合肥大學
HEFEI UNIVERSITY

Programming with Python

8. Der Datentyp float

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

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

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-07-25

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

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 transcendental[11,19,24], 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 transcendental[11,19,24], 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 transcendental[11,19,24], 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 keine beliebigen Nachkommastellen genau darstellen 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 keine beliebigen Nachkommastellen genau darstellen 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 keine beliebigen Nachkommastellen genau darstellen 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 Datentype `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 keine beliebigen Nachkommastellen genau darstellen 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 Datentype `float`.
- `float` hat die selbe interne Struktur wie der Datentyp `doubles` in C[25] – er folgt dem *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17].

Wie funktioniert das?

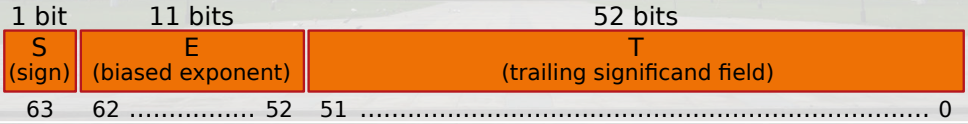
- Aber wie kann man überhaupt eine brauchbare Untermenge von $\mathbb{R}$ auf dem Computer darstellen?
- Wie können wir mit dem Fakt umgehen, dass wir keine beliebigen Nachkommastellen genau darstellen können, selbst in so häufig vorkommenden Fällen wie π und e?
- Wie gehen wir damit um, dass es reelle Zahlen gibt so groß wie 10^{300} und so klein wie 10^{-300}?
- In Python gibt es dafür den Datentype `float`.
- `float` hat die selbe interne Struktur wie der Datentyp `doubles` in C[25] – er folgt dem *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17].
- 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 keine beliebigen Nachkommastellen genau darstellen 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 Datentype `float`.
- `float` hat die selbe interne Struktur wie der Datentyp `doubles` in C[25] – er folgt dem *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17].
- 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*[14,17] kurz an.

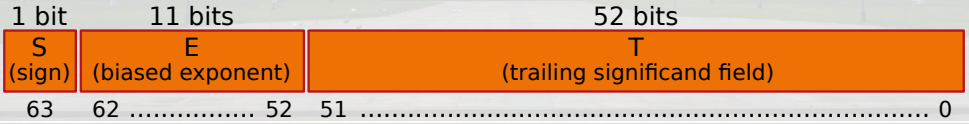


64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] kurz an.
- Die ersten 52 Bit – der sogenannte Significand oder die Mantisse – speichern die Ziffern der Zahl.

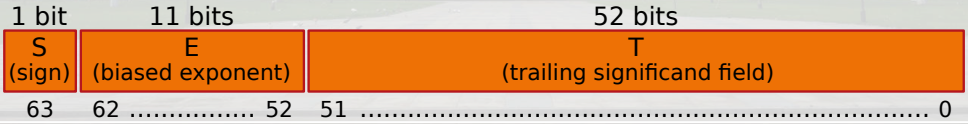
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] kurz an.
- Die ersten 52 Bit – der sogenannte Significand oder die Mantisse – speichern die Ziffern der Zahl.
- Mit 52 Bits können wir $52 \log_2 10 \approx 15$ bis 16 dezimale Ziffern darstellen.



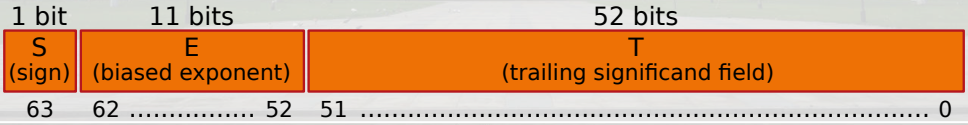
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] kurz an.
- Die ersten 52 Bit – der sogenannte Significand 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.



