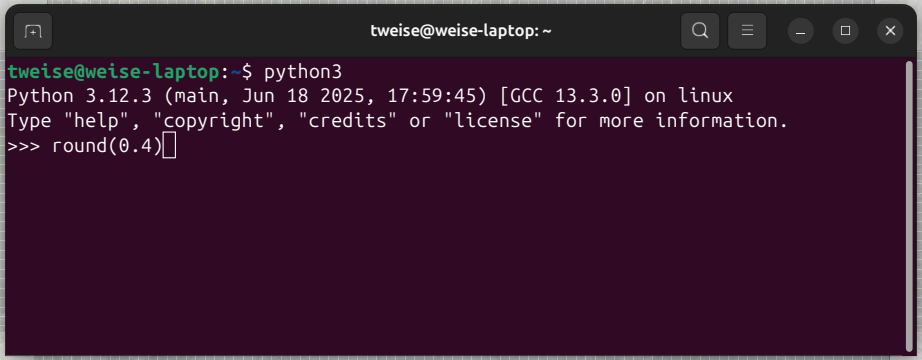
- Die Funktion `round(x)` führt sogenanntes „banker's rounding“ durch.
- Sind zwei Ganzzahlen gleichweit entfernt, also wenn `x` in der Form `z.5` ist, dann wird zur der Zahl gerundet, die gerade ist.[15]
- In der Schule hatte ich gelernt, dass `z.5` zu `z + 1` gerundet wird. Java's `Math.round` macht das auch so[50] ... aber Python's `round` macht das nicht!



Runden



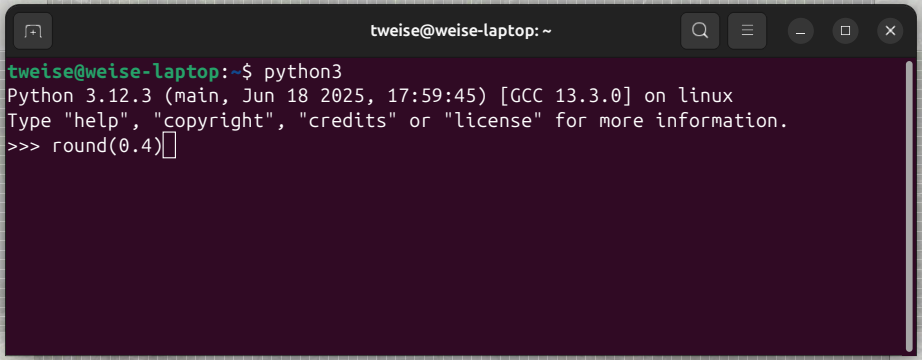
- Die Funktion `round(x)` führt sogenanntes „banker's rounding“ durch.
- Banker's rounding hat den Vorteil, dass es keinen Bias auslöst: Rundet man `z.5` immer zu `z + 1` und es gibt viele Zahlen der Form `z.5`, dann tendiert der Durchschnitt der gerundeten Zahlen dazu, größer als der Durchschnitt der ungerundeten Zahlen zu sein. Beim banker's rounding ist das nicht so[46,60].



Runden

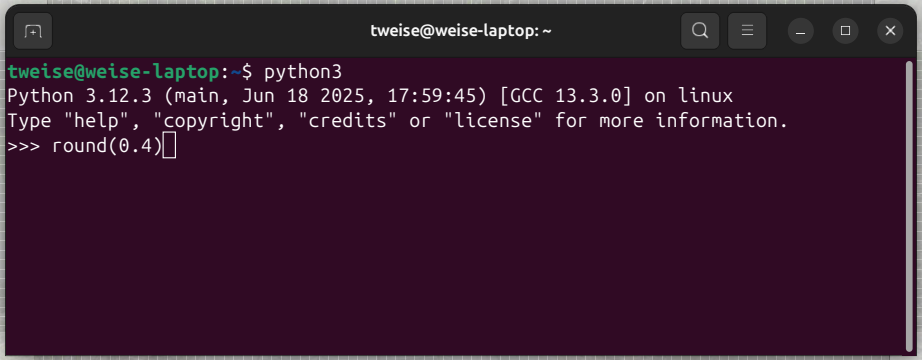


- Die Funktion `round(x)` führt sogenanntes „banker's rounding“ durch.
- Deshalb ist es in Python und im IEEE 754 Standard[48] bevorzugt und wird z. B. auch von Alipay+ genutzt[3].



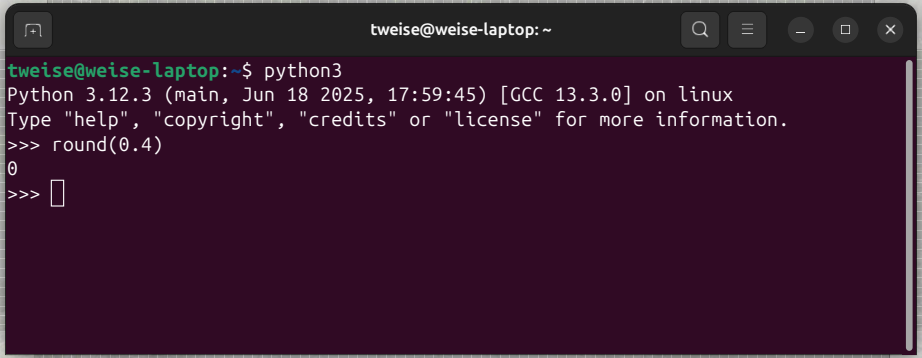
Runden

- Deshalb ist es in Python und im IEEE 754 Standard[48] bevorzugt und wird z. B. auch von Alipay+ genutzt[3].
- Wenn wir `round(0.4)` aufrufen...



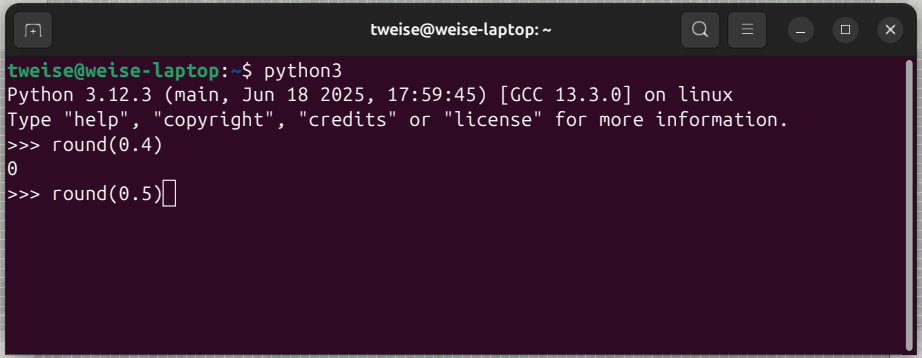
Runden

- Wenn wir `round(0.4)` aufrufen, dann kommt `0` heraus.



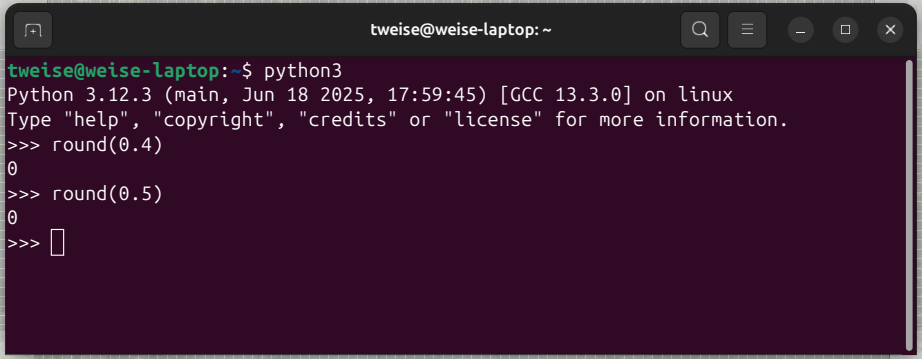
Runden

- Wenn wir `round(0.5)` aufrufen



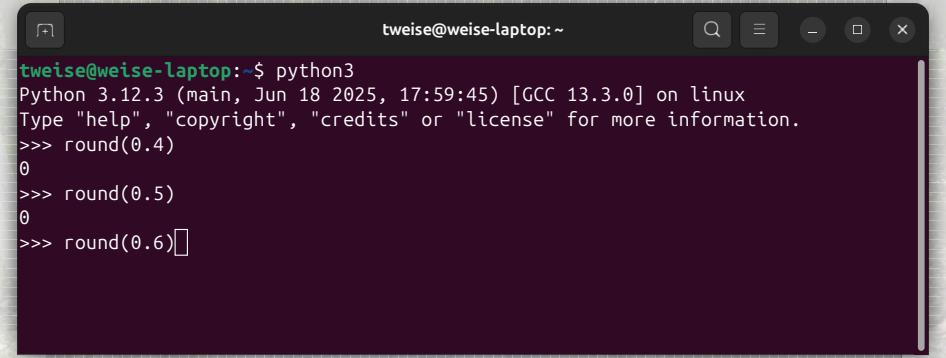
Runden

- Wenn wir `round(0.5)` aufrufen, dann kommt auch `0` heraus, weil 0 und 1 gleichweit von 0.5 entfernt sind, aber 0 gerade ist.



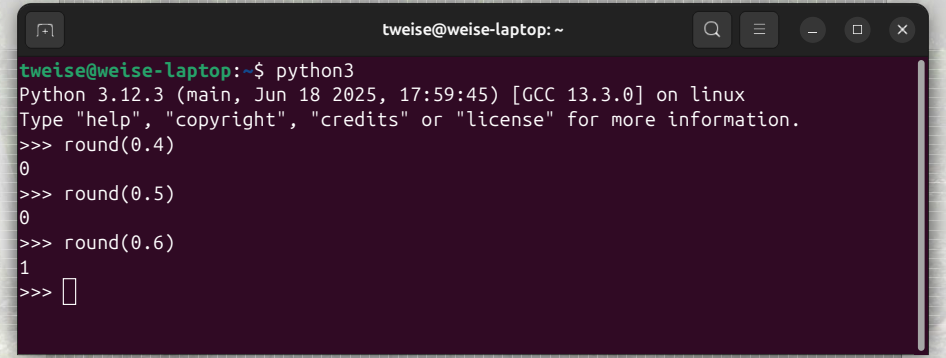
Runden

- Wenn wir `round(0.6)` aufrufen...



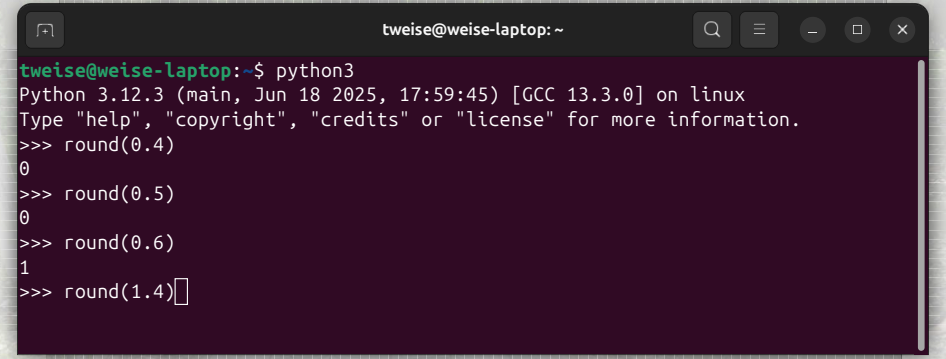
Runden

- Wenn wir `round(0.6)` aufrufen, dann kommt `1` heraus.



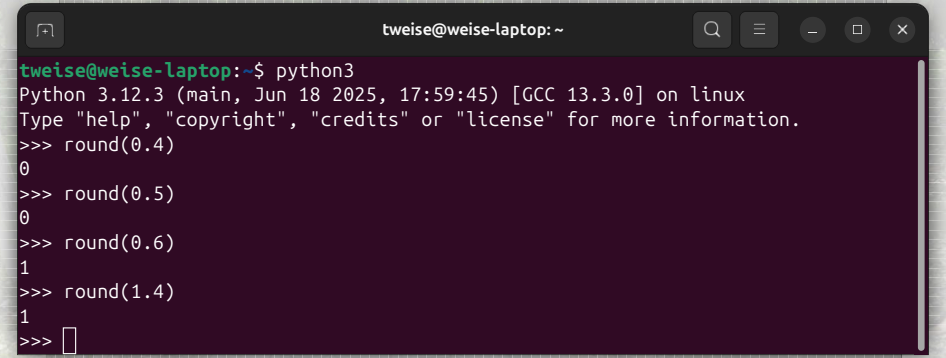
Runden

- Wenn wir `round(1.4)` aufrufen...



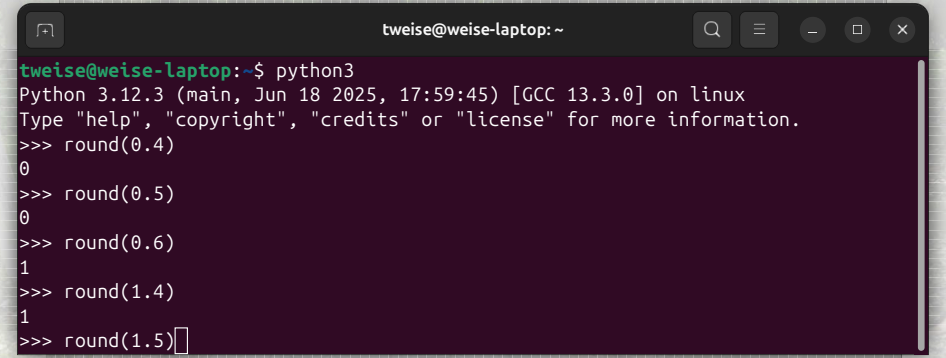
Runden

- Wenn wir `round(1.4)` aufrufen, dann kommt `1` heraus.



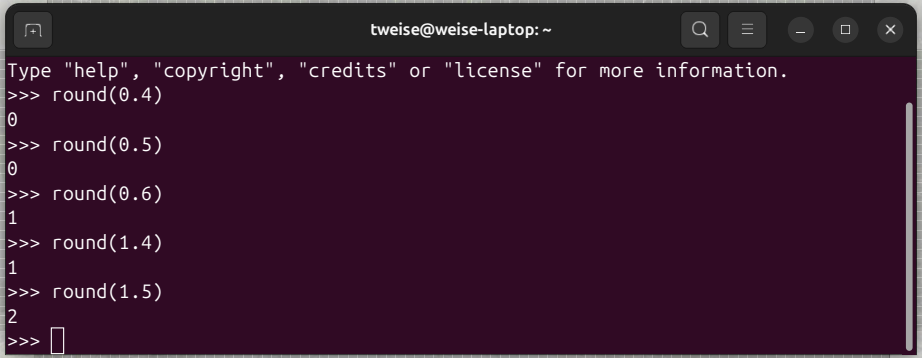
Runden

- Wenn wir `round(1.5)` aufrufen...



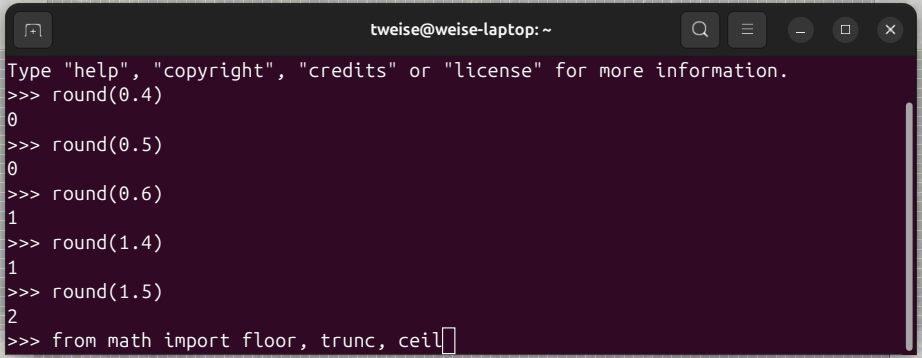
Runden

- Wenn wir `round(1.5)` aufrufen, dann kommt `2` heraus, weil 1 und 2 gleichweit von 1.5 entfernt sind, 2 aber gerade ist.



Runden

- Das Modul `math` stellt die Funktionen `floor` (Abrunden), `ceil` (Aufrunden), und `trunc` (Nachkommastellen abschneiden) zur Verfügung.



Runden

- Das Modul `math` stellt die Funktionen `floor` (Abrunden), `ceil` (Aufrunden), und `trunc` (Nachkommastellen abschneiden) zur Verfügung.
- Wir importieren sie also.



Runden

- Wenn wir `floor(0.4)` aufrufen...



Runden



- Wenn wir `floor(0.4)` aufrufen, dann kommt `0` heraus, weil wir abrunden.



Runden

- Wenn wir `floor(0.5)` aufrufen...



Runden



- Wenn wir `floor(0.5)` aufrufen, dann kommt `0` heraus, weil wir abrunden.



Runden

- Wenn wir `floor(0.6)` aufrufen...



Runden



- Wenn wir `floor(0.6)` aufrufen, dann kommt `0` heraus, weil wir abrunden.



Runden

- Wenn wir `floor(-0.4)` aufrufen...



Runden



- Wenn wir `floor(-0.4)` aufrufen, dann kommt `-1` heraus, weil wir abrunden.

```
tweise@weise-laptop: ~
>>> round(1.5)
2
>>> from math import floor, trunc, ceil
>>> floor(0.4)
0
>>> floor(0.5)
0
>>> floor(0.6)
0
>>> floor(-0.4)
-1
>>>
```

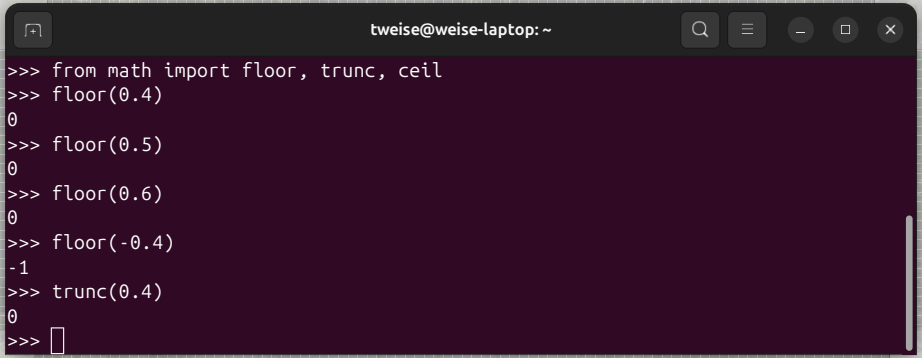
Runden

- Wenn wir `trunc(0.4)` aufrufen...



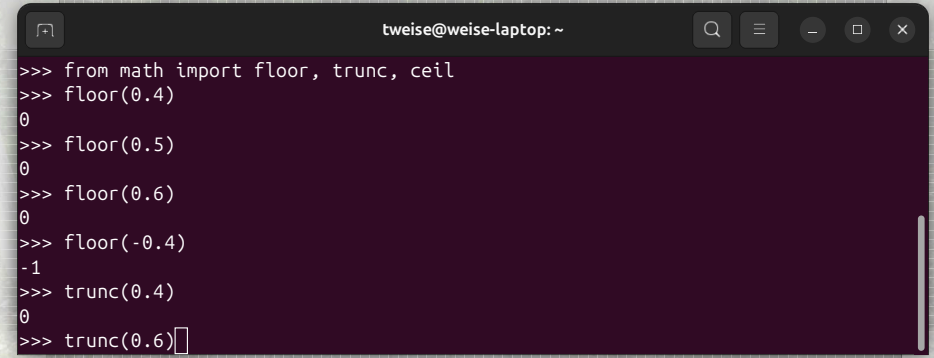
Runden

- Wenn wir `trunc(0.4)` aufrufen, dann kommt `0` heraus, weil wir die Nachkommastellen abschneiden.



Runden

- Wenn wir `trunc(0.6)` aufrufen...



Runden

- Wenn wir `trunc(0.6)` aufrufen, dann kommt `0` heraus, weil wir die Nachkommastellen abschneiden.



Runden

- Wenn wir `trunc(-0.4)` aufrufen...



Runden

- Wenn wir `trunc(-0.4)` aufrufen, dann kommt `0` heraus, weil wir die Nachkommastellen abschneiden.



Runden

- Wenn wir `trunc(-0.6)` aufrufen...



Runden

- Wenn wir `trunc(-0.6)` aufrufen, dann kommt `0` heraus, weil wir die Nachkommastellen abschneiden.



Runden

- Wenn wir `ceil(0.4)` aufrufen...



Runden



- Wenn wir `ceil(0.4)` aufrufen, dann kommt `1` heraus, weil aufrunden.

tweise@weise-laptop: ~

```
-1
>>> trunc(0.4)
0
>>> trunc(0.6)
0
>>> trunc(-0.4)
0
>>> trunc(-0.6)
0
>>> ceil(0.4)
1
>>>
```

Runden

- Wenn wir `ceil(-11.1)` aufrufen...



Runden

- Wenn wir `ceil(-11.1)` aufrufen, dann kommt `-11` heraus, weil aufrunden.



Runden

- Wenn wir `ceil(-11.6)` aufrufen...



Runden

- Wenn wir `ceil(-11.6)` aufrufen, dann kommt `-11` heraus, weil aufrunden.



Runden

- Wenn wir `ceil(11.6)` aufrufen...



Runden



- Wenn wir `ceil(11.6)` aufrufen, dann kommt `12` heraus, weil aufrunden.

tweise@weise-laptop: ~

```
0
>>> trunc(-0.6)
0
>>> ceil(0.4)
1
>>> ceil(-11.1)
-11
>>> ceil(-11.6)
-11
>>> ceil(11.6)
12
>>>
```

Runden

- Wenn wir `trunc(11.6)` aufrufen...



Runden

- Wenn wir `trunc(11.6)` aufrufen, dann kommt `11` heraus, weil wir die Nachkommastellen abschneiden.



Runden

- Die Funktion `int(x)` funktioniert genau wie `trunc(x)`, akzeptiert aber auch andere Datentypen als Parameter und nicht nur `float`.



Runden

- Die Funktion `int(x)` funktioniert genau wie `trunc(x)`, akzeptiert aber auch andere Datentypen als Parameter und nicht nur `float`.
- `trunc(x)` ist als mathematische Operation zu verstehen, wohingegen `int(x)` as „Datentypumwandlung“ zu betrachten ist ... die genau wie `trunc` funktioniert[100].



Runden

- Wenn wir `int(0.9)` aufrufen...