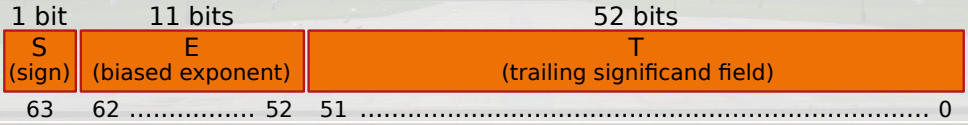
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] kurz an.
- Die ersten 52 Bit – der sogenannte Significand 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.



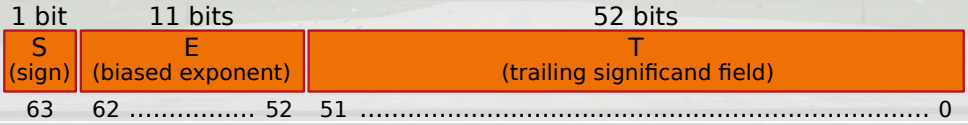
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] kurz an.
- Die ersten 52 Bit – der sogenannte Significand 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.



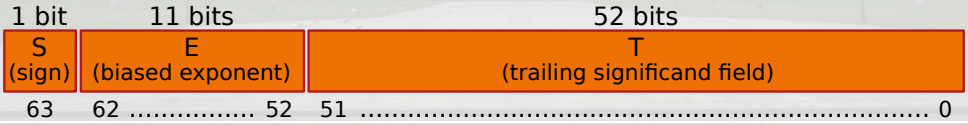
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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.



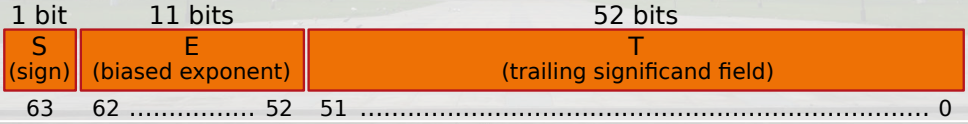
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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.



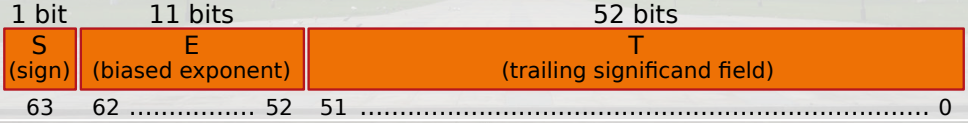
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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.



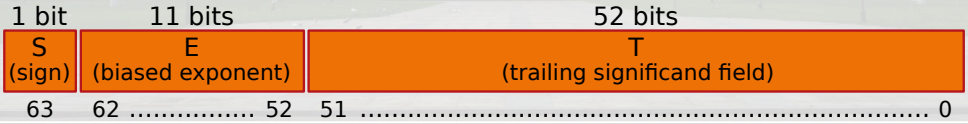
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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 Significand multipliziert wird.



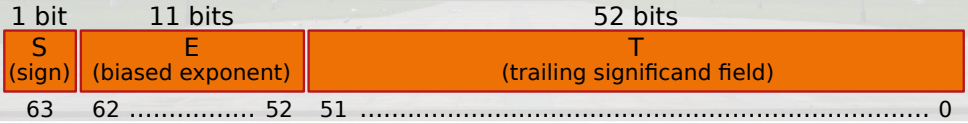
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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 Significand multipliziert wird.
- Damit wir sowohl große als auch kleine Zahlen darstellen können, muss der Exponent sowohl positiv als auch negativ seien können.



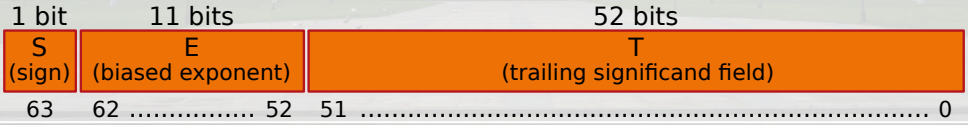
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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 Significand 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.