Runden

- Wenn wir `int(0.9)` aufrufen, dann kommt `0` heraus, weil der Ganzzahlteil von `0.9` nämlich `1` ist.



Runden

- Wenn wir `int(-0.9)` aufrufen...



Runden

- Wenn wir `int(-0.9)` aufrufen, dann kommt `0` heraus, weil der Ganzzahlteil von `-0.9` nämlich `1` ist.



Runden



- Wenn wir `int(11.6)` aufrufen...

tweise@weise-laptop: ~

```
-11
>>> ceil(-11.6)
-11
>>> ceil(11.6)
12
>>> trunc(11.6)
11
>>> int(0.9)
0
>>> int(-0.9)
0
>>> int(11.6)
```

Runden

- Wenn wir `int(11.6)` aufrufen, dann kommt `11` heraus, weil der Ganzzahlteil von `11.6` nämlich `11` ist.



Runden

- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.



Runden

- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Haben wir `x = 11.5`, dann rechnen wir `int(11.5 + 0.5)`...



Runden

- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Haben wir `x = 11.5`, dann rechnen wir `int(11.5 + 0.5)` und es kommt `12` heraus.



Runden

- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Haben wir `x = 12.5`, dann rechnen wir `int(12.5 + 0.5)`...



Runden

- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Haben wir `x = 12.5`, dann rechnen wir `int(12.5 + 0.5)` und es kommt `13` heraus.



Runden



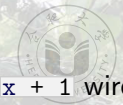
- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Damit sind wir mit dem Thema „runden“ fertig.

Runden



- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Damit sind wir mit dem Thema „runden“ fertig.
- Beachten Sie, wie wichtig es seien kann, die Funktionen, die wir benutzen, genau zu verstehen.

Runden



- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Damit sind wir mit dem Thema „runden“ fertig.
- Beachten Sie, wie wichtig es seien kann, die Funktionen, die wir benutzen, genau zu verstehen.
- `round` funktioniert zum Beispiel anders als `Math.round` in Java[50]...

Wissenschaftliche Notation



Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.
- Es wäre schon sehr komisch, eine 1 gefolgt von 300 Nullen zu schreiben, um 10^{300} auszugeben.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.
- Es wäre schon sehr komisch, eine 1 gefolgt von 300 Nullen zu schreiben, um 10^{300} auszugeben.
- Es wäre auch falsch.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.
- Es wäre schon sehr komisch, eine 1 gefolgt von 300 Nullen zu schreiben, um 10^{300} auszugeben.
- Es wäre auch falsch.
- Ein `float` ist genau auf zwischen 15 und 16 Ziffern.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.
- Es wäre schon sehr komisch, eine 1 gefolgt von 300 Nullen zu schreiben, um 10^{300} auszugeben.
- Es wäre auch falsch.
- Ein `float` ist genau auf zwischen 15 und 16 Ziffern.
- Die ersten 15 Nullen könnten vielleicht korrekt sein ... die restlichen Ziffern wären aber genaugenommen *undefiniert*.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.
- Es wäre schon sehr komisch, eine 1 gefolgt von 300 Nullen zu schreiben, um 10^{300} auszugeben.
- Es wäre auch falsch.
- Ein `float` ist genau auf zwischen 15 und 16 Ziffern.
- Die ersten 15 Nullen könnten vielleicht korrekt sein ... die restlichen Ziffern wären aber genaugenommen *undefiniert*.
- Python löst dieses Problem wir viele Programmiersprachen: mit der wissenschaftlichen Notation für Fließkommazahlen.

Die Wissenschaftliche Notation: Einleitung



- Wir haben gelernt, dass `floats` Zahlen so groß wie 10^{300} und so klein wie 10^{-300} darstellen können.
- Das führt uns zu der Frage, wie man denn solche Zahlen anzeigen kann.
- Es wäre schon sehr komisch, eine 1 gefolgt von 300 Nullen zu schreiben, um 10^{300} auszugeben.
- Es wäre auch falsch.
- Ein `float` ist genau auf zwischen 15 und 16 Ziffern.
- Die ersten 15 Nullen könnten vielleicht korrekt sein ... die restlichen Ziffern wären aber genaugenommen *undefiniert*.
- Python löst dieses Problem wir viele Programmiersprachen: mit der wissenschaftlichen Notation für Fließkommazahlen.
- Das ist eine Schreibweise für die Ein- und Ausgabe von Zahlen (intern werden die Zahlen jedoch ganz normal nach dem IEEE 754 Standard[45,48] dargestellt).

Die Wissenschaftliche Notation: Arbeitsweise



- Python benutzt die wissenschaftliche Notation automatisch für die Ausgabe aller `float`-Werte, deren Betrag kleiner als 10^{-4} oder größer/gleich 10^{16} ist.

Die Wissenschaftliche Notation: Arbeitsweise



- Python benutzt die wissenschaftliche Notation automatisch für die Ausgabe aller `float`-Werte, deren Betrag kleiner als 10^{-4} oder größer/gleich 10^{16} ist.
- Solche Zahlen, nennen wir sie α, werden dann repräsentiert als $\alpha = \beta * 10^{\gamma}$ und wir schreiben $\beta\mathbf{e}\gamma$.

Die Wissenschaftliche Notation: Arbeitsweise



- Python benutzt die wissenschaftliche Notation automatisch für die Ausgabe aller `float`-Werte, deren Betrag kleiner als 10^{-4} oder größer/gleich 10^{16} ist.
- Solche Zahlen, nennen wir sie α, werden dann repräsentiert als $\alpha = \beta * 10^{\gamma}$ und wir schreiben $\beta \mathbf{e} \gamma$.
- Um diese Darstellung eindeutig zu machen, ist spezifiziert dass β genau eine Ziffer vor dem Dezimalpunkt hat, welche nicht 0 sein kann.

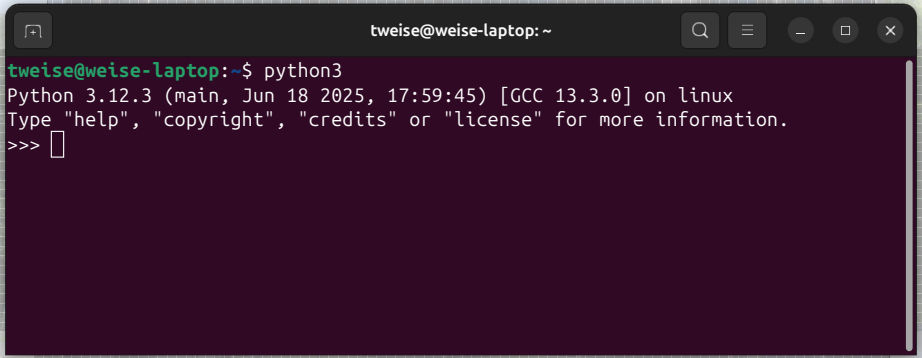
Die Wissenschaftliche Notation: Arbeitsweise

- Python benutzt die wissenschaftliche Notation automatisch für die Ausgabe aller `float`-Werte, deren Betrag kleiner als 10^{-4} oder größer/gleich 10^{16} ist.
- Solche Zahlen, nennen wir sie α, werden dann repräsentiert als $\alpha = \beta * 10^{\gamma}$ und wir schreiben $\beta\mathtt{e}\gamma$.
- Um diese Darstellung eindeutig zu machen, ist spezifiziert dass β genau eine Ziffer vor dem Dezimalpunkt hat, welche nicht 0 sein kann.

β	γ	$\equiv$	α
A.BCDEFG...	e+HIJ	$\equiv$	A.BCDEFG... $* 10^{HIJ}$
A.BCDEFG...	e-HIJ	$\equiv$	A.BCDEFG... $* 10^{-HIJ}$
-A.BCDEFG...	e+HIJ	$\equiv$	$-$A.BCDEFG... $* 10^{HIJ}$
-A.BCDEFG...	e-HIJ	$\equiv$	$-$A.BCDEFG... $* 10^{-HIJ}$

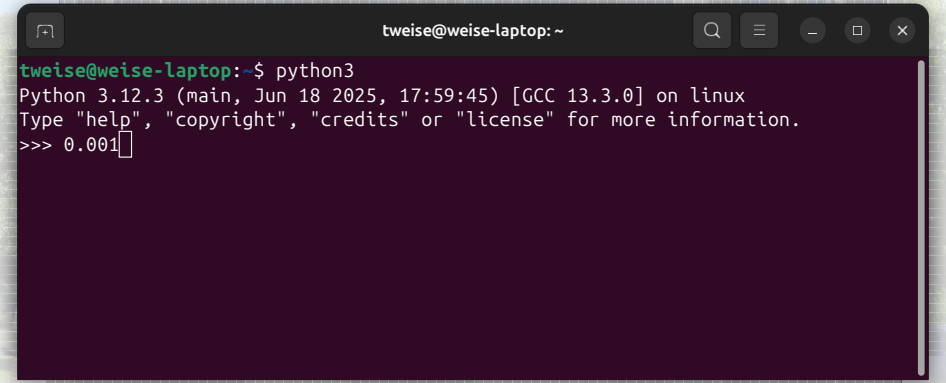
Die Wissenschaftliche Notation: Beispiele

- Probieren wir das mal aus.



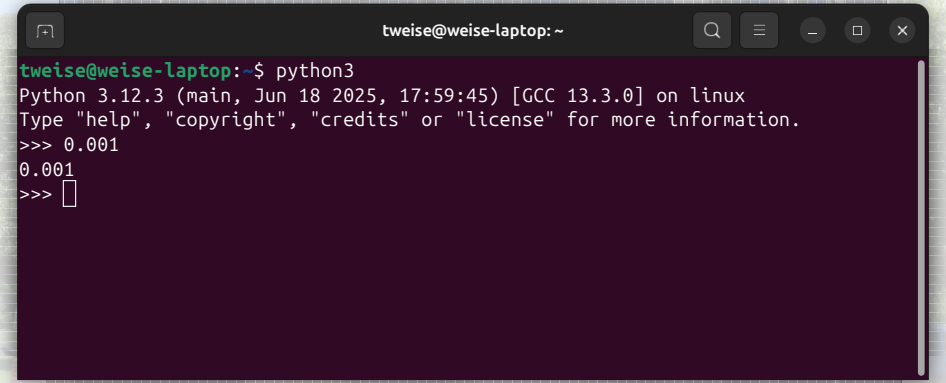
Die Wissenschaftliche Notation: Beispiele

- Wenn wir `0.001` schreiben und Python damit zwingen, diese Zahl auszugeben...



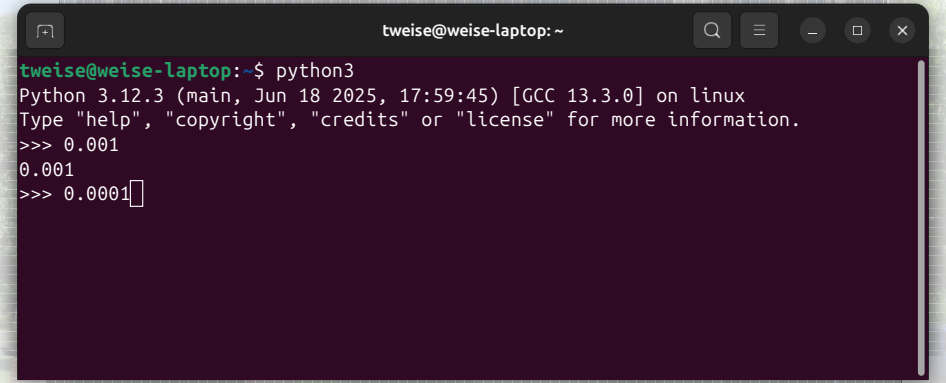
Die Wissenschaftliche Notation: Beispiele

- Wenn wir `0.001` schreiben und Python damit zwingen, diese Zahl auszugeben, dann gibt es auch `0.001` wieder aus.



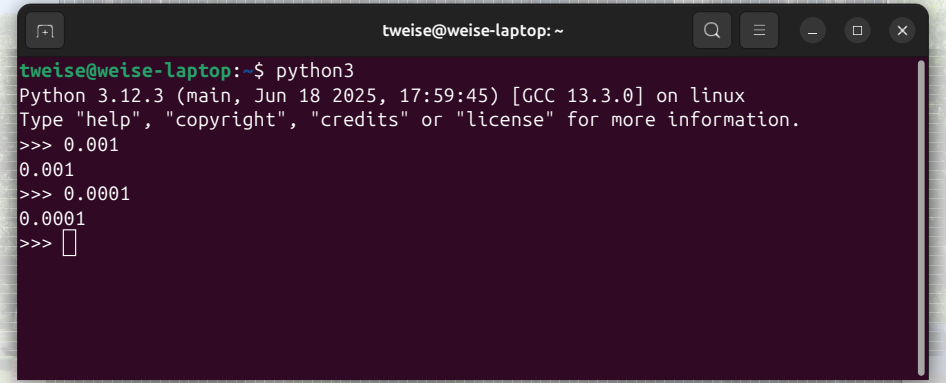
Die Wissenschaftliche Notation: Beispiele

- Wenn wir `0.0001` schreiben und Python damit zwingen, diese Zahl auszugeben...



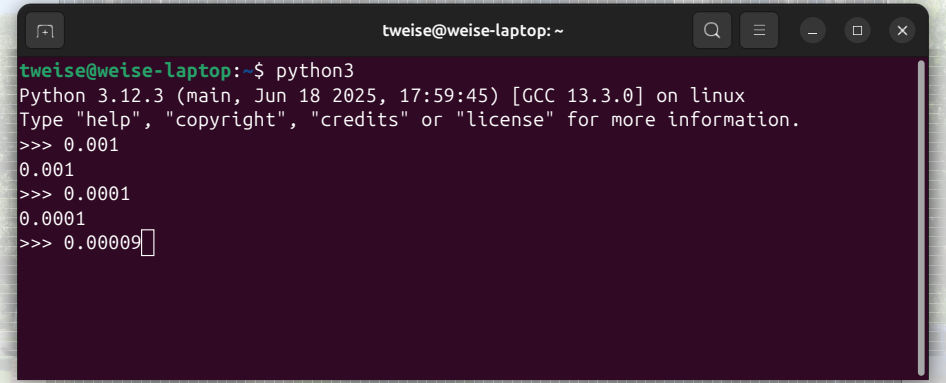
Die Wissenschaftliche Notation: Beispiele

- Wenn wir `0.0001` schreiben und Python damit zwingen, diese Zahl auszugeben, dann gibt es auch `0.0001` wieder aus.



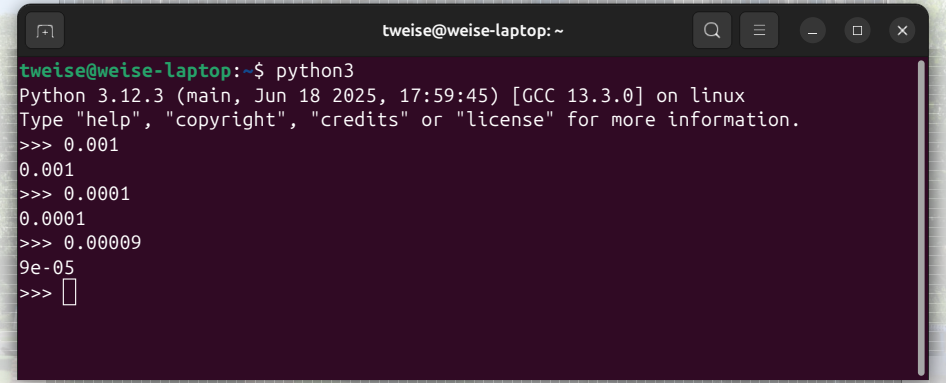
Die Wissenschaftliche Notation: Beispiele

- Wenn wir aber `0.00009` schreiben und Python damit zwingen, diese Zahl auszugeben...



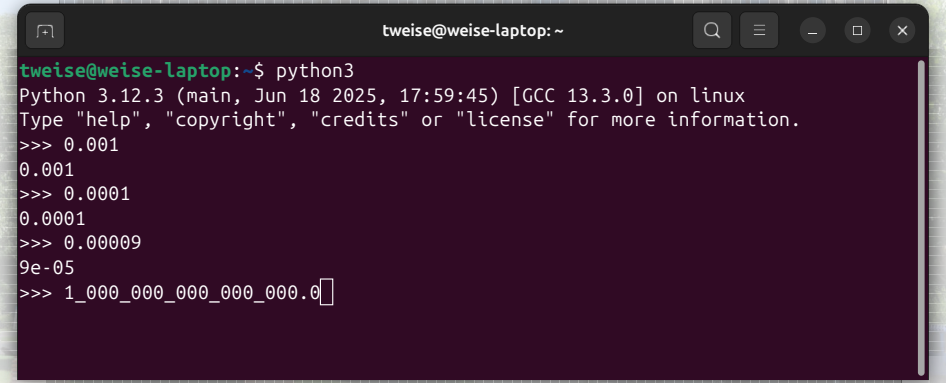
Die Wissenschaftliche Notation: Beispiele

- Wenn wir aber `0.00009` schreiben und Python damit zwingen, diese Zahl auszugeben, dann benutzt es die wissenschaftliche Notation und gibt `9e-05` aus.



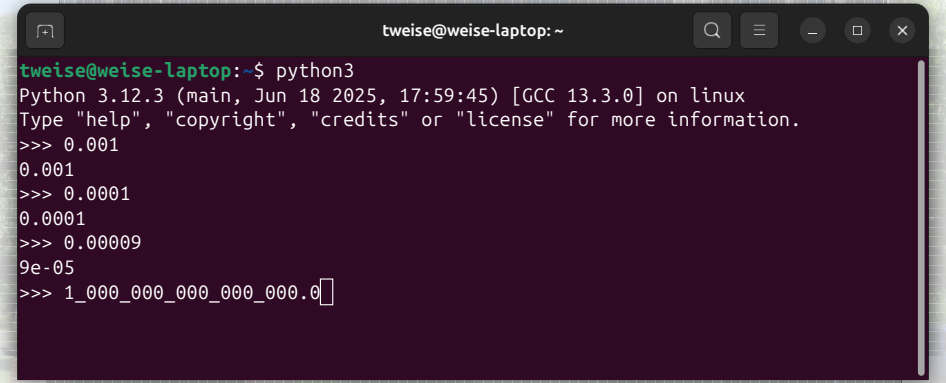
Die Wissenschaftliche Notation: Beispiele

- Python erlaubt uns, Unterstriche (_) beliebig in Zahlen einzufügen, um die Lesbarkeit zu erhöhen.



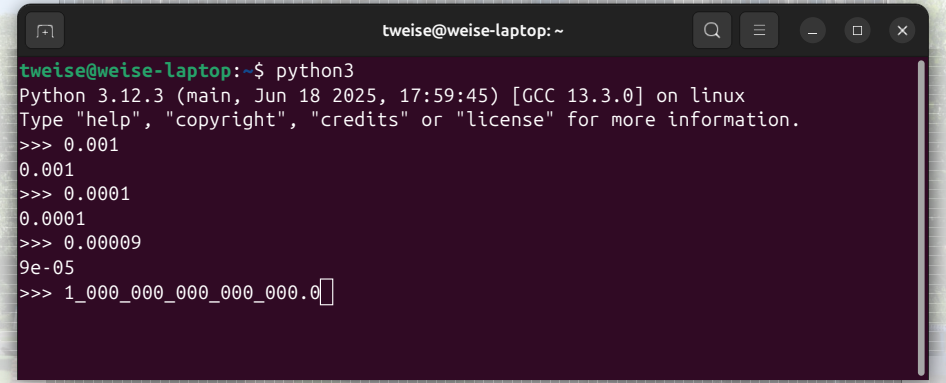
Die Wissenschaftliche Notation: Beispiele

- Python erlaubt uns, Unterstriche (_) beliebig in Zahlen einzufügen, um die Lesbarkeit zu erhöhen.
- Diese Unterstriche werden von Python ignoriert.



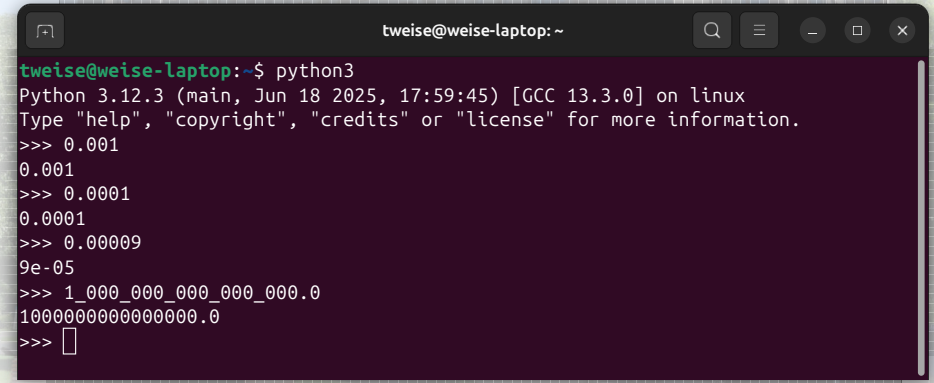
Die Wissenschaftliche Notation: Beispiele

- Wenn wir `1_000_000_000_000_000.0` schreiben (was das gleiche ist wie `1000000000000000.0` bzw. $1 * 10^{15}$) und Python damit zwingen, diese Zahl auszugeben...



Die Wissenschaftliche Notation: Beispiele

- Wenn wir `1_000_000_000_000_000.0` schreiben (was das gleiche ist wie `1000000000000000.0` bzw. $1 * 10^{15}$) und Python damit zwingen, diese Zahl auszugeben, dann schreibt es `1000000000000000.0`.



Die Wissenschaftliche Notation: Beispiele

- Wenn wir `9_000_000_000_000_000.0` schreiben (was das gleiche ist wie `9000000000000000.0` bzw. $9 * 10^{15}$) und Python damit zwingen, diese Zahl auszugeben...

```
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.
>>> 0.001
0.001
>>> 0.0001
0.0001
>>> 0.00009
9e-05
>>> 1_000_000_000_000_000.0
1000000000000000.0
>>> 9_000_000_000_000_000.0
```

Die Wissenschaftliche Notation: Beispiele

- Wenn wir `9_000_000_000_000_000.0` schreiben (was das gleiche ist wie `9000000000000000.0` bzw. $9 * 10^{15}$) und Python damit zwingen, diese Zahl auszugeben, dann schreibt es `9000000000000000.0`.

```
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.
>>> 0.001
0.001
>>> 0.0001
0.0001
>>> 0.00009
9e-05
>>> 1_000_000_000_000_000.0
1000000000000000.0
>>> 9_000_000_000_000_000.0
9000000000000000.0
>>>
```

Die Wissenschaftliche Notation: Beispiele

- Wenn wir aber `9_999_999_999_999_999.0` schreiben (was das gleiche ist wie `9999999999999999.0` bzw. $9.999999999999999 * 10^{15}$) und Python damit zwingen, diese Zahl auszugeben...

```
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.
>>> 0.001
0.001
>>> 0.0001
0.0001
>>> 0.00009
9e-05
>>> 1_000_000_000_000_000.0
1000000000000000.0
>>> 9_000_000_000_000_000.0
9000000000000000.0
>>> 9_999_999_999_999_999.0
```

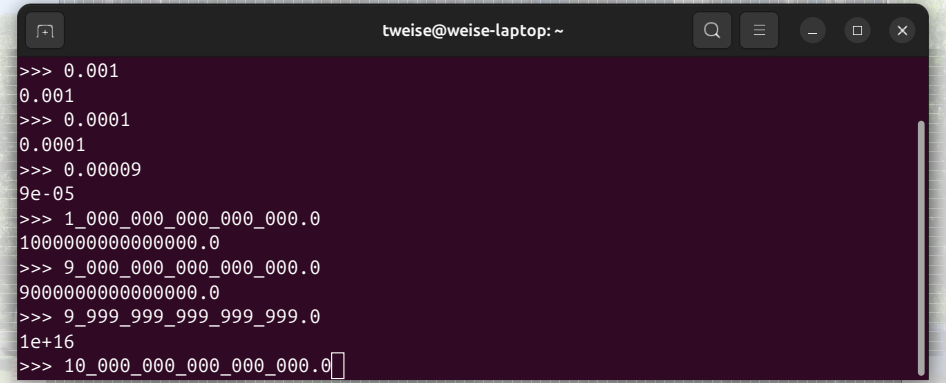
Die Wissenschaftliche Notation: Beispiele

- Wenn wir aber `9_999_999_999_999_999.0` schreiben (was das gleiche ist wie `9999999999999999.0` bzw. $9.999999999999999 * 10^{15}$) und Python damit zwingen, diese Zahl auszugeben, dann schreibt es `1e+16` in der wissenschaftlichen Notation (und weil wir nur 15 Ziffern haben können aber 16 9en angegeben haben...).



Die Wissenschaftliche Notation: Beispiele

- Wenn wir aber `10_000_000_000_000_000.0` schreiben (was das gleiche ist wie `10000000000000000.0`) und Python damit zwingen, diese Zahl auszugeben...



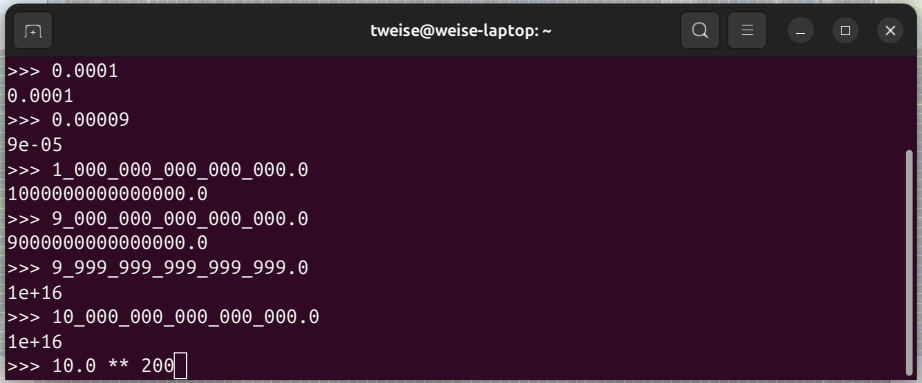
Die Wissenschaftliche Notation: Beispiele

- Wenn wir aber `10_000_000_000_000_000.0` schreiben (was das gleiche ist wie `10000000000000000.0`) und Python damit zwingen, diese Zahl auszugeben, dann schreibt es `1e+16` in der wissenschaftlichen Notation.



Die Wissenschaftliche Notation: Beispiele

- Berechnen wir 10^{200} als `float` in dem wir schreiben `10.0 ** 200`...



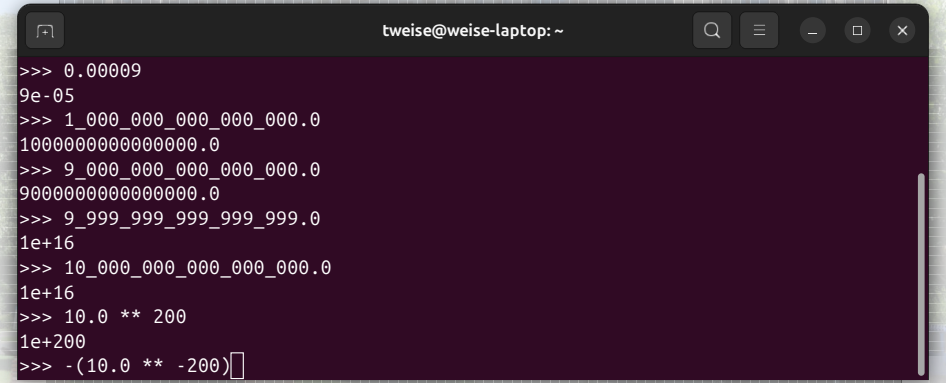
Die Wissenschaftliche Notation: Beispiele

- Berechnen wir 10^{200} als `float` in dem wir schreiben `10.0 ** 200`, dann schreibt es `1e+200` in der wissenschaftlichen Notation.



Die Wissenschaftliche Notation: Beispiele

- Berechnen wir $-(10^{-200})$ als `float` in dem wir schreiben `-(10.0 ** -200)`...



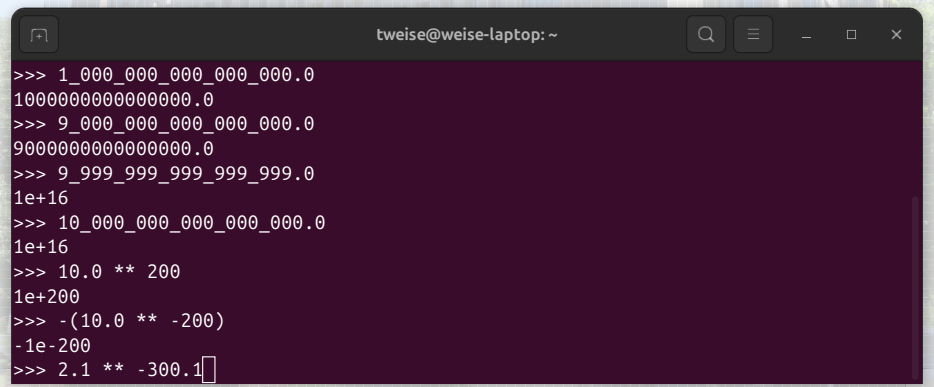
Die Wissenschaftliche Notation: Beispiele

- Berechnen wir $-(10^{-200})$ als `float` in dem wir schreiben `-(10.0 ** -200)`, dann schreibt es `-1e-200` in der wissenschaftlichen Notation.



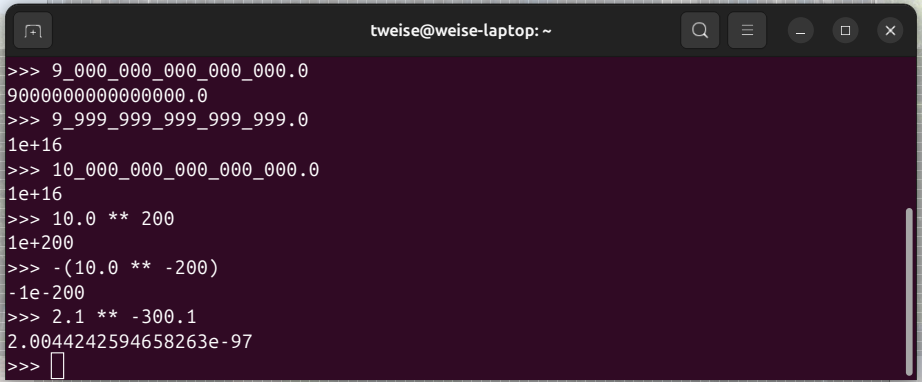
Die Wissenschaftliche Notation: Beispiele

- Berechnen wir $2.1^{-300.1}$ als `float` in dem wir schreiben `2.1 ** -300.1`...



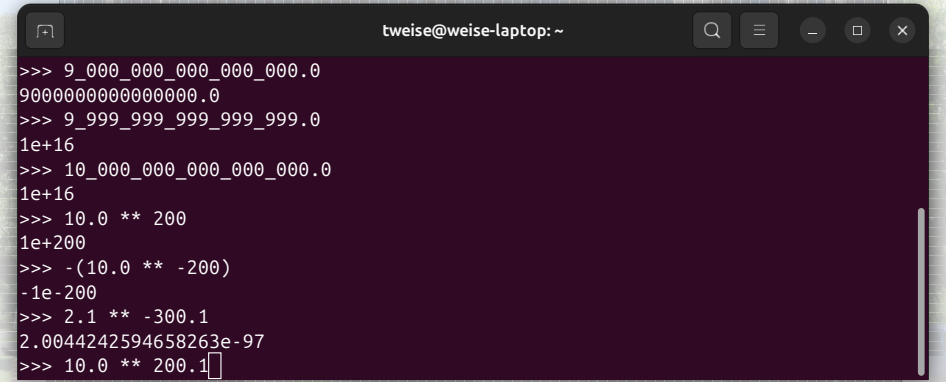
Die Wissenschaftliche Notation: Beispiele

- Berechnen wir $2.1^{-300.1}$ als `float` in dem wir schreiben `2.1 ** -300.1`, dann schreibt es `2.0044242594658263e-97` in der wissenschaftlichen Notation.



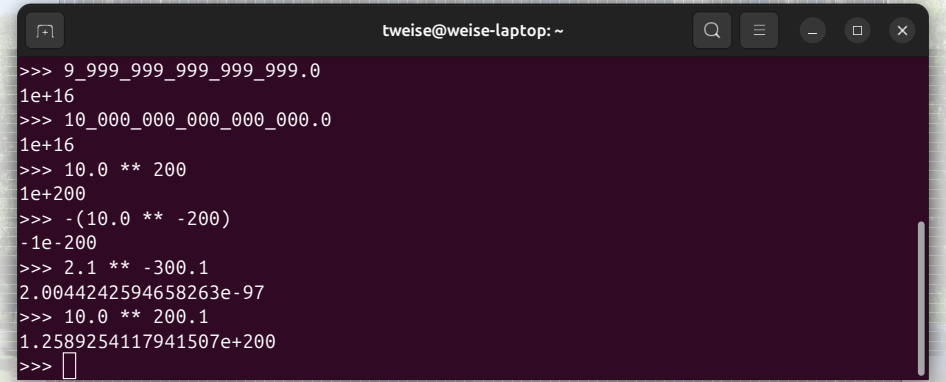
Die Wissenschaftliche Notation: Beispiele

- Berechnen wir $10^{200.1}$ als `float` in dem wir schreiben `10.0 ** 200.1`...



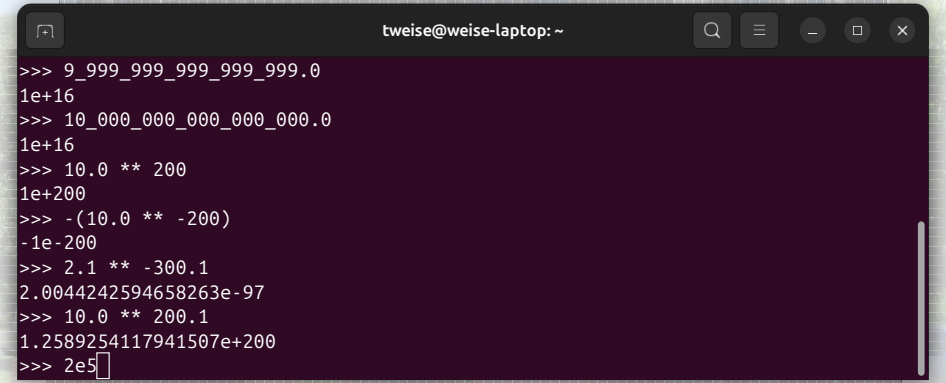
Die Wissenschaftliche Notation: Beispiele

- Berechnen wir $10^{200.1}$ als `float` in dem wir schreiben `10.0 ** 200.1`, dann schreibt es `1.2589254117941507e+200` in der wissenschaftlichen Notation.



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir dagegen `2e5` in der wissenschaftlichen Notation...



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir dagegen `2e5` in der wissenschaftlichen Notation, dann schreibt Python es als `200000.0`.



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir `2.34e10` in der wissenschaftlichen Notation...



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir `2.34e10` in der wissenschaftlichen Notation, dann schreibt Python es als `23400000000.0`.



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir stattdessen `2.3456e+16` in der wissenschaftlichen Notation...



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir stattdessen `2.3456e+16` in der wissenschaftlichen Notation, dann schreibt Python es ebenfalls als `2.3456e+16` in der wissenschaftlichen Notation.



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir stattdessen `-12e30` in fast der wissenschaftlichen Notation...



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir stattdessen `-12e30` in fast der wissenschaftlichen Notation, dann schreibt Python es ebenfalls als `-1.2e31` in der wissenschaftlichen Notation mit nur einer Vorkomma-Stelle.



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir `0.023e-20` in fast der wissenschaftlichen Notation...



Die Wissenschaftliche Notation: Beispiele

- Schreiben wir `0.023e-20` in fast der wissenschaftlichen Notation, dann schreibt Python es ebenfalls als `2.3e-22` in der wissenschaftlichen Notation mit genau einer Vorkomma-Stelle, die nicht 0 ist.



Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.

Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.
- Es ust Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder „normal“ – der Wert wird zur selben binären Darstellung gemapped.

Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.
- Es ust Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder „normal“ – der Wert wird zur selben binären Darstellung gemapped.
- Die wissenschaftliche Notation ist wieder nur ein Textformat für die Ein- und Ausgabe von (`float`) Zahlen.

Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.
- Es ust Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder „normal“ – der Wert wird zur selben binären Darstellung gemapped.
- Die wissenschaftliche Notation ist wieder nur ein Textformat für die Ein- und Ausgabe von (`float`) Zahlen.
- Ein `float` „weiß nicht“ wie es eingegeben wurde.

Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.
- Es ust Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder „normal“ – der Wert wird zur selben binären Darstellung gemapped.
- Die wissenschaftliche Notation ist wieder nur ein Textformat für die Ein- und Ausgabe von (`float`) Zahlen.
- Ein `float` „weiß nicht“ wie es eingegeben wurde.
- Bei der Ausgabe entscheided Python automatisch, welches das sinnvollste Format ist.

Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.
- Es ust Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder „normal“ – der Wert wird zur selben binären Darstellung gemapped.
- Die wissenschaftliche Notation ist wieder nur ein Textformat für die Ein- und Ausgabe von (`float`) Zahlen.
- Ein `float` „weiß nicht“ wie es eingegeben wurde.
- Bei der Ausgabe entscheided Python automatisch, welches das sinnvollste Format ist.

Gute Praxis

Wenn Sie große `int`- oder `float`-Werte spezifizieren müssen, benutzen Sie Unterstriche (`_`) um Gruppen von Ziffern zu trennen[13]. `37_859_378` ist z. B. viel leichter zu lesen als `37859378`.

Grenzen

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?
- Python kann sehr kleine Zahlen, z. B. 10^{-300}, als `floats` speichern.

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?
- Python kann sehr kleine Zahlen, z. B. 10^{-300}, als `floats` speichern.
- Aber wie klein *genau*?

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?
- Python kann sehr kleine Zahlen, z. B. 10^{-300}, als `floats` speichern.
- Aber wie klein *genau*?
- Java benutzt den selben IEEE 754 Standard[48] für seinen Datentyp `double`.

Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?
- Python kann sehr kleine Zahlen, z. B. 10^{-300}, als `floats` speichern.
- Aber wie klein *genau*?
- Java benutzt den selben IEEE 754 Standard[48] für seinen Datentyp `double`.
- In der entsprechenden Dokumentation[21] finden wir, dass der kleinste Wert 2^{-1074} ist, was ein etwa $4.940\,656\,458\,412\,465\,44 * 10^{-324}$ entspricht.

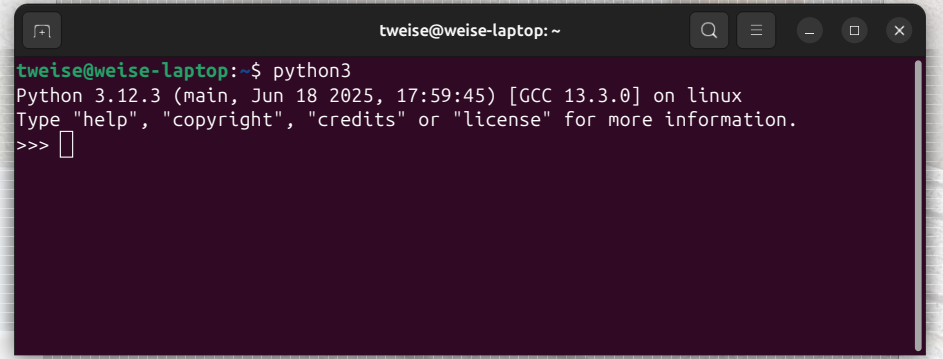
Sehr kleine Zahlen



- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?
- Python kann sehr kleine Zahlen, z. B. 10^{-300}, als `floats` speichern.
- Aber wie klein *genau*?
- Java benutzt den selben IEEE 754 Standard[48] für seinen Datentyp `double`.
- In der entsprechenden Dokumentation[21] finden wir, dass der kleinste Wert 2^{-1074} ist, was ein etwa $4.940\,656\,458\,412\,465\,44 * 10^{-324}$ entspricht.
- Der kleinste Wert für Python's `floats` sollte also auch in etwa in dieser Gegend liegen.

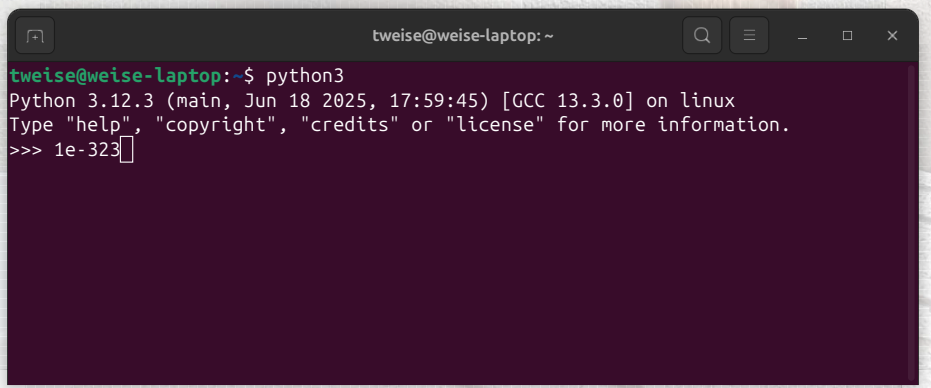
Sehr kleine Zahlen: Ausprobieren

- Probieren wir das mal aus: Wir geben verschiedene sehr kleine Zahlen in den Python-Interpreter ein und gucken, was passiert.



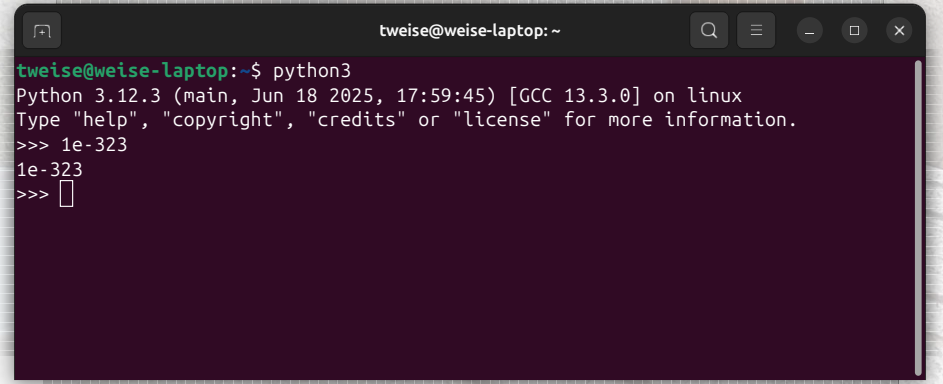
Sehr kleine Zahlen: Ausprobieren

- Wir geben `1e-323` ein...



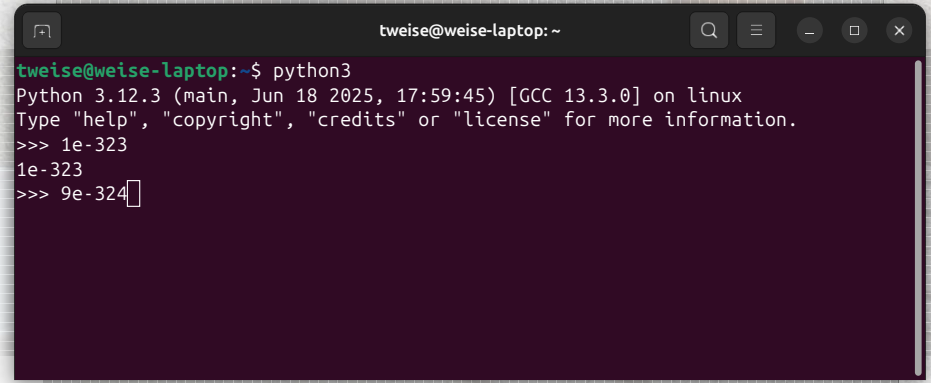
Sehr kleine Zahlen: Ausprobieren

- Wir geben `1e-323` ein und Python gibt auch wieder `1e-323` aus.