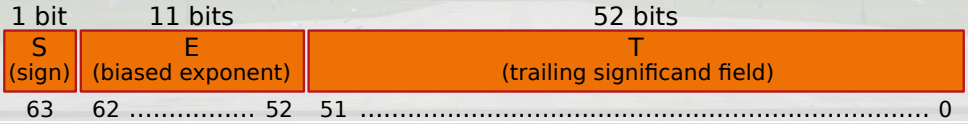
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] kurz an.
- `float` Zahlen haben eine Präzision von etwa 15 Ziffern.
- Der Exponent repräsentiert eine Potenz von 2, mit der der Significand 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 Significand wird mit $2^{27} = 134\,217\,728$ multipliziert.



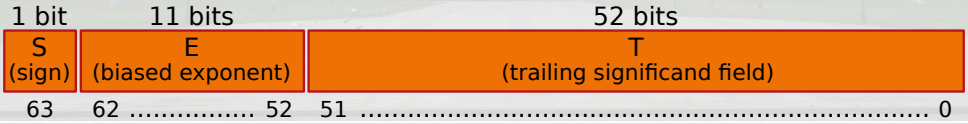
64 Bit IEEE Standard 754 Floating Point Number Layout

- Schauen wir uns das *64 Bit IEEE Standard 754 Floating Point Number Layout*[14,17] 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 Significand 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*[14,17] 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 Significand 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 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 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Erbenis 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 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Erbenis immer noch 10^{16}, weil nur 15 Ziffern "gespeichert" werden und die 1 einfach "herunterfällt."
- Zahlen können mit `floats` nicht beliebig genau gespeichert werden[12].

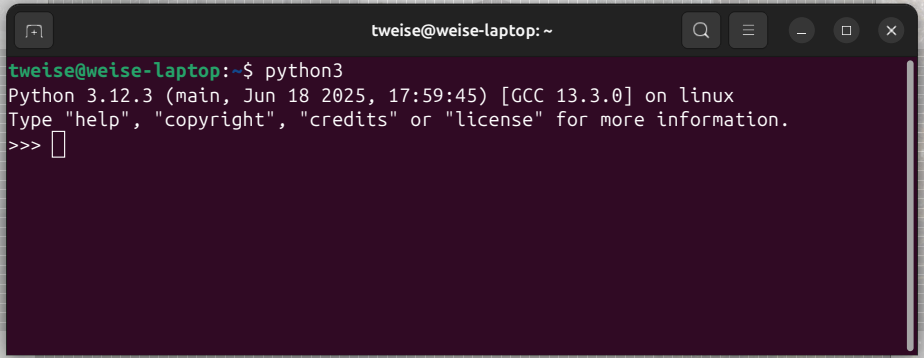
Das Wichtige

- 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 Ziffern begrenzt.
- Wenn Sie zum Beispiel 1 zu 10^{16} addieren, dann ist das Erbenis immer noch 10^{16}, weil nur 15 Ziffern "gespeichert" werden und die 1 einfach "herunterfällt."
- Zahlen können mit `floats` nicht beliebig genau gespeichert werden[12].
- Ansonsten sind `floats` aber schon ziemlich cool.

Rechnen mit float

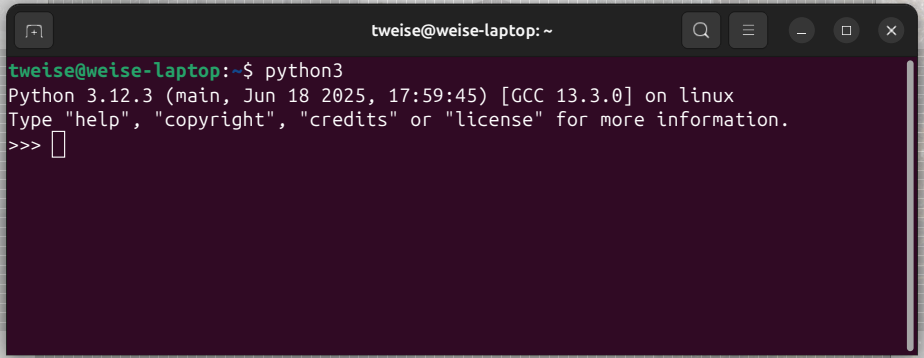
Rechnen mit float

- Lassen Sie uns nun mit `floats` rechnen.