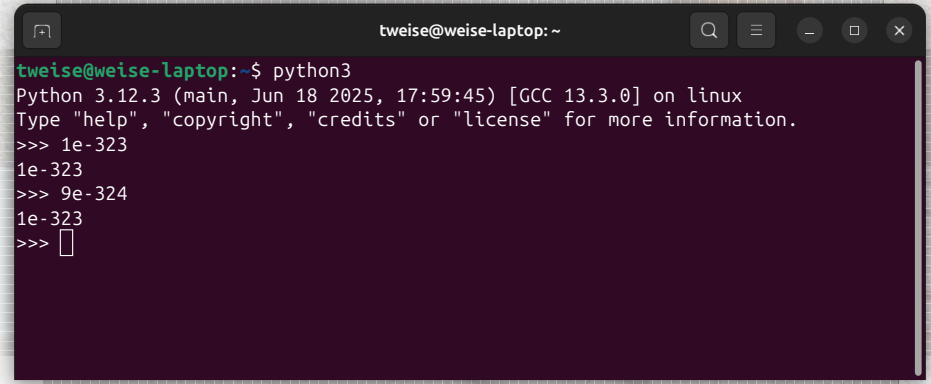
Sehr kleine Zahlen: Ausprobieren

- Wir geben `9e-324` ein...



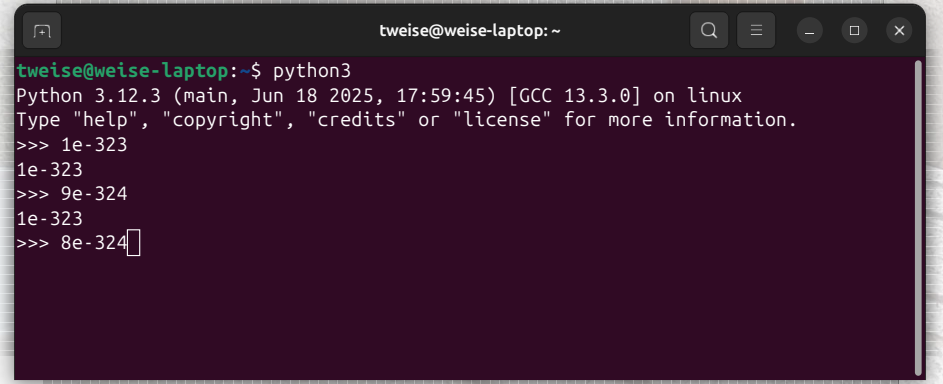
Sehr kleine Zahlen: Ausprobieren

- Wir geben `9e-324` ein und Python gibt uns stattdessen wieder `1e-323` aus. Wir sind bereits so nahe an 0, dass nur noch die letzten Bits des Signifikand genutzt werden und viele Zahlen zusammenfallen.



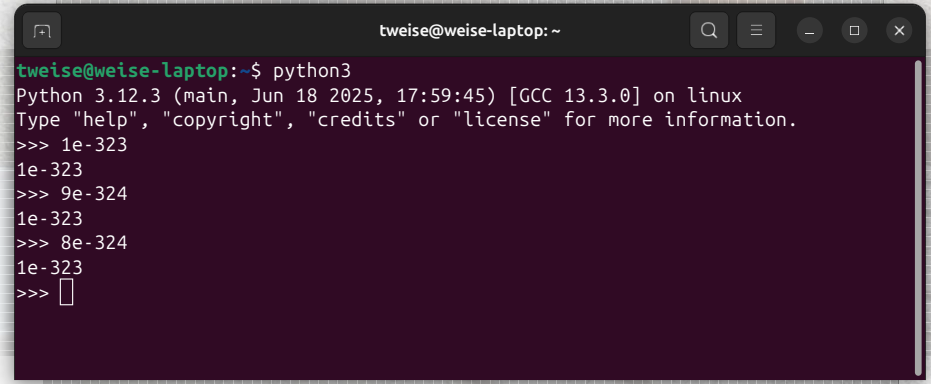
Sehr kleine Zahlen: Ausprobieren

- Wir geben `8e-324` ein...



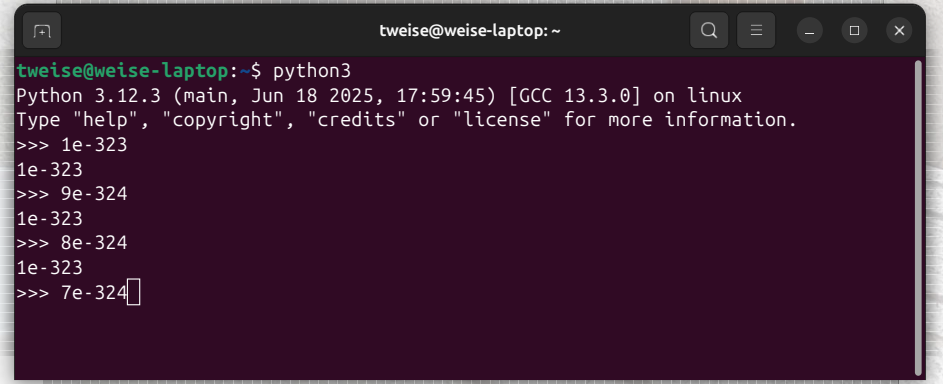
Sehr kleine Zahlen: Ausprobieren

- Wir geben `8e-324` ein und Python gibt uns wieder `1e-323` aus. Wir sind bereits so nahe an 0, dass nur noch die letzten Bits des Signifikand genutzt werden und viele Zahlen zusammenfallen.



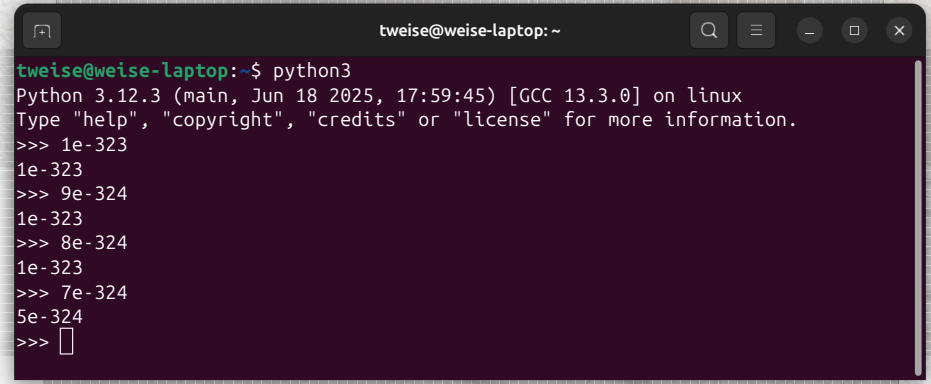
Sehr kleine Zahlen: Ausprobieren

- Wir geben `7e-324` ein



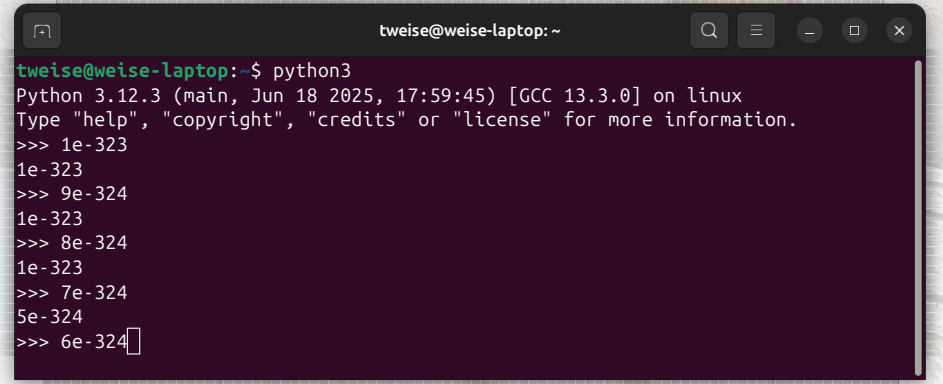
Sehr kleine Zahlen: Ausprobieren

- Wir geben `7e-324` ein und Python gibt uns nun `5e-324` aus. Wir sind bereits so nahe an 0, dass nur noch die letzten Bits des Signifikand genutzt werden und viele Zahlen zusammenfallen.



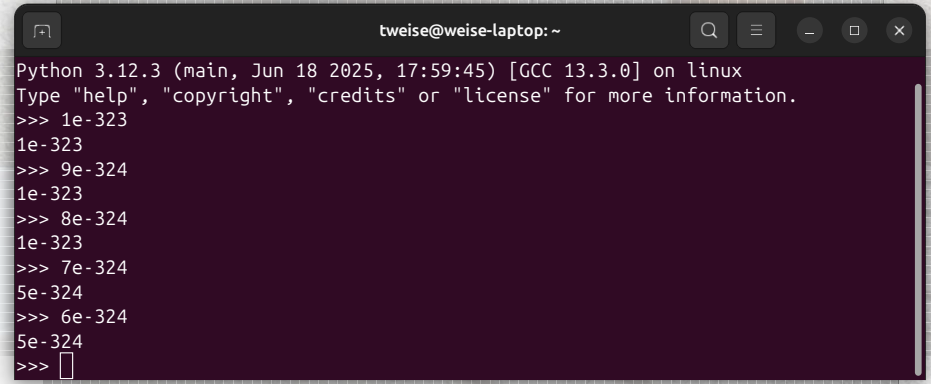
Sehr kleine Zahlen: Ausprobieren

- Wir geben `6e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `6e-324` ein und Python gibt uns wieder `5e-324` aus. Wir sind bereits so nahe an 0, dass nur noch die letzten Bits des Signifikand genutzt werden und viele Zahlen zusammenfallen.



Sehr kleine Zahlen: Ausprobieren

- Wir geben `5e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `5e-324` ein und Python gibt uns wieder `5e-324` aus. Das bedeutet, dass im `float` Format `7e-324 == 6e-324 == 5e-324` gilt.



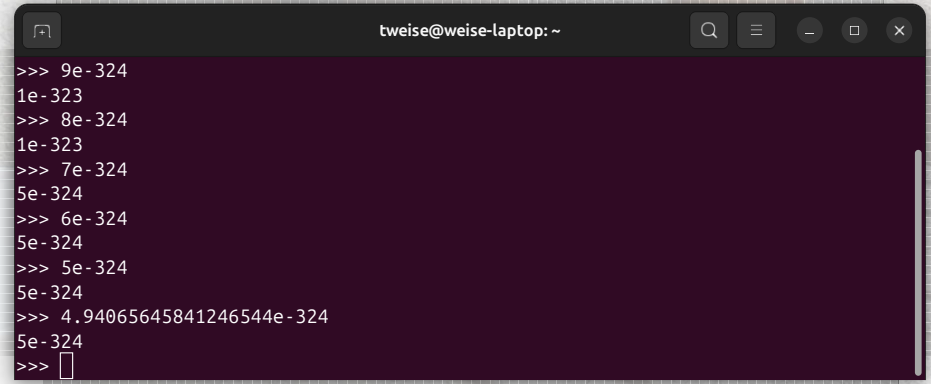
Sehr kleine Zahlen: Ausprobieren

- Wir geben `4.94065645841246544e-324` ein, nämlich die kleinste `double`-Zahl in Java[21]...



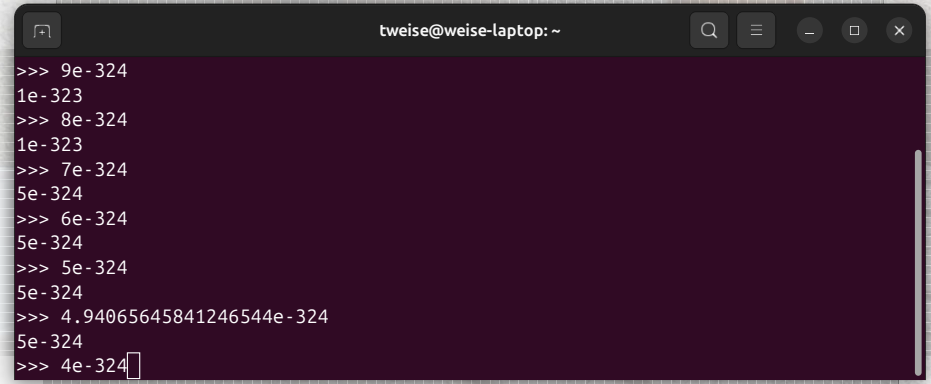
Sehr kleine Zahlen: Ausprobieren

- Wir geben `4.94065645841246544e-324` ein, nämlich die kleinste `double`-Zahl in Java[21] und Python gibt uns wieder `5e-324` aus.



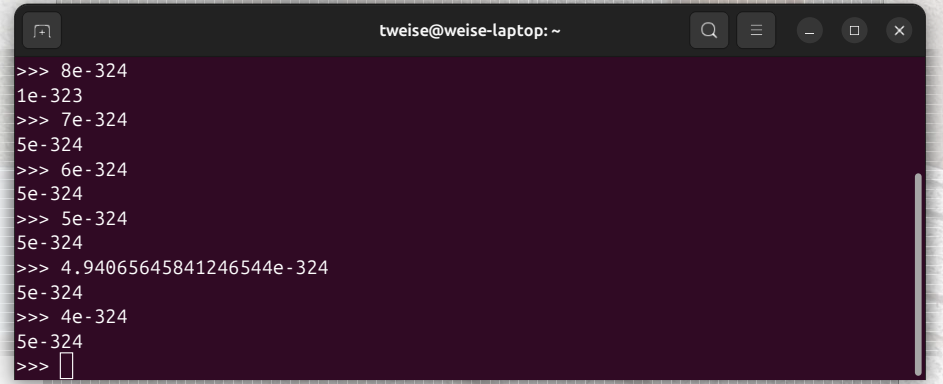
Sehr kleine Zahlen: Ausprobieren

- Wir geben `4e-324` ein...



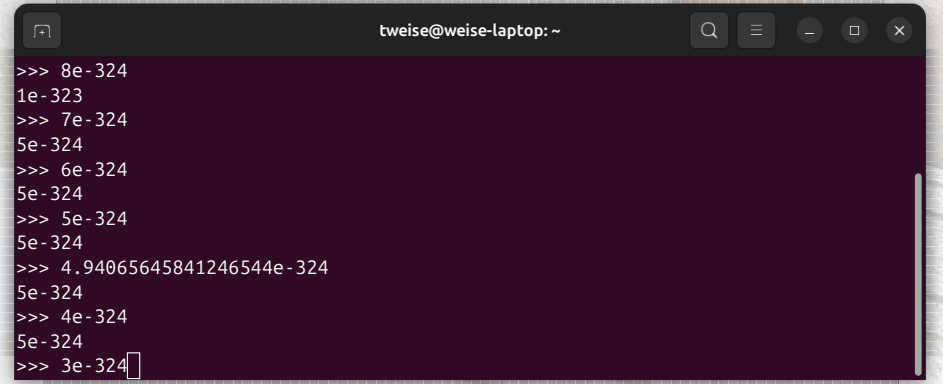
Sehr kleine Zahlen: Ausprobieren

- Wir geben `4e-324` ein und Python gibt uns wieder `5e-324` aus. Das bedeutet, dass im `float` Format `7e-324 == 4e-324` gilt.



Sehr kleine Zahlen: Ausprobieren

- Wir geben `3e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `3e-324` ein und Python gibt uns wieder `5e-324` aus. Das bedeutet, dass im `float` Format `7e-324 == 3e-324` gilt.



Sehr kleine Zahlen: Ausprobieren

- Wir geben `2e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `2e-324` ein und Python gibt uns `0` aus. `2e-324` ist also einfach zu klein, um noch dargestellt zu werden.



Sehr kleine Zahlen: Ausprobieren

- Wir geben `1e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `1e-324` ein und Python gibt uns natürlich auch `0` aus. `1e-324` ist natürlich dann auch zu klein, um noch dargestellt zu werden.



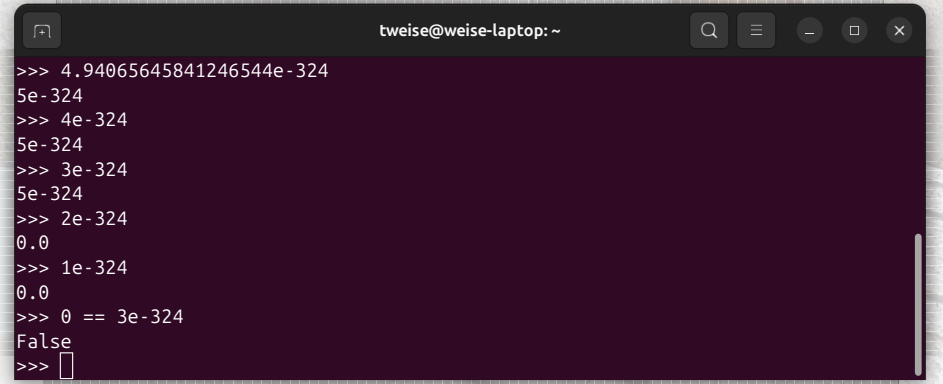
Sehr kleine Zahlen: Ausprobieren

- Wir geben `0 == 3e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `0 == 3e-324` ein und Python sagt, dass das `False`, also *nicht wahr* ist.



Sehr kleine Zahlen: Ausprobieren

- Wir geben `0 == 2e-324` ein...



Sehr kleine Zahlen: Ausprobieren

- Wir geben `0 == 2e-324` ein und Python sagt, dass das `True`, also *wahr* ist. Stimmt ja auch: `2e-324` ist so klein, dass es als `0` gespeichert wird.



Sehr kleine Zahlen: Ausprobieren

- Wir können also sehr kleine Zahlen mit `floats` speichern.



Sehr kleine Zahlen: Ausprobieren



- Wir können also sehr kleine Zahlen mit `floats` speichern.
- Aber es gibt eine Grenze, und wenn wir diese unterschreiten, werden die Zahlen als `0` interpretiert.

Sehr kleine Zahlen: Ausprobieren



- Wir können also sehr kleine Zahlen mit `floats` speichern.
- Aber es gibt eine Grenze, und wenn wir diese unterschreiten, werden die Zahlen als `0` interpretiert.
- Und wenn wir der Grenze sehr nahe kommen, dann fallen immer mehr Zahlen aufeinander, weil wir nur noch die letzten paar Bits des Signifikand nutzen.

Sehr große Zahlen

- Für kleine Zahlen gibt es ein natürliches Ende: sehr sehr kleine Zahlen sind einfach 0.

Sehr große Zahlen

- Für kleine Zahlen gibt es ein natürliches Ende: sehr sehr kleine Zahlen sind einfach 0.
- Was passiert mit sehr großen Zahlen?

Sehr große Zahlen



- Für kleine Zahlen gibt es ein natürliches Ende: sehr sehr kleine Zahlen sind einfach 0.
- Was passiert mit sehr großen Zahlen?
- Die Java-Dokumentation[21] nennt $(2 - 2^{-52}) * 2^{1023} \approx 1.797\,693\,134\,862\,315\,708 \cdots * 10^{308}$ als die größte Zahl, die mit 64-Bit Fließkommazahlen dargstellt werden kann.

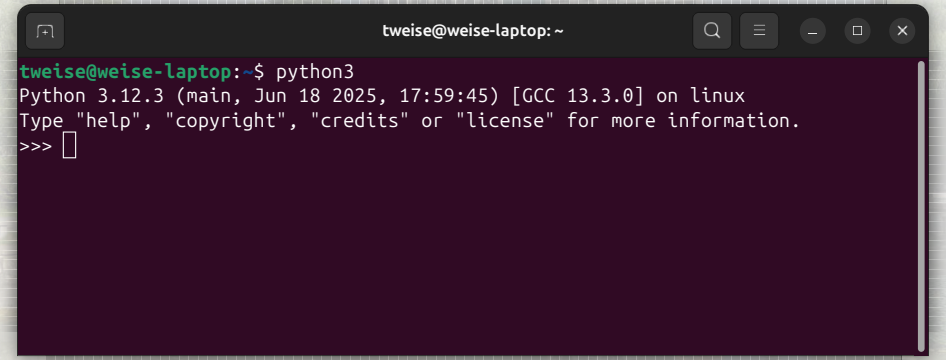
Sehr große Zahlen



- Für kleine Zahlen gibt es ein natürliches Ende: sehr sehr kleine Zahlen sind einfach 0.
- Was passiert mit sehr großen Zahlen?
- Die Java-Dokumentation[21] nennt $(2 - 2^{-52}) * 2^{1023} \approx 1.797\,693\,134\,862\,315\,708 \cdots * 10^{308}$ als die größte Zahl, die mit 64-Bit Fließkommazahlen dargstellt werden kann.
- Was passiert danach?

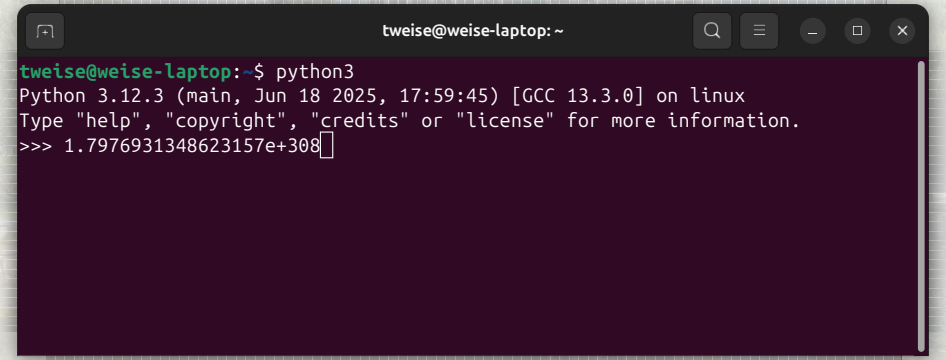
Sehr große Zahlen: Ausprobieren

- Probieren wir das mal aus: Wir geben verschiedene sehr große Zahlen in den Python-Interpreter ein und gucken, was passiert.



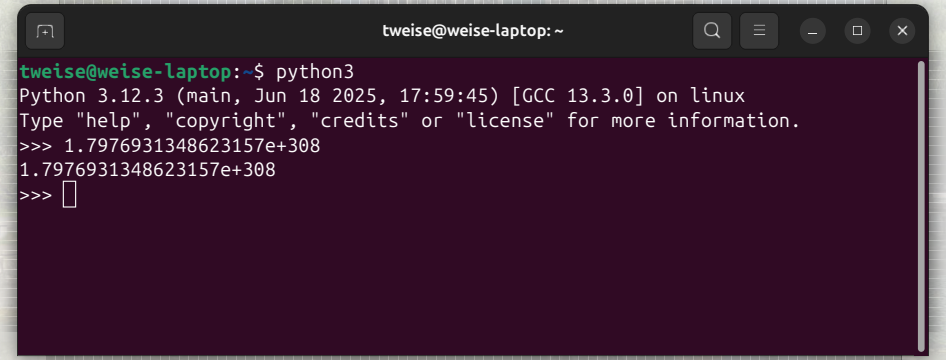
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623157e+308` ein, was geringfügig kleiner als die in der Java-Dokumentation genannte Zahl ist...



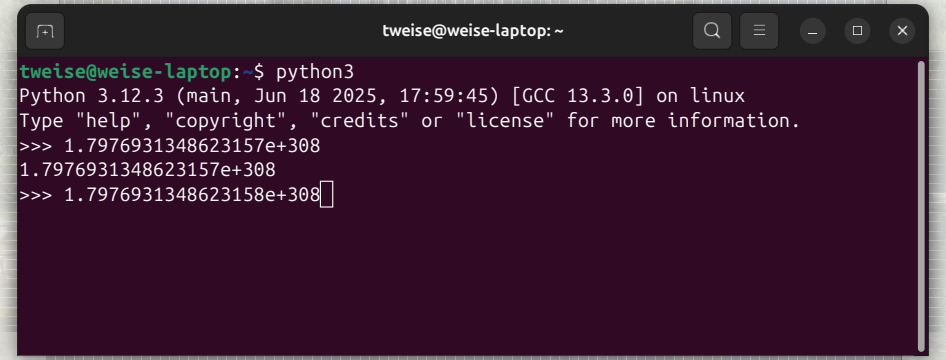
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623157e+308` ein, was geringfügig kleiner als die in der Java-Dokumentation genannte Zahl ist und Python gibt auch wieder `1.7976931348623157e+308` aus.



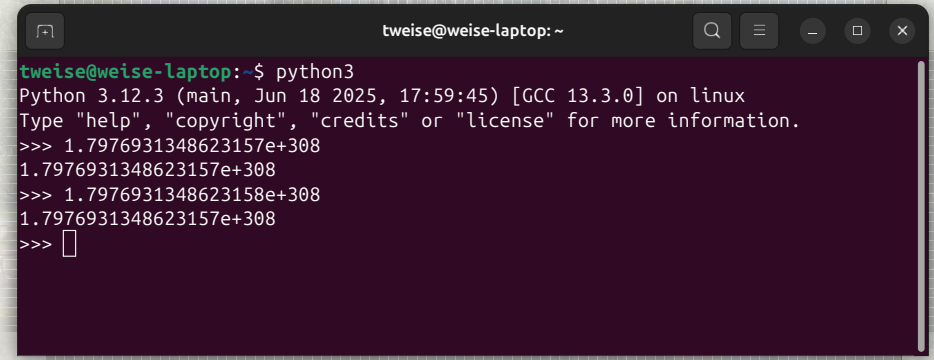
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623158e+308` ein, also fast genau die in der Java-Dokumentation genannte Zahl $1.797\,693\,134\,862\,315\,708 \cdots * 10^{308} \ldots$



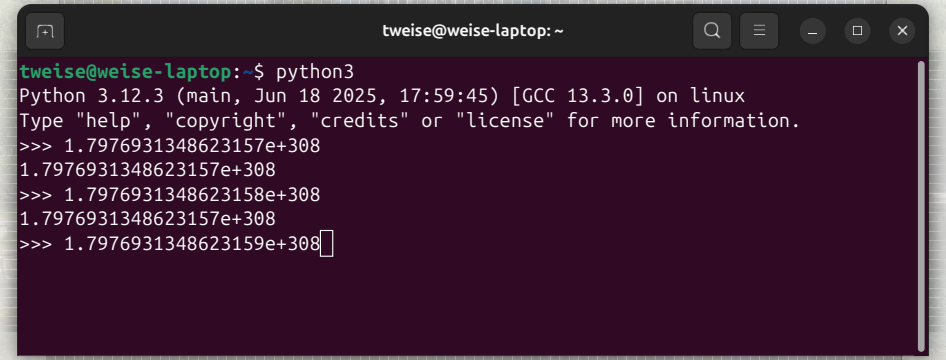
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623158e+308` ein, also fast genau die in der Java-Dokumentation genannte Zahl $1.797\,693\,134\,862\,315\,708 \cdots * 10^{308}$, Python gibt aber wieder `1.7976931348623157e+308` aus ... die beiden Zahlen sind zu nahe beieinander und können bei der Genauigkeit von 15 Ziffern nicht unterschieden werden.



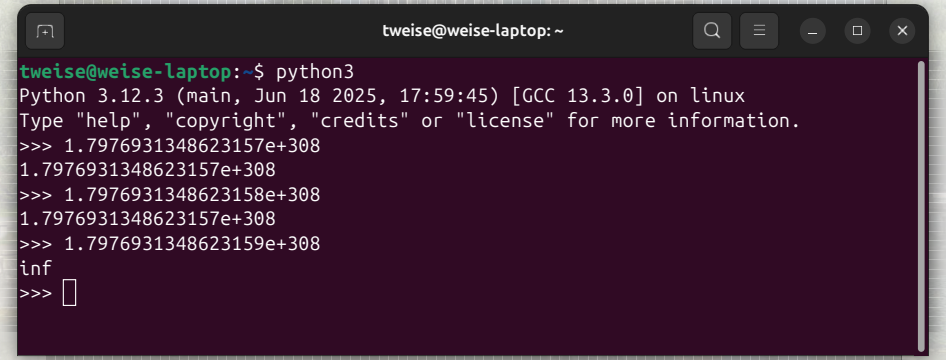
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623159e+308` ein, was geringfügig größer als die in der Java-Dokumentation genannte Zahl ist...



Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623159e+308` ein, was geringfügig größer als die in der Java-Dokumentation genannte Zahl ist und Python gibt `inf` aus.



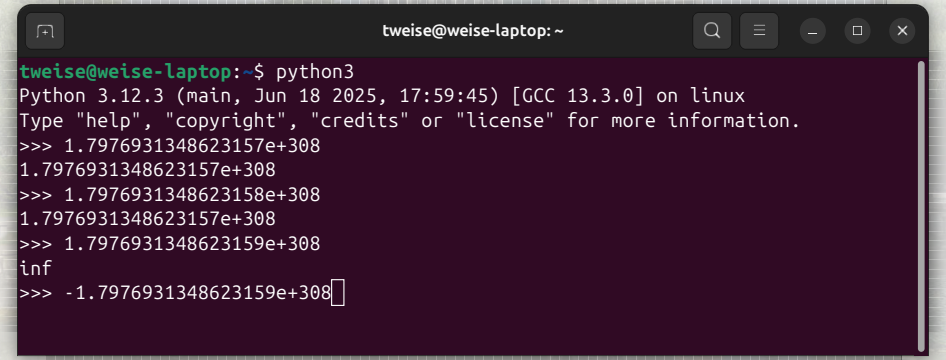
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623159e+308` ein, was geringfügig größer als die in der Java-Dokumentation genannte Zahl ist und Python gibt `inf` aus.
- `inf` ist eine Konstante aus dem Modul `math` die bedeutet *„zu groß, um mit einem `float` als Zahl dargestellt zu werden“*, was unendlich ($+\infty$) einschließt, aber eben auch einfach alle Zahlen größer als `1.7976931348623158e+308`.

```
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.
>>> 1.7976931348623157e+308
1.7976931348623157e+308
>>> 1.7976931348623158e+308
1.7976931348623157e+308
>>> 1.7976931348623159e+308
inf
>>>
```

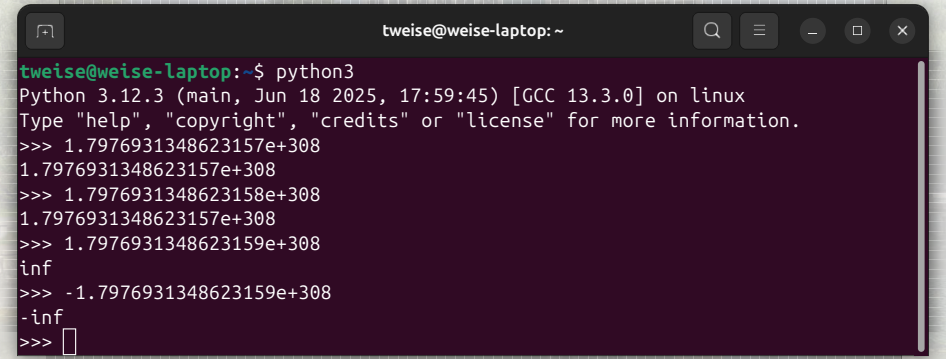
Sehr große Zahlen: Ausprobieren

- Wir geben `-1.7976931348623159e+308` ein...



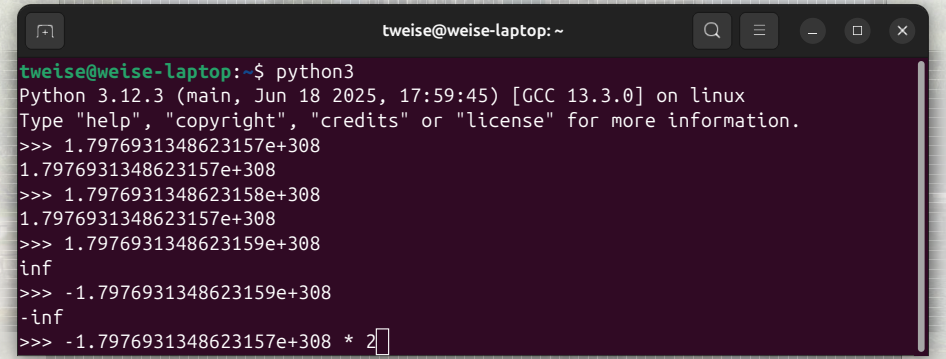
Sehr große Zahlen: Ausprobieren

- Wir geben `-1.7976931348623159e+308` ein und bekommen `-inf` angezeigt.



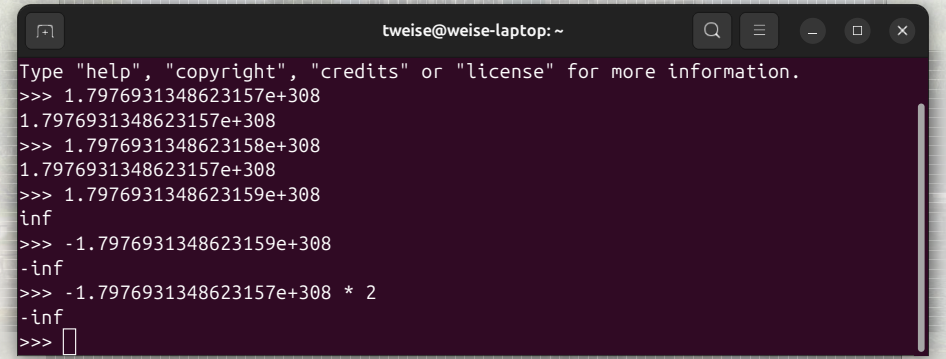
Sehr große Zahlen: Ausprobieren

- Wenn wir rechnen `-1.7976931348623157e+308 * 2`...



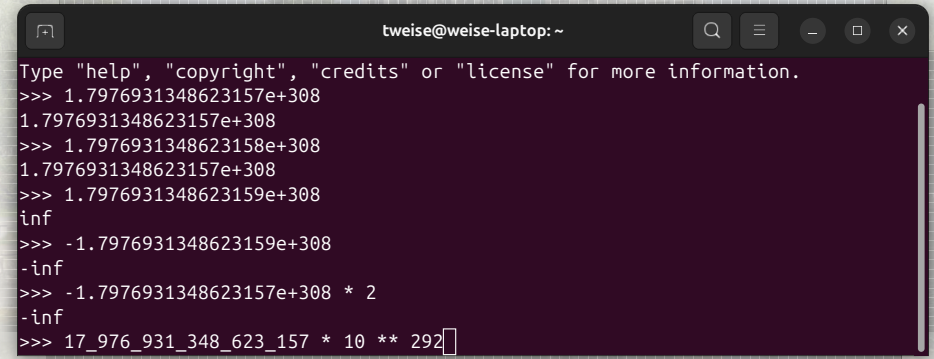
Sehr große Zahlen: Ausprobieren

- Wenn wir rechnen `-1.7976931348623157e+308 * 2`, dann ergibt das ebenfalls `-inf`.



Sehr große Zahlen: Ausprobieren

- `int` hat einen unbegrenzten Wertebereich, wir können also `1.7976931348623157e+308` als `int` schreiben...



Sehr große Zahlen: Ausprobieren

- `int` hat einen unbegrenzten Wertebereich, wir können also `1.7976931348623157e+308` als `int` schreiben und das geht problemlos.

```
>>> 1.7976931348623159e+308
inf
>>> -1.7976931348623159e+308
-inf
>>> -1.7976931348623157e+308 * 2
-inf
>>> 17_976_931_348_623_157 * 10 ** 292
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> 
```

Sehr große Zahlen: Ausprobieren

- Diesen `int` können wir in einen `float` umwandeln, in dem wir ihn mit `1.0` multiplizieren. . .



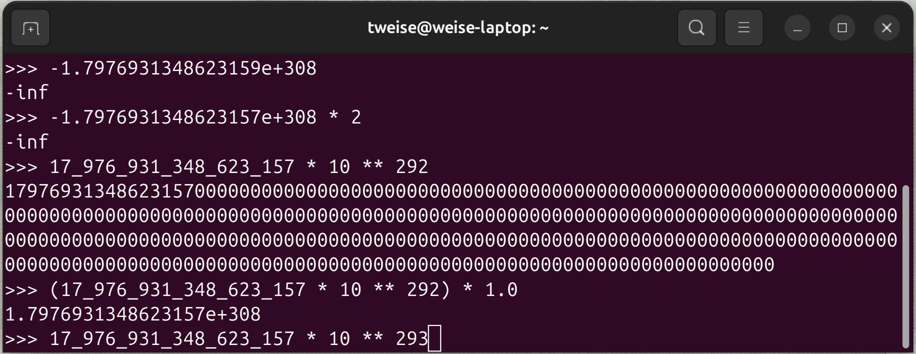
Sehr große Zahlen: Ausprobieren

- Diesen `int` können wir in einen `float` umwandeln, in dem wir ihn mit `1.0` multiplizieren und es kommt `1.7976931348623157e+308` heraus.

```
tweise@weise-laptop: ~
>>> -1.7976931348623159e+308
-inf
>>> -1.7976931348623157e+308 * 2
-inf
>>> 17_976_931_348_623_157 * 10 ** 292
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
1.7976931348623157e+308
>>> 
```

Sehr große Zahlen: Ausprobieren

- Wir schreiben jetzt eine Zahl, die zehnmal so groß ist, in dem wir mit `10 ** 293` anstelle von `10 ** 292` multiplizieren...



Sehr große Zahlen: Ausprobieren

- Wir schreiben jetzt eine Zahl, die zehnmal so groß ist, in dem wir mit `10 ** 293` anstelle von `10 ** 292` multiplizieren und das geht problemlos.

tweise@weise-laptop: ~

```
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> 
```

Sehr große Zahlen: Ausprobieren

- Diesen `int` können wir versuchen in einen `float` umwandeln, in dem wir ihn mit `1.0` multiplizieren...

```
tweise@weise-laptop: ~
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
```

Sehr große Zahlen: Ausprobieren

- Diesen `int` können wir versuchen in einen `float` umwandeln, in dem wir ihn mit `1.0` multiplizieren und Python sagt: „Nein, könnt Ihr nicht.“. (`Exceptions` lernen wir später.)

tweise@weise-laptop: ~

```
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
1.797693134862315 7e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> 
```

Sehr große Zahlen: Ausprobieren

- Importieren wir mal `e` und `log` aus dem Modul `math` um mehr Unsinn zu machen....

```
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
```

Sehr große Zahlen: Ausprobieren

- Importieren wir mal `e` und `log` aus dem Modul `math` um mehr Unsinn zu machen. Und das geht problemlos.

tweise@weise-laptop: ~

```
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
>>> 
```

Sehr große Zahlen: Ausprobieren

- Die größte `float`-Zahl, die noch geht, war `1.7976931348623157e+308`. Rechnen wir mal `log(1.7976931348623157e+308)` aus. . .

```
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
>>> log(1.7976931348623157e+308)
```

Sehr große Zahlen: Ausprobieren

- Die größte `float`-Zahl, die noch geht, war `1.7976931348623157e+308`. Rechnen wir mal `log(1.7976931348623157e+308)` aus, was `709.782712893384` ergibt.

tweise@weise-laptop: ~

```
1797693134862315700000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
>>> log(1.7976931348623157e+308)
709.782712893384
>>> 
```

Sehr große Zahlen: Ausprobieren

- `e ** 709.782712893384` sollte also in etwa wieder die größte Zahl ergeben...

```
tweise@weise-laptop: ~
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
>>> log(1.7976931348623157e+308)
709.782712893384
>>> e ** 709.782712893384
```

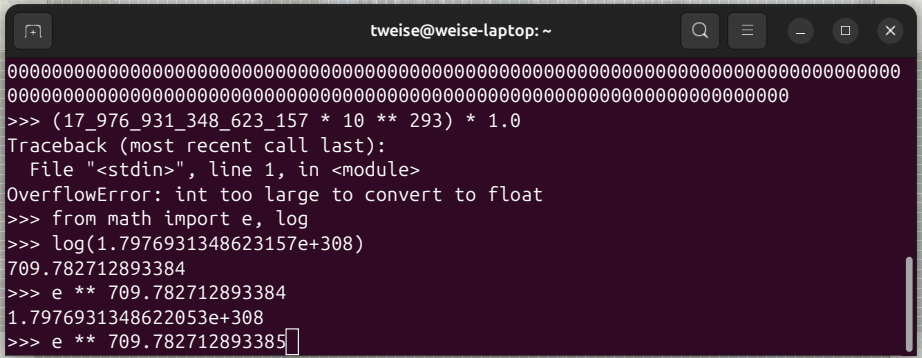
Sehr große Zahlen: Ausprobieren

- `e ** 709.782712893384` sollte also in etwa wieder die größte Zahl ergeben. `1.7976931348622053e+308` liegt nahe genug dran ... wie gesagt, wir haben begrenzte Genauigkeit mit `floats`.

```
tweise@weise-laptop: ~
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
>>> log(1.7976931348623157e+308)
709.782712893384
>>> e ** 709.782712893384
1.7976931348622053e+308
>>>
```

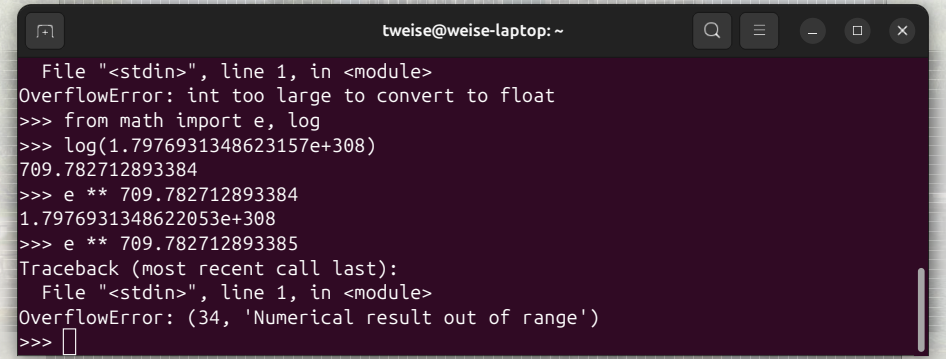
Sehr große Zahlen: Ausprobieren

- Was passiert, wenn wir `e ** 709.782712893385` anstatt `e ** 709.782712893384` versuchen?



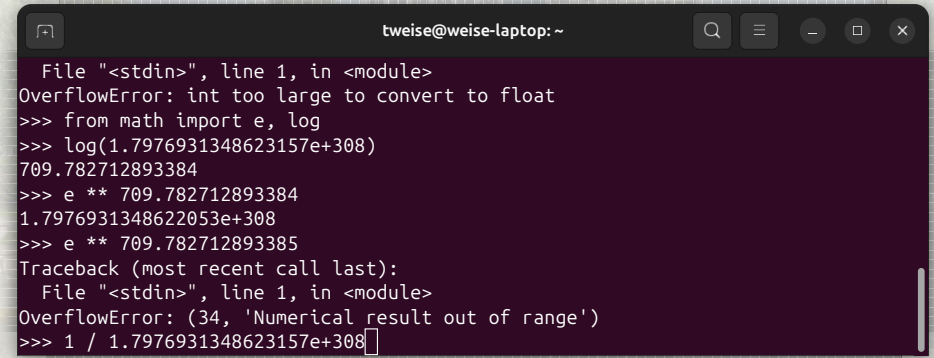
Sehr große Zahlen: Ausprobieren

- Was passiert, wenn wir `e ** 709.782712893385` anstatt `e ** 709.782712893384` versuchen? Dann sagt Python: „Nö. Macht Ihr nicht.“



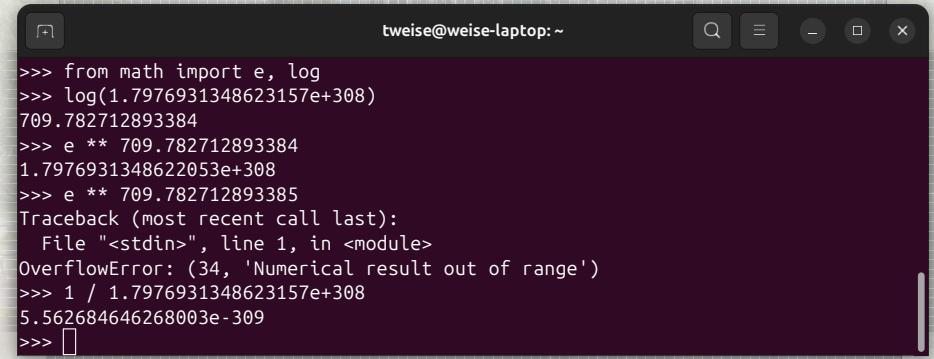
Sehr große Zahlen: Ausprobieren

- Wenn `1.7976931348623157e+308` die größte Zahl ist, was ist dann `1 / 1.7976931348623157e+308`?



Sehr große Zahlen: Ausprobieren

- Wenn `1.7976931348623157e+308` die größte Zahl ist, was ist dann `1 / 1.7976931348623157e+308`?. `5.562684646268003e-309`, was schon sehr klein ist.