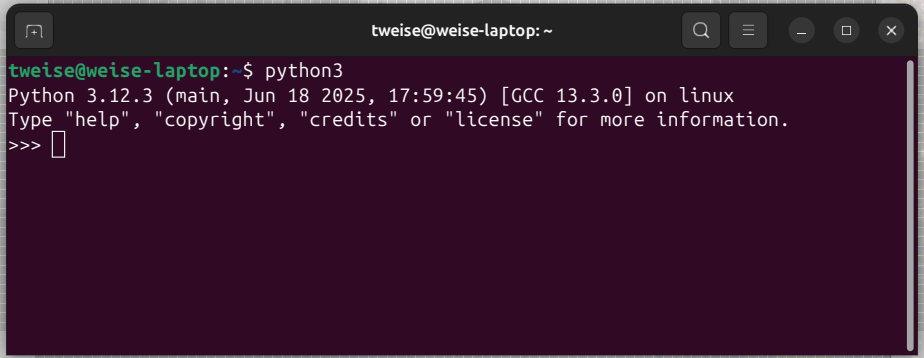
Rechnen mit float

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



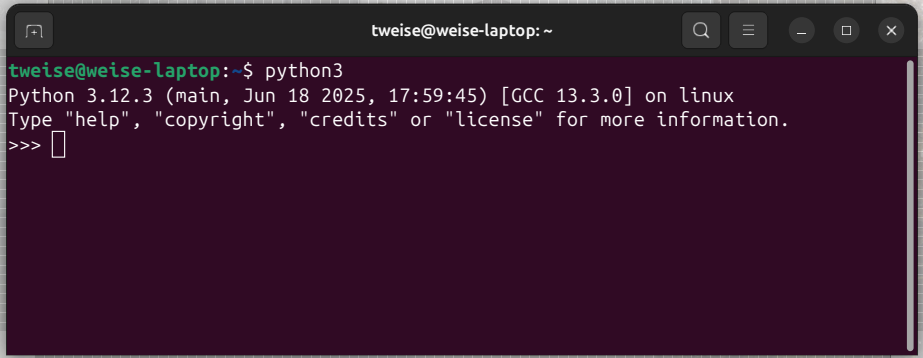
Rechnen mit float

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



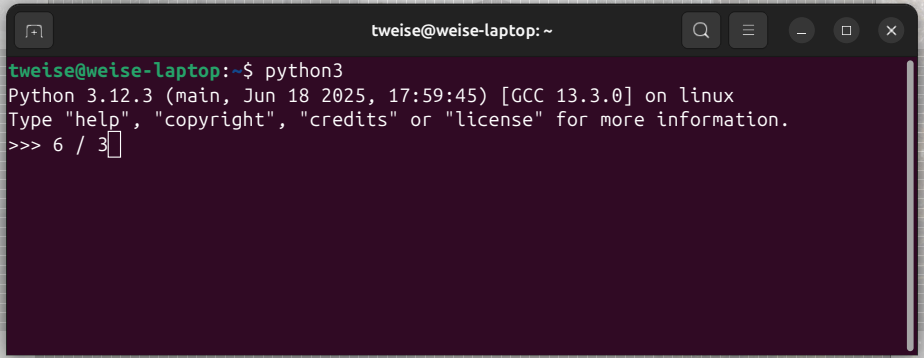
Rechnen mit float

- 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`.