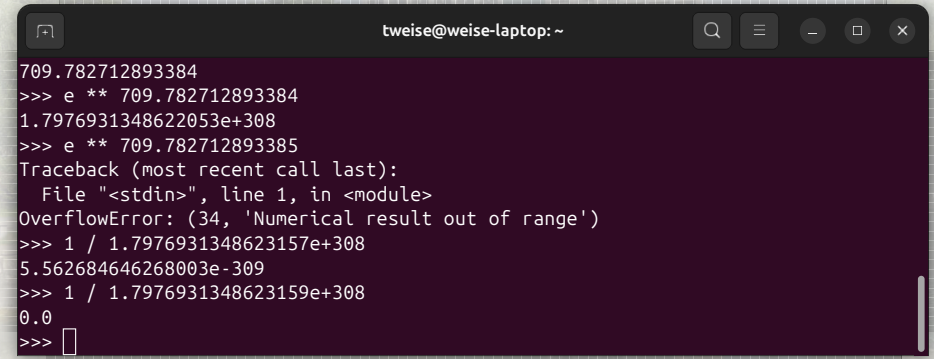
Sehr große Zahlen: Ausprobieren

- Wenn `1.7976931348623159e+308` zu `inf` wird, was passiert mit `1 / 1.7976931348623159e+308`?



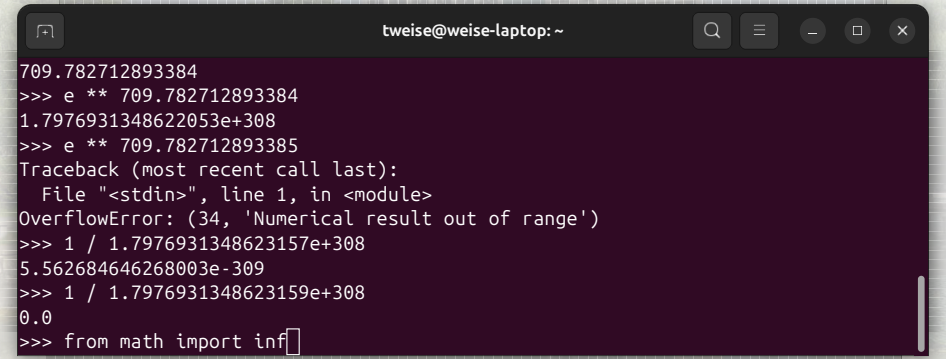
Sehr große Zahlen: Ausprobieren

- Wenn `1.7976931348623159e+308` zu `inf` wird, was passiert mit `1 / 1.7976931348623159e+308`?. Es wird zu `0.0`.



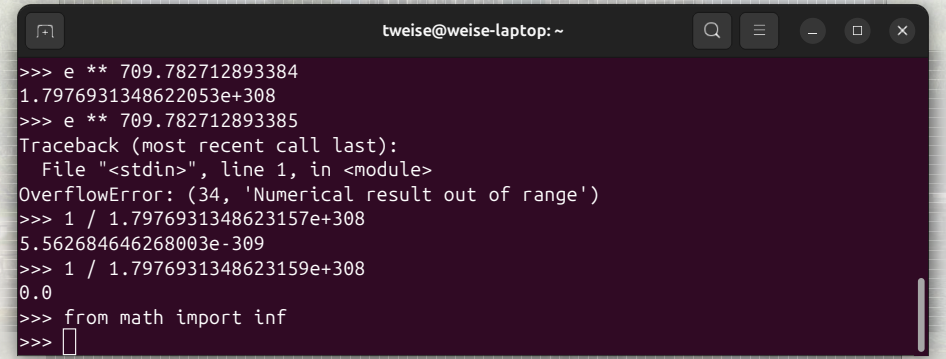
Sehr große Zahlen: Ausprobieren

- Importieren wir mal `inf` aus dem Modul `math`...



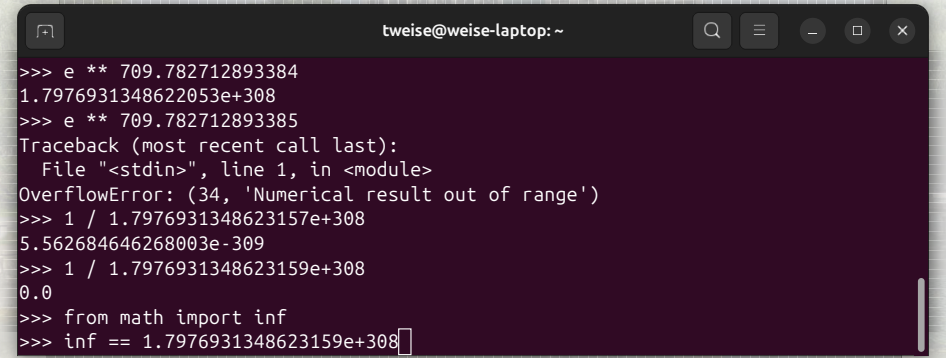
Sehr große Zahlen: Ausprobieren

- Importieren wir mal `inf` aus dem Modul `math` Und das geht problemlos.



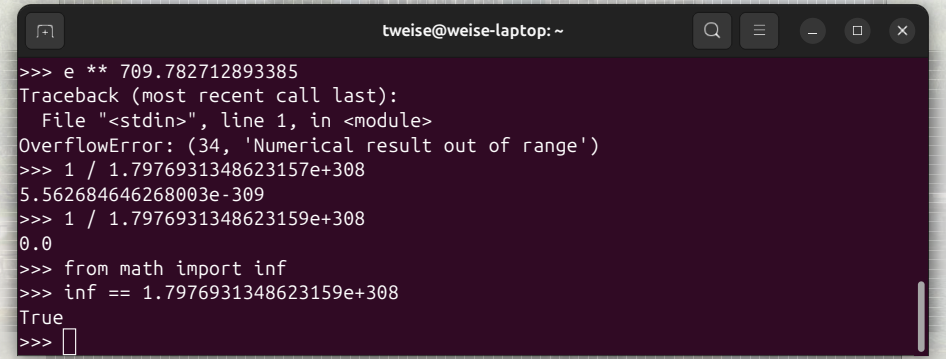
Sehr große Zahlen: Ausprobieren

- Wenn `1.7976931348623159e+308` zu `inf` wird, was ist dann `1.7976931348623159e+308 == inf`?



Sehr große Zahlen: Ausprobieren

- Wenn `1.7976931348623159e+308` zu `inf` wird, was ist dann `1.7976931348623159e+308 == inf`? Es ist `True`, also „Wahr“.



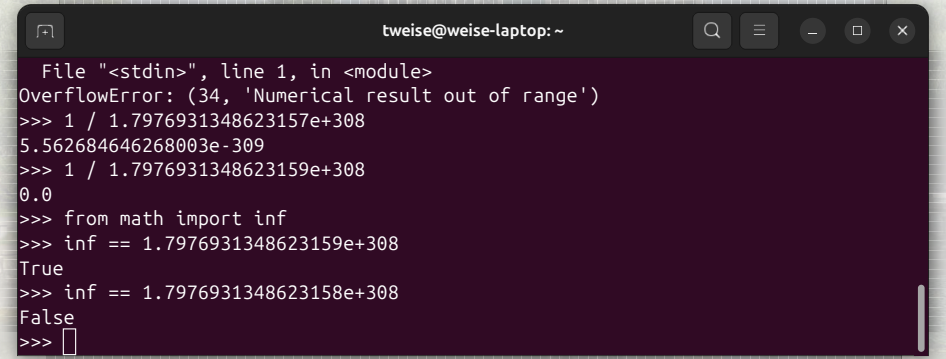
Sehr große Zahlen: Ausprobieren

- `1.7976931348623158e+308` ist die größte Zahl, die noch „geht“. Was ist dann `1.7976931348623158e+308 == inf`?



Sehr große Zahlen: Ausprobieren

- `1.7976931348623158e+308` ist die größte Zahl, die noch „geht“. Was ist dann `1.7976931348623158e+308 == inf`?. Es ist `False`, also „Falsch“.



Sehr große Zahlen: Ausprobieren



- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.
- Es bedeutet nicht unendlich.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.
- Es bedeutet nicht unendlich.
- $1.797\,693\,134\,862\,315\,9 * 10^{308}$ ist nämlich nicht das gleiche wie $+\infty$.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.
- Es bedeutet nicht unendlich.
- $1.797\,693\,134\,862\,315\,9 * 10^{308}$ ist nämlich nicht das gleiche wie $+\infty$.
- `inf` verhält sich aber in einigen Rechnungen ähnlich wie unendlich.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.
- Es bedeutet nicht unendlich.
- $1.797\,693\,134\,862\,315\,9 * 10^{308}$ ist nämlich nicht das gleiche wie $+\infty$.
- `inf` verhält sich aber in einigen Rechnungen ähnlich wie unendlich.
- Ein sogenannter Überlauf bei einer Berechnung kann zu `inf` führen, manchmal aber auch zu einem Fehler der die Berechnung direkt stoppt.

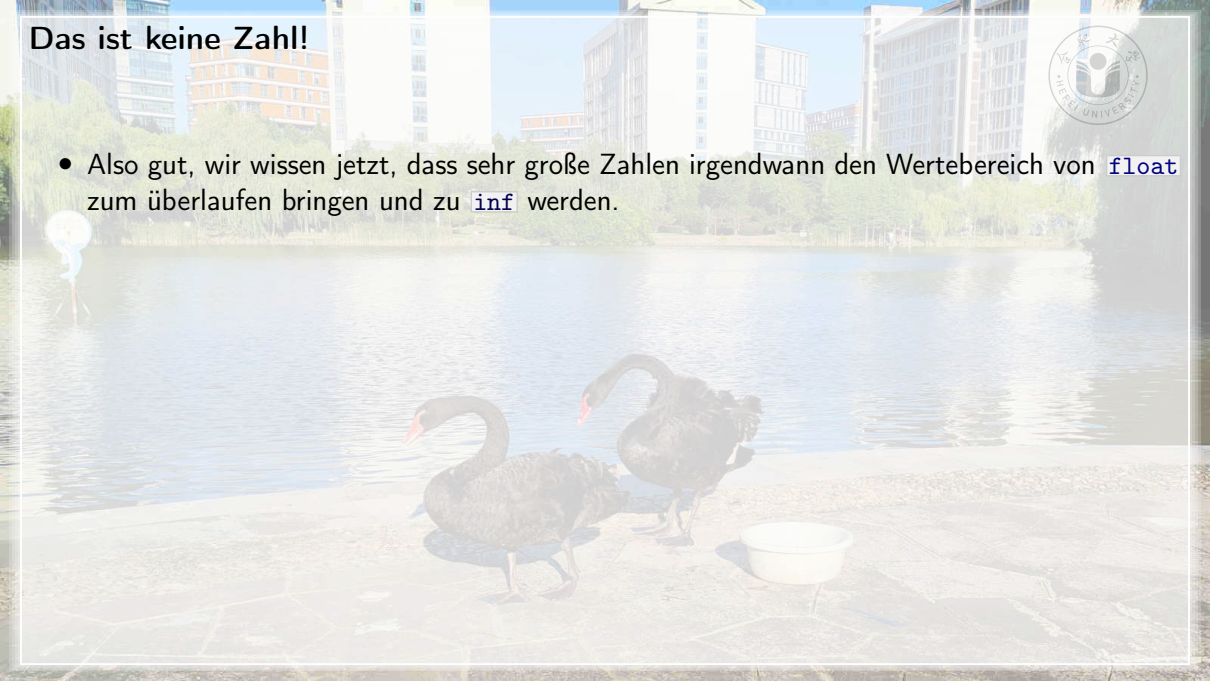
Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.
- Es bedeutet nicht unendlich.
- $1.797\,693\,134\,862\,315\,9 * 10^{308}$ ist nämlich nicht das gleiche wie $+\infty$.
- `inf` verhält sich aber in einigen Rechnungen ähnlich wie unendlich.
- Ein sogenannter Überlauf bei einer Berechnung kann zu `inf` führen, manchmal aber auch zu einem Fehler der die Berechnung direkt stoppt.
- Solche Fehler sind eigentlich die bessere Variante: Es gibt sehr wenige echte Berechnungen, bei denen `inf` ein vernünftiges, erwartetes Ergebnis ist.

Sehr große Zahlen: Ausprobieren

- Während `0` der natürliche Grenzwert für kleine Zahlen ist, definiert Python die Konstante `inf` als Wert für Zahlen, die *zu groß* sind.
- Nun ist 0 eine verständliche Konstante mit klaren mathematischen Eigenschaften.
- `inf` ist viel verwirrender.
- Es bedeutet nicht unendlich.
- $1.797\,693\,134\,862\,315\,9 * 10^{308}$ ist nämlich nicht das gleiche wie $+\infty$.
- `inf` verhält sich aber in einigen Rechnungen ähnlich wie unendlich.
- Ein sogenannter Überlauf bei einer Berechnung kann zu `inf` führen, manchmal aber auch zu einem Fehler der die Berechnung direkt stoppt.
- Solche Fehler sind eigentlich die bessere Variante: Es gibt sehr wenige echte Berechnungen, bei denen `inf` ein vernünftiges, erwartetes Ergebnis ist.
- Es ist also sogar besser, wenn die Berechnung einfach mit einem Fehler abbricht ... dann wissen wir wenigstens, das etwas schief gelaufen ist (vielleicht waren die Eingabewerte ja falsch...)

Das ist keine Zahl!



- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.

Das ist keine Zahl!

- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.

Das ist keine Zahl!

- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.
- Das sind sehr viele Zahlen.

Das ist keine Zahl!

- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.
- Das sind sehr viele Zahlen.
- Es schließt auch $+\infty$, also unendlich, mit ein.

Das ist keine Zahl!

- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.
- Das sind sehr viele Zahlen.
- Es schließt auch $+\infty$, also unendlich, mit ein.
- Wenn wir mit `inf` rechnen (was eigentlich selten eine gute Idee ist), dann verhalten sich die meisten Funktionen so, als wäre es $+\infty$.

Das ist keine Zahl!



- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.
- Das sind sehr viele Zahlen.
- Es schließt auch $+\infty$, also unendlich, mit ein.
- Wenn wir mit `inf` rechnen (was eigentlich selten eine gute Idee ist), dann verhalten sich die meisten Funktionen so, als wäre es $+\infty$.
- Was bedeutet das?

Das ist keine Zahl!

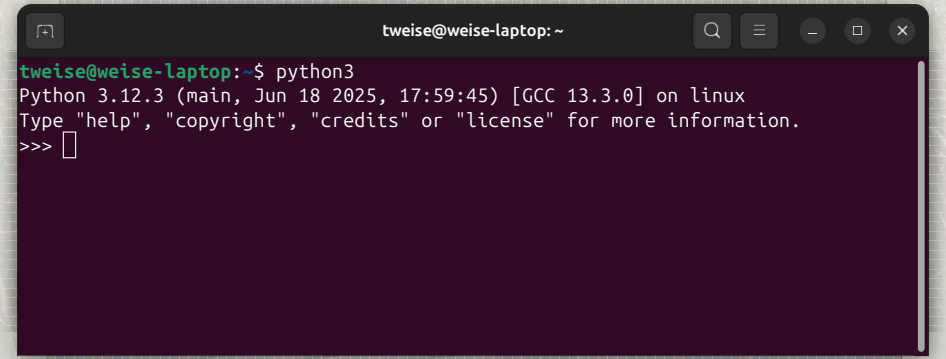


- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.
- Das sind sehr viele Zahlen.
- Es schließt auch $+\infty$, also unendlich, mit ein.
- Wenn wir mit `inf` rechnen (was eigentlich selten eine gute Idee ist), dann verhalten sich die meisten Funktionen so, als wäre es $+\infty$.
- Was bedeutet das?
- Und was passiert, wenn wir ganz komische Sachen mit `inf` machen?

Das ist keine Zahl! – Ausprobieren

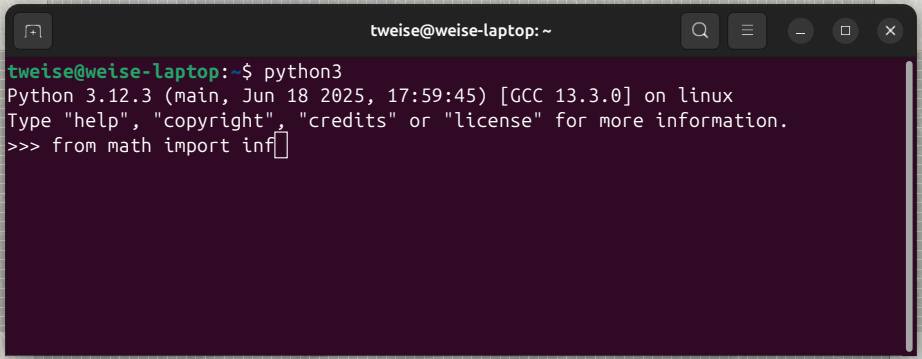


- Probieren wir das mal aus: Rechnen wild mit `inf` und gucken, was passiert.



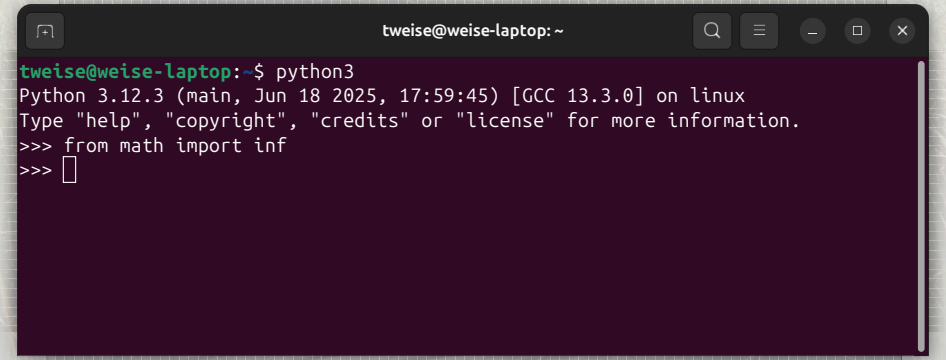
Das ist keine Zahl! – Ausprobieren

- Zuerst importieren wir wieder `inf` aus dem Modul `math`...



Das ist keine Zahl! – Ausprobieren

- Zuerst importieren wir wieder `inf` aus dem Modul `math` Und das geht problemlos.



Das ist keine Zahl! – Ausprobieren



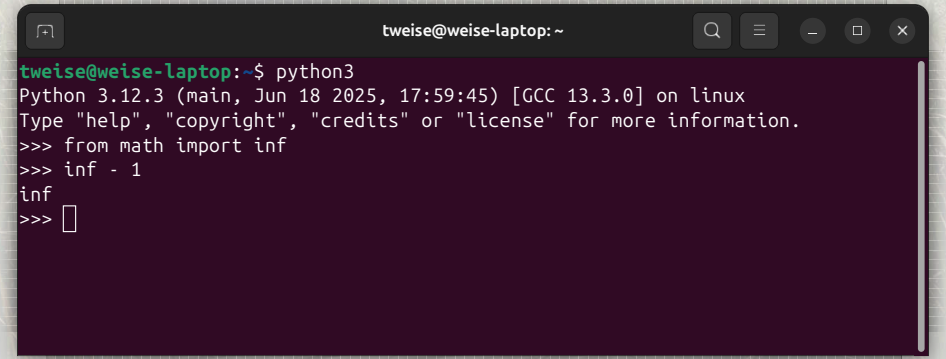
- Fangen wir einfach an: `inf - 1`...

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.
>>> from math import inf
>>> inf - 1
```

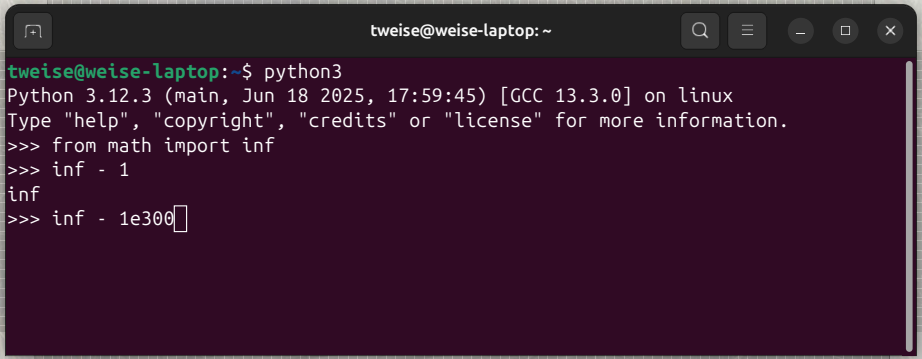
Das ist keine Zahl! – Ausprobieren

- Fangen wir einfach an: `inf - 1` bleibt natürlich `inf`.



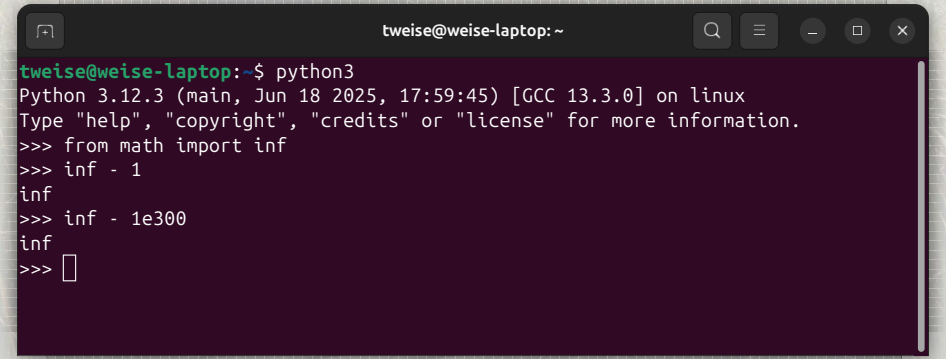
Das ist keine Zahl! – Ausprobieren

- `inf - 1e300` ist eigentlich schon etwas komisch, weil wir ja auch zu `inf` über eine sehr große Zahl, z. B. `1e310` hätten gekommen seien können...



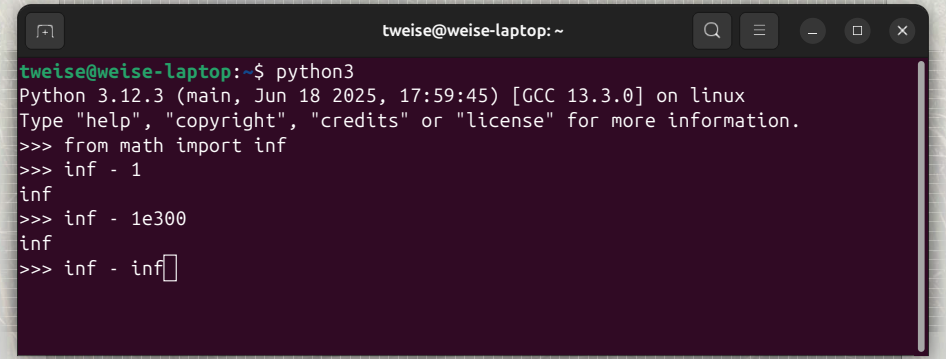
Das ist keine Zahl! – Ausprobieren

- `inf - 1e300` ist eigentlich schon etwas komisch, weil wir ja auch zu `inf` über eine sehr große Zahl, z. B. `1e310` hätten gekommen seien können. Aber das Ergebnis bleibt `inf`, denn `inf` verhält sich hier wie $+\infty$.



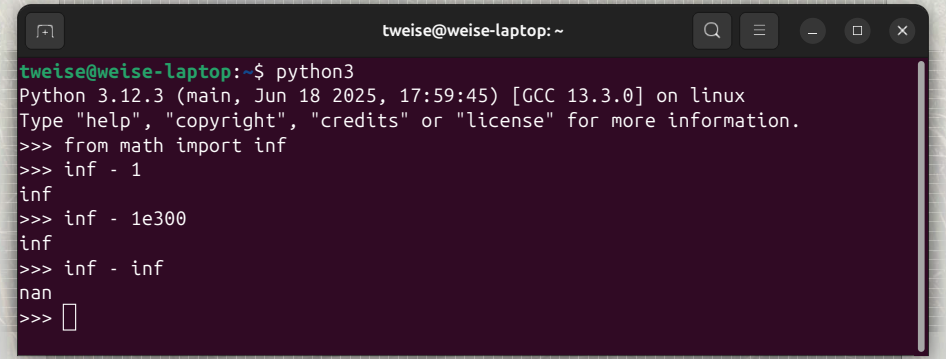
Das ist keine Zahl! – Ausprobieren

- Dann machen wir doch `inf - inf`...



Das ist keine Zahl! – Ausprobieren

- Dann machen wir doch `inf - inf`; `inf` verhält sich wie $+\infty$, aber $\infty - \infty$ ist nicht definiert.



Das ist keine Zahl! – Ausprobieren

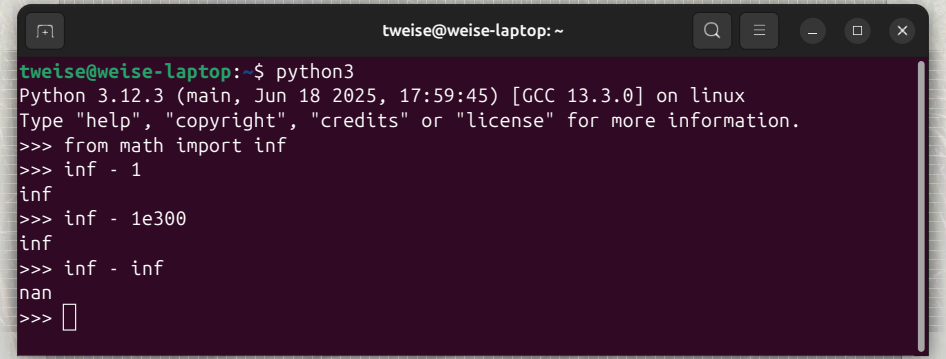


- Dann machen wir doch `inf - inf`; `inf` verhält sich wie $+\infty$, aber $\infty - \infty$ ist nicht definiert.
- `nan` bedeutet *„Not a Number“* also *„Das ist keine Zahl!“*[39].

```
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.
>>> from math import inf
>>> inf - 1
inf
>>> inf - 1e300
inf
>>> inf - inf
nan
>>>
```

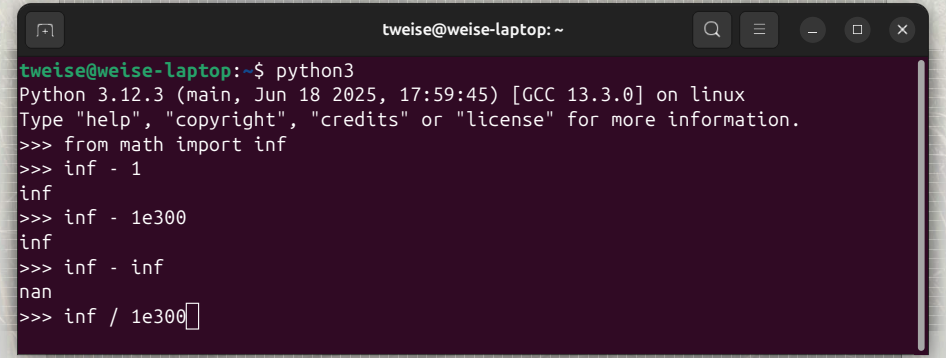
Das ist keine Zahl! – Ausprobieren

- Dann machen wir doch `inf - inf`; `inf` verhält sich wie $+\infty$, aber $\infty - \infty$ ist nicht definiert.
- `nan` bedeutet *„Not a Number“* also *„Das ist keine Zahl!“*[39].
- Kommt irgendwo in einer Rechnung `nan` vor, dann ist das Ergebnis immer auch `nan`.



Das ist keine Zahl! – Ausprobieren

- Nagut. Probieren wir `inf / 1e300`...



Das ist keine Zahl! – Ausprobieren

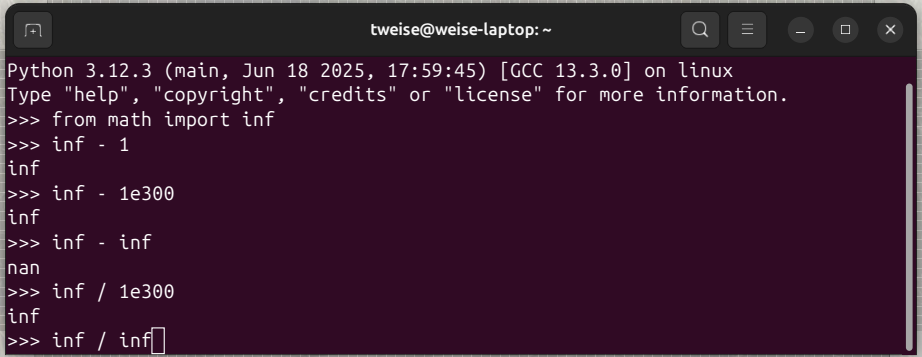


- Nagut. Probieren wir `inf / 1e300`, was wieder `inf` ergibt.

```
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.
>>> from math import inf
>>> inf - 1
inf
>>> inf - 1e300
inf
>>> inf - inf
nan
>>> inf / 1e300
inf
>>>
```

Das ist keine Zahl! – Ausprobieren

- Wie wäre es dann mit `inf / inf`?



Das ist keine Zahl! – Ausprobieren

- Wie wäre es dann mit `inf / inf`? Das ist nicht definiert und ergibt `nan`[39].



Das ist keine Zahl! – Ausprobieren



- Und was ist mit `inf * 0`? Das ist auch nicht definiert und ergibt `nan`[39].



Das ist keine Zahl! – Ausprobieren



- Und was ist mit `inf * 0`? Das ist auch nicht definiert und ergibt `nan`[39].



Das ist keine Zahl! – Ausprobieren

- Wir wollen direkt mit `nan` herumspielen und importieren daher die Konstante `nan` aus dem Modul `math`...



Das ist keine Zahl! – Ausprobieren

- Wir wollen direkt mit `nan` herumspielen und importieren daher die Konstante `nan` aus dem Modul `math`. Und schwupps haben wir sie importiert.



Das ist keine Zahl! – Ausprobieren

- Nun können wir `nan + 1` rechnen...



Das ist keine Zahl! – Ausprobieren

- Nun können wir `nan + 1` rechnen, was wieder `nan` ergibt.



Das ist keine Zahl! – Ausprobieren



- Aber was ist `nan + inf`?

tweise@weise-laptop: ~

```
>>> inf - inf
nan
>>> inf / 1e300
inf
>>> inf / inf
nan
>>> 0 * inf
nan
>>> from math import nan
>>> nan + 1
nan
>>> nan + inf
```

Das ist keine Zahl! – Ausprobieren



- Aber was ist `nan + inf`? Wieder `nan`. Es gibt kein Entkommen.

tweise@weise-laptop: ~

```
>>> inf / 1e300
inf
>>> inf / inf
nan
>>> 0 * inf
nan
>>> from math import nan
>>> nan + 1
nan
>>> nan + inf
nan
>>>
```

Das ist keine Zahl! – Ausprobieren



- Was genau ist eigentlich `nan`? Ist es gleich 1?

tweise@weise-laptop: ~

```
>>> inf / 1e300
inf
>>> inf / inf
nan
>>> 0 * inf
nan
>>> from math import nan
>>> nan + 1
nan
>>> nan + inf
nan
>>> nan == 1
```

Das ist keine Zahl! – Ausprobieren



- Was genau ist eigentlich `nan`? Ist es gleich 1? Nein, natürlich nicht.



Das ist keine Zahl! – Ausprobieren

- Was genau ist eigentlich `nan`? Ist es ungleich 1?



Das ist keine Zahl! – Ausprobieren



- Was genau ist eigentlich `nan`? Ist es ungleich 1? Ja. Na klar.

tweise@weise-laptop: ~

```
>>> 0 * inf
nan
>>> from math import nan
>>> nan + 1
nan
>>> nan + inf
nan
>>> nan == 1
False
>>> nan != 1
True
>>>
```

Das ist keine Zahl! – Ausprobieren



- Was genau ist eigentlich `nan`? Ist `nan` gleich `nan`?

tweise@weise-laptop: ~

```
>>> 0 * inf
nan
>>> from math import nan
>>> nan + 1
nan
>>> nan + inf
nan
>>> nan == 1
False
>>> nan != 1
True
>>> nan == nan
```

Das ist keine Zahl! – Ausprobieren



- Was genau ist eigentlich `nan`? Ist `nan` gleich `nan`? NEIN! `nan` ist der *einzige* `float`-Wert, der ungleich seiner selbst ist!



Das ist keine Zahl! – Ausprobieren



- Dann gilt also `nan` $\neq$ `nan`?

tweise@weise-laptop: ~

```
>>> from math import nan
>>> nan + 1
nan
>>> nan + inf
nan
>>> nan == 1
False
>>> nan != 1
True
>>> nan == nan
False
>>> nan != nan
```

Das ist keine Zahl! – Ausprobieren



- Dann gilt also `nan` $\neq$ `nan`? JA! Wie gesagt, `nan` ist der *einzige* `float`-Wert, der ungleich seiner selbst ist!



Das ist keine Zahl! – Ausprobieren



- Also `inf` = `inf` gilt aber?

tweise@weise-laptop: ~

```
nan
>>> nan + inf
nan
>>> nan == 1
False
>>> nan != 1
True
>>> nan == nan
False
>>> nan != nan
True
>>> inf == inf
```

Das ist keine Zahl! – Ausprobieren



- Also `inf` = `inf` gilt aber? Ja. Was komisch ist, weil `inf - inf` ja nicht definiert ist...

```
tweise@weise-laptop: ~
nan
>>> nan == 1
False
>>> nan != 1
True
>>> nan == nan
False
>>> nan != nan
True
>>> inf == inf
True
>>>
```

Das ist keine Zahl! – Ausprobieren



- Und `inf` $\neq$ `inf` stimmt dann aber nicht?

tweise@weise-laptop: ~

```
nan
>>> nan == 1
False
>>> nan != 1
True
>>> nan == nan
False
>>> nan != nan
True
>>> inf == inf
True
>>> inf != inf
```

Das ist keine Zahl! – Ausprobieren



- Und `inf` $\neq$ `inf` stimmt dann aber nicht? Ja. Was immer noch komisch ist, weil `inf - inf` ja nicht definiert ist...