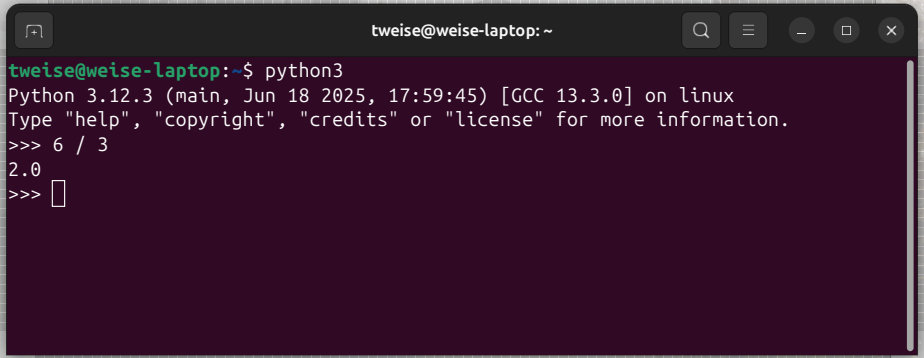
Rechnen mit float

- Die Fließkommadivision `6 / 3`...



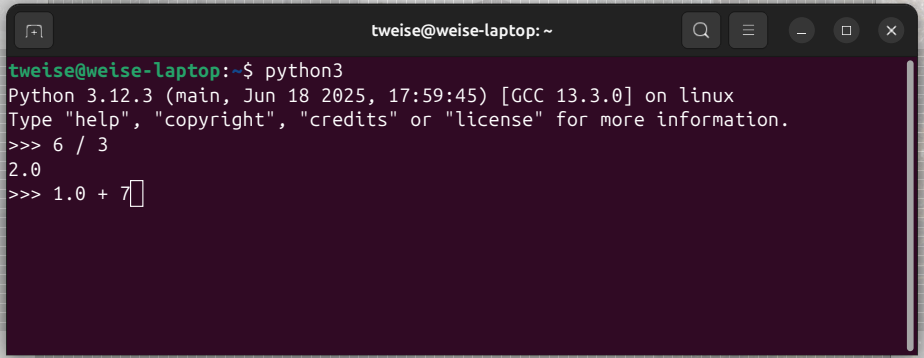
Rechnen mit float

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



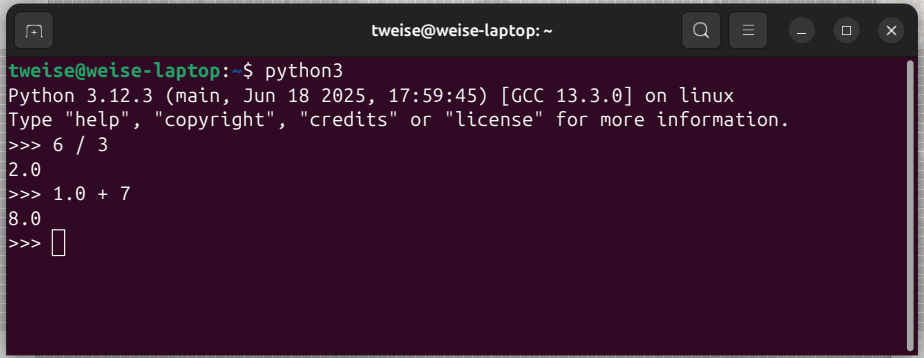
Rechnen mit float

- Addieren wir den `float` `1.0` zum `int` `1`...



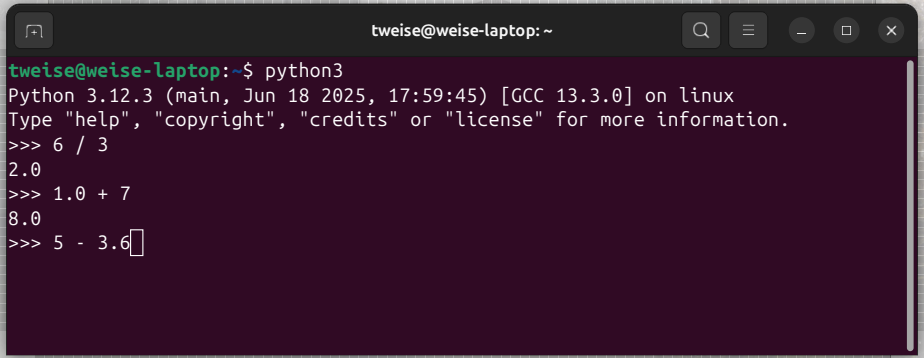
Rechnen mit float

- Addieren wir den `float` `1.0` zum `int` `1` ergibt das den `float` `8.0`.