```
tweise@weise-laptop: ~
False
>>> nan != 1
True
>>> nan == nan
False
>>> nan != nan
True
>>> inf == inf
True
>>> inf != inf
False
>>>
```

Das ist keine Zahl! – Ausprobieren



- `nan` ist eine Konstante, die für „undefiniert“ steht.

Das ist keine Zahl! – Ausprobieren



- `nan` ist eine Konstante, die für „undefiniert“ steht.
- Genauso, wie ein `float`-Wert irgendwo in einer Rechnunung mit `int`s die Berechnung „infiziert“ und das Ergebnis dann immer ein `float` ist...

Das ist keine Zahl! – Ausprobieren



- `nan` ist eine Konstante, die für „undefiniert“ steht.
- Genauso, wie ein `float`-Wert irgendwo in einer Rechnunung mit `int`s die Berechnung „infiziert“ und das Ergebnis dann immer ein `float` ist, so „infiziert“ `nan` jede Berechnung so dass das Ergebnis immer `nan` wird.

Das ist keine Zahl! – Ausprobieren



- `nan` ist eine Konstante, die für „undefiniert“ steht.
- Genauso, wie ein `float`-Wert irgendwo in einer Rechnunung mit `int`s die Berechnung „infiziert“ und das Ergebnis dann immer ein `float` ist, so „infiziert“ `nan` jede Berechnung so dass das Ergebnis immer `nan` wird.
- Es gilt `nan != nan`.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)
  2. Vielleicht sind ja die Eingabedaten falsch. Die Zahlen mit denen wir rechnen stimmen nicht.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)
  2. Vielleicht sind ja die Eingabedaten falsch. Die Zahlen mit denen wir rechnen stimmen nicht.
  3. Vielleicht ist auch unsere Formel falsch. Vielleicht haben wir uns ja vertippt.

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)
  2. Vielleicht sind ja die Eingabedaten falsch. Die Zahlen mit denen wir rechnen stimmen nicht.
  3. Vielleicht ist auch unsere Formel falsch. Vielleicht haben wir uns ja vertippt.
  4. Vielleicht hat jemand mit Absicht falsche Daten eingegeben, damit sich unser Program falsch verhält? Werte wie `nan` können durchaus ein Problem für den Systemschutz (Security) sein[24].

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)
  2. Vielleicht sind ja die Eingabedaten falsch. Die Zahlen mit denen wir rechnen stimmen nicht.
  3. Vielleicht ist auch unsere Formel falsch. Vielleicht haben wir uns ja vertippt.
  4. Vielleicht hat jemand mit Absicht falsche Daten eingegeben, damit sich unser Program falsch verhält? Werte wie `nan` können durchaus ein Problem für den Systemschutz (Security) sein[24].

Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)
  2. Vielleicht sind ja die Eingabedaten falsch. Die Zahlen mit denen wir rechnen stimmen nicht.
  3. Vielleicht ist auch unsere Formel falsch. Vielleicht haben wir uns ja vertippt.
  4. Vielleicht hat jemand mit Absicht falsche Daten eingegeben, damit sich unser Program falsch verhält? Werte wie `nan` können durchaus ein Problem für den Systemschutz (Security) sein[24].
- So oder so: Wir brauchen verlässliche Möglichkeiten, alle „komischen“ `float`-Werte zu erkennen und abzufangen.

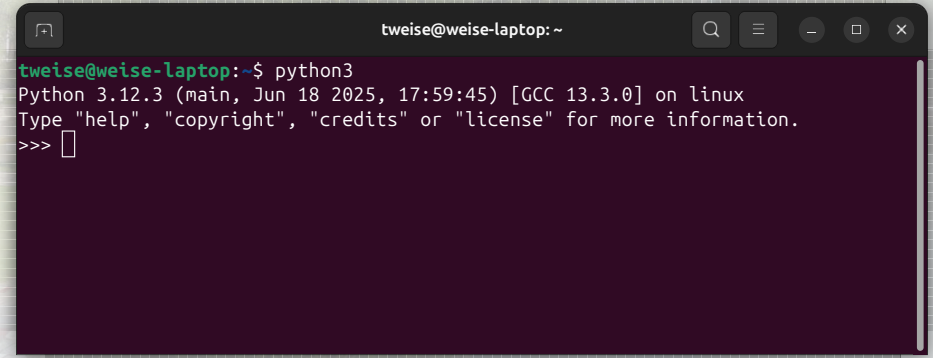
Wo kommen `inf` und `nan` her?



- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `inf` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind sie ja genau das Ergebnis, das herauskommen soll. Das korrekte Ergebnis einer korrekten Berechnung mit korrekten Eingabedaten. Genau das was wir wollten. (Klingt das sehr wahrscheinlich?)
  2. Vielleicht sind ja die Eingabedaten falsch. Die Zahlen mit denen wir rechnen stimmen nicht.
  3. Vielleicht ist auch unsere Formel falsch. Vielleicht haben wir uns ja vertippt.
  4. Vielleicht hat jemand mit Absicht falsche Daten eingegeben, damit sich unser Program falsch verhält? Werte wie `nan` können durchaus ein Problem für den Systemschutz (Security) sein[24].
- So oder so: Wir brauchen verlässliche Möglichkeiten, alle „komischen“ `float`-Werte zu erkennen und abzufangen.
- Dafür gibt es im modul `math` die Funktionen `isfinite`, `isinf`, und `isnan`.

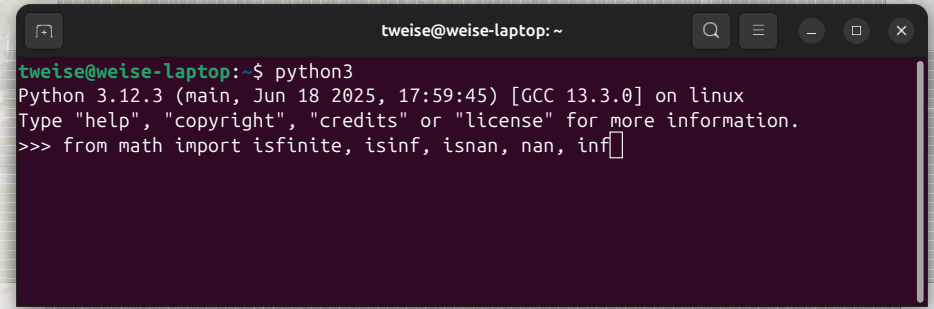
Testen auf inf und nan: Ausprobieren

- Probieren wir das mal aus.



Testen auf inf und nan: Ausprobieren

- Zuerst importieren wir alle Funktionen und Konstanten, die wir brauchen, vom Modul `math`...



Testen auf inf und nan: Ausprobieren

- Zuerst importieren wir alle Funktionen und Konstanten, die wir brauchen, vom Modul `math`. Und schwupps haben wir sie importiert.



Testen auf inf und nan: Ausprobieren

- Mit `isfinite` prüfen wir, ob eine Zahl finite ist, also weder unendlich noch undefiniert...



Testen auf inf und nan: Ausprobieren

- Mit `isfinite` prüfen wir, ob eine Zahl finite ist, also weder unendlich noch undefiniert. `isfinite(1e34)` ist daher `True`.



Testen auf inf und nan: Ausprobieren

- Der Test `isfinite(inf)` jedoch...



Testen auf inf und nan: Ausprobieren

- Der Test `isfinite(inf)` jedoch schlägt fehl und ergibt `False`.



Testen auf inf und nan: Ausprobieren

- Der Test `isfinite(nan)`...



Testen auf inf und nan: Ausprobieren

- Der Test `isfinite(nan)` ergibt ebenfalls `False`.



Testen auf inf und nan: Ausprobieren

- `isinf` prüft, ob eine Zahl `inf` oder `-inf` ist. Der Test `isinf(0.3)`...



Testen auf inf und nan: Ausprobieren

- `isinf` prüft, ob eine Zahl `inf` oder `-inf` ist. Der Test `isinf(0.3)` ist natürlich `False`.



Testen auf inf und nan: Ausprobieren

- `isinf` prüft, ob eine Zahl `inf` oder `-inf` ist. Der Test `isinf(-inf)` ...



Testen auf inf und nan: Ausprobieren

- `isinf` prüft, ob eine Zahl `inf` oder `-inf` ist. Der Test `isinf(-inf)` ist folgerichtig `True`.



Testen auf inf und nan: Ausprobieren

- `nan` ist undefiniert. Es hat gar keinen Wert oder auch jeden beliebigen Wert. Der Test `isinf(nan)` ...



Testen auf inf und nan: Ausprobieren

- `nan` ist undefiniert. Es hat gar keinen Wert oder auch jeden beliebigen Wert. Der Test `isinf(nan)` ist deshalb `False`.



Testen auf inf und nan: Ausprobieren

- `isnan` prüft, ob ein Wert `nan` ist, also undefiniert. `isnan(0.3455)`...



Testen auf inf und nan: Ausprobieren

- `isnan` prüft, ob ein Wert `nan` ist, also undefiniert. `isnan(0.3455)` ist `False`, weil 0.3455 natürlich definiert ist.



Testen auf inf und nan: Ausprobieren

- Fragen wir ob `isnan(inf)`...



Testen auf inf und nan: Ausprobieren

- Fragen wir ob `isnan(inf)` dann ist die Antwort `False`. `inf` ist zwar auch keine Zahl (not a number, pardon the pun), aber definiert, also nicht nicht-definiert.



Testen auf inf und nan: Ausprobieren

- Interessanterweise ist ja `nan == nan` `False` und `nan != nan` PythonTrue. `isnan` nützlich, weil es sich leichter liest als der Test `x != x` für eine Zahl `x`. `isnan(nan)` ergibt...



Testen auf inf und nan: Ausprobieren

- Interessanterweise ist ja `nan == nan` `False` und `nan != nan` PythonTrue. `isnan` nützlich, weil es sich leichter liest als der Test `x != x` für eine Zahl `x`. `isnan(nan)` ergibt `True`.



Testen auf inf und nan: Ausprobieren

- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.

Testen auf inf und nan: Ausprobieren



- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.

Testen auf inf und nan: Ausprobieren

- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.
- Manchmal machen wir das während unserer Berechnungen.

Testen auf inf und nan: Ausprobieren

- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.
- Manchmal machen wir das während unserer Berechnungen.
- Was wir in der Regel nicht wollen, ist Kode der solche Werte ausgibt.

Testen auf inf und nan: Ausprobieren

- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.
- Manchmal machen wir das während unserer Berechnungen.
- Was wir in der Regel nicht wollen, ist Kode der solche Werte ausgibt.
- Ein gutes Werkzeug dafür ist `isfinite`.

Testen auf inf und nan: Ausprobieren

- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.
- Manchmal machen wir das während unserer Berechnungen.
- Was wir in der Regel nicht wollen, ist Kode der solche Werte ausgibt.
- Ein gutes Werkzeug dafür ist `isfinite`.
- Normalerweise wollen wir nur mit Werten `x` arbeiten, für die `isfinite(x)== True`.

Testen auf inf und nan: Ausprobieren

- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.
- Manchmal machen wir das während unserer Berechnungen.
- Was wir in der Regel nicht wollen, ist Kode der solche Werte ausgibt.
- Ein gutes Werkzeug dafür ist `isfinite`.
- Normalerweise wollen wir nur mit Werten `x` arbeiten, für die `isfinite(x)== True`.
- Alles andere ist normalerweise ein Zeichen für einen Fehler.

Zusammenfassung

Zusammenfassung

- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.

Zusammenfassung



- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentyp `float` benutzen.

Zusammenfassung



- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentyp `float` benutzen.
- Dieser Datentyp ist nicht beliebig genau, sondern sollte immer als ungenau betrachtet werden.

Zusammenfassung

- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentyp `float` benutzen.
- Dieser Datentyp ist nicht beliebig genau, sondern sollte immer als ungenau betrachtet werden.
- Er reicht aber aus, um viele Berechnungen genau genug durchzuführen ... 15 Ziffern sind schon ziemlich viel....

Zusammenfassung



- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentyp `float` benutzen.
- Dieser Datentyp ist nicht beliebig genau, sondern sollte immer als ungenau betrachtet werden.
- Er reicht aber aus, um viele Berechnungen genau genug durchzuführen ... 15 Ziffern sind schon ziemlich viel....
- Durch runden können `floats` in `ints` überführt werden (wobei zu beachten ist, das die Funktion `round` banker's rounding durchführt).

Zusammenfassung



- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentyp `float` benutzen.
- Dieser Datentyp ist nicht beliebig genau, sondern sollte immer als ungenau betrachtet werden.
- Er reicht aber aus, um viele Berechnungen genau genug durchzuführen ... 15 Ziffern sind schon ziemlich viel....
- Durch runden können `floats` in `ints` überführt werden (wobei zu beachten ist, das die Funktion `round` banker's rounding durchführt).
- Sehr große oder kleine `float`-Werte können in der wissenschaftlichen Notation dargestellt werden.

Zusammenfassung

- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentyp `float` benutzen.
- Dieser Datentyp ist nicht beliebig genau, sondern sollte immer als ungenau betrachtet werden.
- Er reicht aber aus, um viele Berechnungen genau genug durchzuführen ... 15 Ziffern sind schon ziemlich viel....
- Durch runden können `floats` in `ints` überführt werden (wobei zu beachten ist, das die Funktion `round` banker's rounding durchführt).
- Sehr große oder kleine `float`-Werte können in der wissenschaftlichen Notation dargestellt werden.
- `inf`, `-inf`, und `nan` sind `float`-Konstanten, die für „zu groß für einen `float`“, „negativ-zu groß für einen `float`“, und „nicht definiert“ stehen.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also [2] (siehe S. 413, 427).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also [1] (siehe S. 413, 427).

[3] "Banker's Rounding". In: *Alipay$^+$ Documentation*. Singapore: Alipay Connect Pte. Lte., 2022. URL: https://docs.alipayplus.com/alipayplus/alipayplus/reconcile_mpp/bank_rounding (besucht am 2025-07-18) (siehe S. 120, 121, 425).

[4] 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. 426, 428).

[5] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 427).

[6] Gary Baumgartner, Danny Heap und Richard Krueger. "Numerical Systems". In: *Course Notes for CSC165H: Mathematical Expression and Reasoning for Computer Science*. Toronto, ON, Canada: Department of Computer Science, University of Toronto, Herbst 2006. Kap. 7. URL: https://www.cs.toronto.edu/~krueger/csc165h/f06/lectures/ch7.pdf (besucht am 2024-07-27) (siehe S. 106–111).

[7] Richard Beigel. "Irrationality without Number Theory". *The American Mathematical Monthly* 98(4):332–335, Apr. 1991. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.1080/00029890.1991.12000762. URL: https://cis.temple.edu/~beigel/papers/irrationality-amm.PS.gz (besucht am 2026-11-04) (siehe S. 426).

[8] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (besucht am 2025-02-05) (siehe S. 428).

[9] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 426).

[10] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: 978-0-321-35668-0 (siehe S. 426).

References II

[11] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 427).

[12] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 427).

[13] Georg Brandl und Serhiy Storchaka. *Underscores in Numeric Literals*. Python Enhancement Proposal (PEP) 515. Beaverton, OR, USA: Python Software Foundation (PSF), 10. Feb. 2016. URL: https://peps.python.org/pep-0515 (besucht am 2024-09-23) (siehe S. 188–228).

[14] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 425).

[15] "Built-in Functions". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/functions.html (besucht am 2024-12-09) (siehe S. 117, 118).

[16] "Built-in Functions: `round(number, ndigits=None)`". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/builtins/functions.html#round (besucht am 2026-10-25) (siehe S. 425).

[17] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 426, 427).

[18] Stephan C. Carlson und The Editors of Encyclopaedia Britannica. *Golden Ratio*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 21. Okt. 2024. URL: https://www.britannica.com/science/golden-ratio (besucht am 2024-12-14) (siehe S. 428).

[19] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, Juli 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (siehe S. 425).

[20] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (besucht am 2025-01-09) (siehe S. 428).

References III

[21] "Class Double". In: *Java® Platform, Standard Edition & Java Development Kit Version 22 API Specification*. Redwood Shores, CA, USA: Oracle Corporation, 9. Apr. 2024. URL: https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Double.html (besucht am 2024-07-07) (siehe S. 230–237, 251, 252, 268–271).

[22] 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. 428).

[23] Edgar Frank „Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 2025-01-05) (siehe S. 427).

[24] Scott A. Craver. *Results of the 2015 Underhanded C Contest: An Overview of NaN Poisoning Attacks*. 3. Feb. 2016. URL: http://underhanded-c.org/#nan (besucht am 2025-07-24) (siehe S. 365–375).

[25] Christopher Cullen. "Learning from Liu Hui? A Different Way to Do Mathematics". *Notices of the American Mathematical Society* 49(7):783–790, Aug. 2002. Providence, RI, USA: American Mathematical Society (AMS). ISSN: 1088-9477. URL: https://www.ams.org/notices/200207/comm-cullen.pdf (besucht am 2024-08-09) (siehe S. 429).

[26] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 428).

[27] Joseph W. Dauben. "Archimedes and Liu Hui on Circles and Spheres". *Ontology Studies (Cuadernos de Ontología)* 10:21–38, 2010. Leioa, Bizkaia, Spain: Universidad del País Vasco / Euskal Herriko Unibertsitatea. ISSN: 1576-2270. URL: https://ddd.uab.cat/pub/ontstu/ontstu_a2010n10/ontstu_a2010n10p21.pdf (besucht am 2026-11-03). Also discusses LIU Hui's (刘徽) method of approximating π (siehe S. 429).

[28] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: https://dataschool.com/how-to-teach-people-sql (besucht am 2025-02-27) (siehe S. 428).

[29] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 428).

References IV

[30] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 428).

[31] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 425).

[32] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 427).

[33] Euclid of Alexandria (Εὐκλείδης). *Euclid's Elements of Geometry (Στοιχεῖα). The Greek Text of J.L. Heiberg (1883-1885) from Euclidis Elementa, Edidit et Latine Interpretatus est I.L. Heiberg in Aedibus B. G. Teubneri, 1883-1885. Edited, and provided with a modern English translation, by Richard Fitzpatrick*. Hrsg. von Richard Fitzpatrick. Übers. von Johan Ludvig Heiberg. revised and corrected. Austin, TX, USA: The University of Texas at Austin, 2008. ISBN: 978-0-615-17984-1. URL: https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf (besucht am 2024-09-30) (siehe S. 428).

[34] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (besucht am 2024-09-24). Translation of [35]. (Siehe S. 416).

[35] 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 [34] for a translation. (Siehe S. 416, 429).

[36] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 427).

[37] Michael Filaseta. "The Transcendence of e and π". In: *Math 785: Transcendental Number Theory*. Columbia, SC, USA: University of South Carolina, Frühling 2011. Kap. 6. URL: https://people.math.sc.edu/filaseta/gradcourses/Math785/Math785Notes6.pdf (besucht am 2024-07-05) (siehe S. 5–14, 429).

References V

[38] "Floating-Point Arithmetic: Issues and Limitations". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 15. URL: https://docs.python.org/3/tutorial/floatingpoint.html (besucht am 2024-12-08) (siehe S. 38–47, 49–82).

[39] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys (CSUR)* 23(1):5–48, März 1991. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/103162.103163. URL: https://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/goldberg-floating-point.pdf (besucht am 2025-09-03) (siehe S. 16–22, 38–47, 333–336, 339–342).

[40] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: https://www.ibm.com/topics/api (besucht am 2024-12-12) (siehe S. 425).

[41] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 427).

[42] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 427).

[43] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 426).

[44] 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. 426, 428).

[45] Steve Hollasch. "IEEE Standard 754 Floating Point Numbers". In: *CSE401: Introduction to Compiler Construction*. Seattle, WA, USA: University of Washington, 8. Jan. 1997. URL: https://courses.cs.washington.edu/courses/cse401/01au/details/fp.html (besucht am 2024-07-05) (siehe S. 16–37, 176–183, 427).

[46] ."Why does Python 3 round half to even?" In: *Stack Overflow*. Hrsg. von wjandrea. New York, NY, USA: Stack Exchange Inc., 31. Mai 2012–13. Juni 2025. URL: https://stackoverflow.com/questions/10825926 (besucht am 2025-07-18). Closely related to banker's rounding (siehe S. 119, 425).

References VI

[47] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 427).

[48] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754™-2019 (Revision of IEEE Std 754-2008). New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), 13. Juni 2019 (siehe S. 16–47, 120, 121, 176–183, 230–237, 425, 427).

[49] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 428).

[50] *Java® Platform, Standard Edition & Java Development Kit Version 22 API Specification*. Redwood Shores, CA, USA: Oracle Corporation, 9. Apr. 2024. URL: https://docs.oracle.com/en/java/javase/22/docs/api (besucht am 2024-07-07) (siehe S. 113–174, 425).

[51] Arthur Jones, Kenneth R. Pearson und Sidney A. Morris. "Transcendence of e and π". In: *Abstract Algebra and Famous Impossibilities*. Universitext (UTX). New York, NY, USA: Springer New York, 1991. Kap. 9, S. 115–161. ISSN: 0172-5939. ISBN: 978-1-4419-8552-1. doi:10.1007/978-1-4419-8552-1_8 (siehe S. 5–14, 429).

[52] Shen Kangshen, John Newsome Crossley und Anthony W.-C. Lun. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. Oxford, Oxfordshire, England, UK: Oxford University Press, 7. Okt. 1999. ISBN: 978-0-19-853936-0. doi:10.1093/oso/9780198539360.001.0001. Translation of the historical Chinese mathematics book *Jiu Zhang Suanshu* (九章算术), including the commentaries of LIU Hui (刘徽). (Siehe S. 429).

[53] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 427).

[54] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 427).

References VII

[55] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: https://www.britannica.com/technology/client-server-architecture (besucht am 2025-01-20) (siehe S. 426).

[56] Marc Loy, Patrick Niemeyer und Daniel Leuck. *Learning Java*. 5. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2020. ISBN: 978-1-4920-5627-0 (siehe S. 426).

[57] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 427).

[58] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 427).

[59] "Mathematical Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.3. URL: https://www.postgresql.org/docs/17/functions-math.html (besucht am 2025-02-27) (siehe S. 429).

[60] Clive Maxfield und Alvin Brown. "Rounding Algorithms 101". In: *DIY Calculator*. Huntsville, AL, USA: DIY Calculator, 2005. URL: https://www.clivemaxfield.com/diycalculator/popup-m-round.shtml (besucht am 2025-07-18). Also discusses banker's rounding (siehe S. 119, 425).

[61] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISSN: 1046-1698. ISBN: 978-1-55860-456-8 (siehe S. 428).

[62] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 425).

[63] Ivan Niven. "The Transcendence of π". *The American Mathematical Monthly* 46(8):469–471, Okt. 1939. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.2307/2302515 (siehe S. 5–14, 429).

[64] "Numeric Types – `int`, `float`, `complex`". 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#typesnumeric (besucht am 2024-07-05) (siehe S. 16–22).

References VIII

[65] John J. O'Connor und Edmund F. Robertson. *Liu Hui*. St Andrews, Scotland, UK: University of St Andrews, School of Mathematics and Statistics, Dez. 2003. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Liu_Hui (besucht am 2024-08-10). Discusses LIU Hui's (刘徽) method of approximating π (siehe S. 429).

[66] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 427).

[67] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 426).

[68] Yun Peng, Yubing Wang, Yingxu Zhang, Lei Liang, Jun Zhang, Jianwei Zhang, Qin Li und Lijun Wang. "Hardware Implementation of Improved Banker's Fixed-Point Rounding Algorithm". *IEEE Access* 13:36679–36686, 19. Feb. 2025. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 2169-3536. doi:10.1109/ACCESS.2025.3543540 (siehe S. 425).

[69] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: https://www.postgresql.org/docs/17/index.html (besucht am 2025-02-25).

[70] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 427).

[71] William H. Press, Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery. "1.1 Error, Accuracy, and Stability". In: *Numerical Recipes: The Art of Scientific Computing*. 3. Aufl. Cambridge, England, UK: Cambridge University Press (CUP), 2007–2011. Kap. 1 Preliminaries, S. 8–12. ISBN: 978-0-521-88068-8. URL: https://numerical.recipes/book.html (besucht am 2024-07-27). Version 3.04 (siehe S. 106–111).

[72] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (besucht am 2024-06-29) (siehe S. 425).

[73] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 426).

References IX

[74] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 427).

[75] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 427).

[76] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 426).

[77] Syamal K. Sen und Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Kap. 5.5, S. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (siehe S. 425).

[78] 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. 426).

[79] Neil James Alexander Sloane. *Decimal Expansion of Golden Ratio phi (or tau) = (1 + sqrt(5))/2*. Hrsg. von John Horton Conway. Bd. A001622 der Reihe The On-Line Encyclopedia of Integer Sequences. Highland Park, NJ, USA: The OEIS Foundation Inc., 13. Dez. 2024. URL: https://oeis.org/A001622 (besucht am 2024-12-14) (siehe S. 428).

[80] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 427).

[81] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (besucht am 2025-02-25) (siehe S. 428).

[82] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Teach Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 422, 428).

References X

[83] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of [82] (siehe S. 428).

[84] Philip D. Straffin Jr. "Liu Hui and the First Golden Age of Chinese Mathematics". *Mathematics Magazine* 71(3):163–181, Juni 1998. London, England, UK: Taylor and Francis Ltd. ISSN: 0025-570X. doi:10.2307/2691200. URL: https://www.researchgate.net/publication/237334342 (besucht am 2024-08-10) (siehe S. 429).

[85] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 427, 428).

[86] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 427).

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

[88] *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library (besucht am 2025-04-27).

[89] 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. 426).

[90] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 426).

[91] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2025. URL: https://thomasweise.github.io/databases (besucht am 2025-01-05) (siehe S. 426, 427).

[92] Thomas Weise (汤卫思). *Programming with C*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2026. URL: https://thomasweise.github.io/programmingWithC (besucht am 2026-05-10) (siehe S. 425).

References XI

[93] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 427).

[94] Eric Wolfgang Weisstein. "Irrational Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/IrrationalNumber.html (besucht am 2026-11-04) (siehe S. 426).

[95] Eric Wolfgang Weisstein. *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com (besucht am 2024-09-24).

[96] Eric Wolfgang Weisstein. "Transcendental Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/TranscendentalNumber.html (besucht am 2026-11-04) (siehe S. 428).

[97] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: https://www.ibm.com/think/topics/relational-databases (besucht am 2025-01-05) (siehe S. 427).

[98] Ulf Michael „Monty" Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 427).

[99] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (besucht am 2025-01-11) (siehe S. 426).

[100] Jeffrey Yasskin. *A Type Hierarchy for Numbers*. Python Enhancement Proposal (PEP) 3141. Beaverton, OR, USA: Python Software Foundation (PSF), 25. Apr.–2. Aug. 2007. URL: https://peps.python.org/pep-3141 (besucht am 2025-07-28) (siehe S. 159, 160).

[101] Wenqi Ying (应雯棋), Hrsg. *Commemoration of Ancient Chinese Mathematical Master Liu Hui for his Timeless Influence on Mathematics and Civilizational Exchange*. Bd. 48 (Special Issue) der Reihe CAST Newsletter. China, Beijing (中国北京市): 中国科学技术协会 (China Association for Science and Technology, CAST), Nov. 2024. URL: https://english.cast.org.cn/cms_files/filemanager/1941250207/attach/202412/8f23655a82364d19ad7874eb37b23035.pdf (besucht am 2025-08-24). Proofreader: Yumeng Wei (魏雨萌), Designer: Shan Zhang (张珊) (siehe S. 429).

References XII

[102] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 425).

[103] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenberg und Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 26. Juli 1999–20. März 2024. URL: https://www.britannica.com/topic/chronology/Christian (besucht am 2025-08-27) (siehe S. 425).

Glossary (in English) I



API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another [40].

banker's rounding means rounding a real number to the closest integer and, in case of ties, to the closest *even* integer [3,68]. Since ties can only occur for numbers ending in `.5`, this means that all numbers of the form `z.5` will be rounded to `z+1` if `z` is odd and to `z` if `z` is even. This is different from the „normal" understanding of rounding as we often learn in school, where all numbers of the form `z.5` are rounded to `z+1`. Banker's rounding has the advantage that it is not biased in any direction [46,60]. If `z.5` is always rounded toward `z+1` and we have many numbers of the form `z.5`, then the sum or average of the results tends to be somewhat larger than the sum or average of the unrounded numbers. Banker's rounding avoids this problem by sometimes rounding numbers ending in `5` to larger and sometimes to smaller values. In the IEEE Floating-Point Arithmetic Standard 754 [48], this method corresponds to the *roundTiesToEven* direction introduced in its section *4.3.1 Rounding-direction attributes to nearest*. Python uses banker's rounding in its `round` function [16], whereas Java's `Math.round` uses the „school-type" rounding [50].

Bash is a shell used under Ubuntu Linux, i.e., the program that „runs" in the terminal and interprets your commands, allowing you to start and interact with other programs [14,62,102]. Learn more at `https://www.gnu.org/software/bash`.

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ [19]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 Common Era (CE) [77,103].

C is a programming language, which is very successful in system programming situations [31,72,92].

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ [19]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 before Common Era (BCE) [77,103].

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples of clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as psql.

Glossary (in English) II



client-server architecture is a system design where a central server receives requests from one or multiple clients [9,55,67,73,76]. These requests and responses are usually sent over network connections. A typical example of such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases* [91].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB [99].

denominator The number b of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *denominator*.

irrational An *irrational* number z is a real number that cannot be expressed as fraction $\frac{p}{q}$ of two integers $p \in \mathbb{Z}$ and $q \in \mathbb{N}_1$ [7,94]. In other words, it holds that $z \in \mathbb{R}$ and $z \notin \mathbb{Q}$. The set of irrational numbers could be denoted as $\mathbb{R} \setminus \mathbb{Q}$. Irrational numbers have decimal expansions that neither terminate nor become periodic. Examples of irrational numbers are $\sqrt{2}$, ϕ, π, and e.

IT information technology

Java is another very successful programming language, with roots in the C family of languages [10,56].

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP [17,44].

Linux is the leading open source operating system, i.e., a free alternative to Microsoft Windows [4,43,78,89,90]. 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.

Glossary (in English) III



MariaDB An open source relational database management system that has forked off from MySQL [1,2,5,32,58,74]. See https://mariadb.org for more information.

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

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

MySQL An open source relational database management system [11,32,75,86,98]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

numerator The number a of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *numerator*.

PostgreSQL An open source object-relational DBMS [36,66,70,86]. See https://postgresql.org for more information.

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

Python The Python programming language [47,54,57,93], i.e., what you will learn about in our book [93]. Learn more at https://python.org.

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other [23,41,42,80,85,91,97].

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples of servers are web servers [17] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“ [53].

significand The significand is the part of a floating point number that stores the digits of the number (in binary representation). In the 64 bit double precision IEEE Standard 754 floating point number layout [45,48], the exponent is 52 bits.

Glossary (in English) IV



SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases [20,26,28,29,49,61,81–83,85]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference [81].

terminal A terminal is a text-based window where you can enter commands and execute them [4,22]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

transcendental A *transcendental* number is a number that is not the root of any integer polynomial, meaning that it is not an algebraic number of any degree [96]. Every real transcendental number must also be irrational, since a rational number is, by definition, an algebraic number of degree one [96]. Examples of transcendental numbers are π and e.

Ubuntu is a variant of the open source operating system Linux [22,44]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

WWW World Wide Web [8,30]

ϕ The golden ratio (or golden section) ϕ is the irrational number $\frac{1+\sqrt{5}}{2}$. It is the ratio of a line segment cut into two pieces of different lengths such that the ratio of the whole segment to that of the longer segment is equal to the ratio of the longer segment to the shorter segment [18,33]. The golden ratio is approximately $\phi \approx 1.618\,033\,988\,749\,894\,848\,204\,586\,834$ [79]. Represented as `float` in Python, its value is `1.618033988749895`.

Glossary (in English) V

π is the ratio of the circumference U of a circle and its diameter d, i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number [37,51,63], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. An early method to approximate π was developed by the Chinese mathematician LIU Hui (刘徽) based on inscribing regular e-gons into a circle somewhere in the third century CE [65,101] in his commentary to the famous Chinese mathematics book *Jiu Zhang Suanshu* (九章算术) [25,27,52,65,84]. In Python, π is provided by the `math` module as variable `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793` [59].

e is Euler's number [35], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number [37,51], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as variable `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045` [59].

$\mathbb{N_0}$ the set of the natural numbers *including* 0, i.e., 0, 1, 2, 3, and so on. It holds that $\mathbb{N_0} \subset \mathbb{Z}$ and that $\mathbb{N_0} = \mathbb{N_1} \cup \{0\}$.

$\mathbb{N_1}$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N_1} \subset \mathbb{Z}$ and that $\mathbb{N_1} = \mathbb{N_0} \setminus \{0\}$.

$\mathbb{Q}$ the set of the rational numbers, i.e., the set of all numbers that can be the result of $\frac{a}{b}$ with $a, b \in \mathbb{Z}$ and $b \neq 0$. a is called the numerator and b is called the denominator. It holds that $\mathbb{Z} \subset \mathbb{Q}$ and $\mathbb{Q} \subset \mathbb{R}$.

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

$\mathbb{Z}$ the set of the integers numbers including positive and negative numbers and 0, i.e., . . . , -3, -2, -1, 0, 1, 2, 3, . . . , and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.