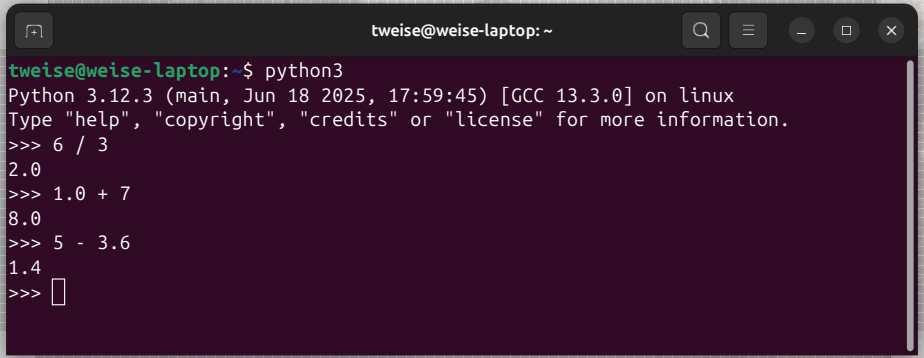
Rechnen mit float

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



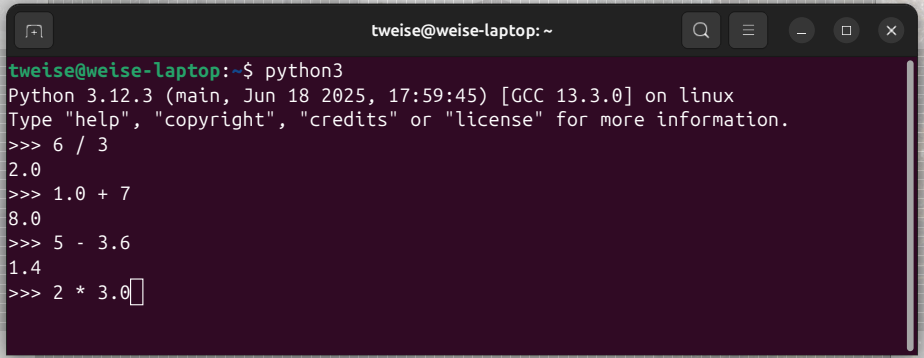
Rechnen mit float

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



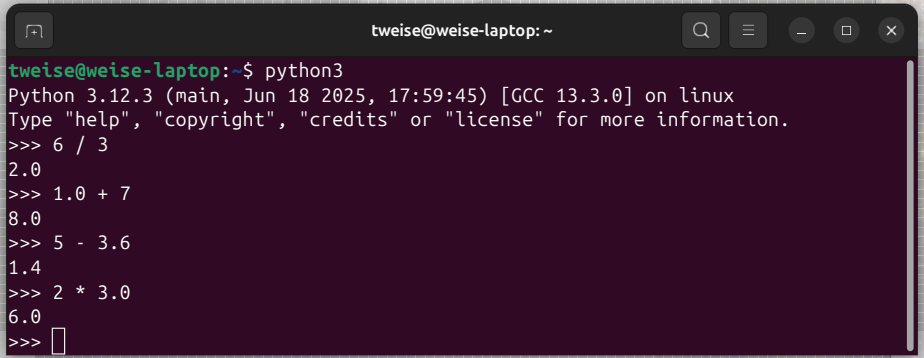
Rechnen mit float

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



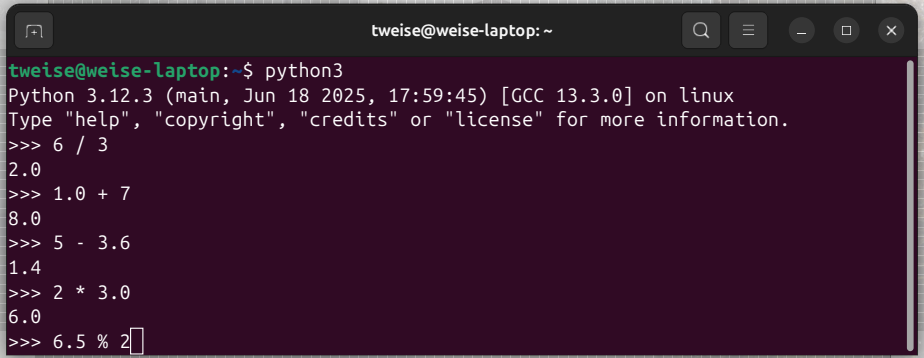
Rechnen mit float

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



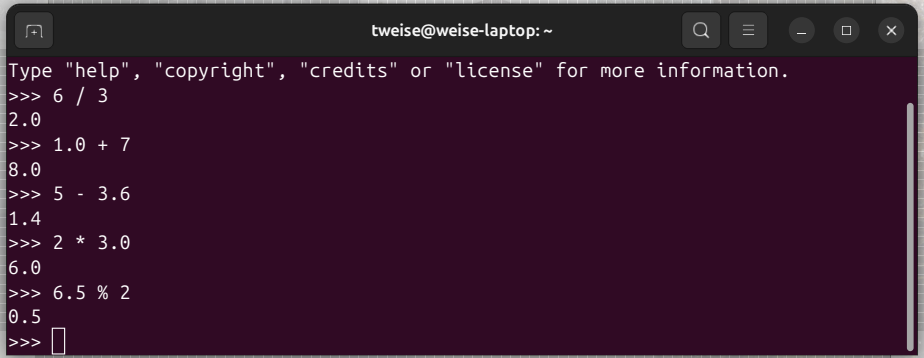
Rechnen mit float

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



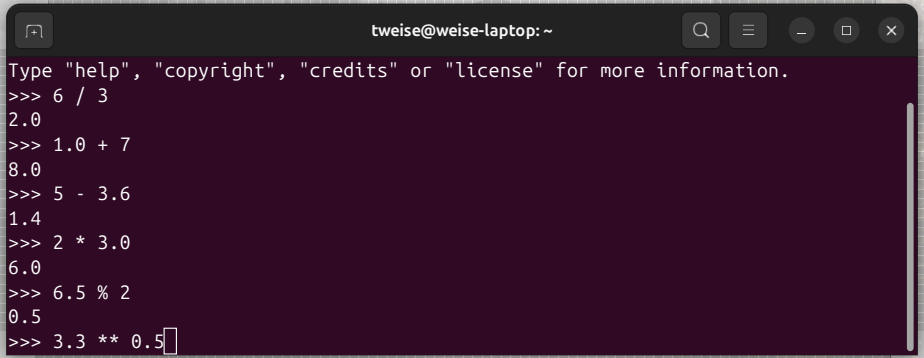
Rechnen mit float

- 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

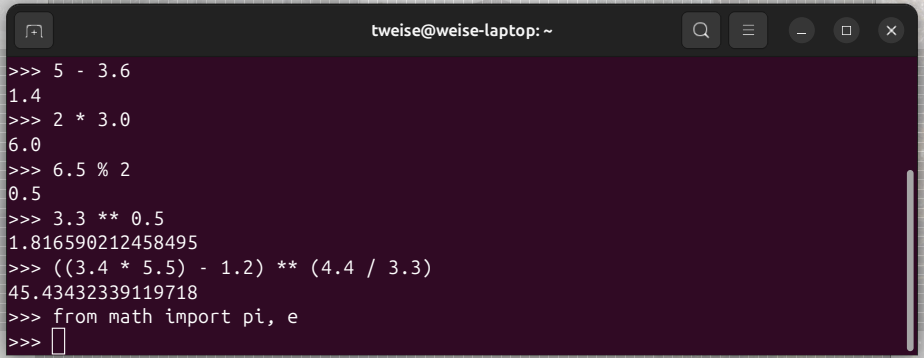
- Wir lernen später, was das genau bedeutet.
- Für jetzt, nehmen wir die Syntax `from math import XXX` an, wenn wir `XXX` nutzen wollen.

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`.

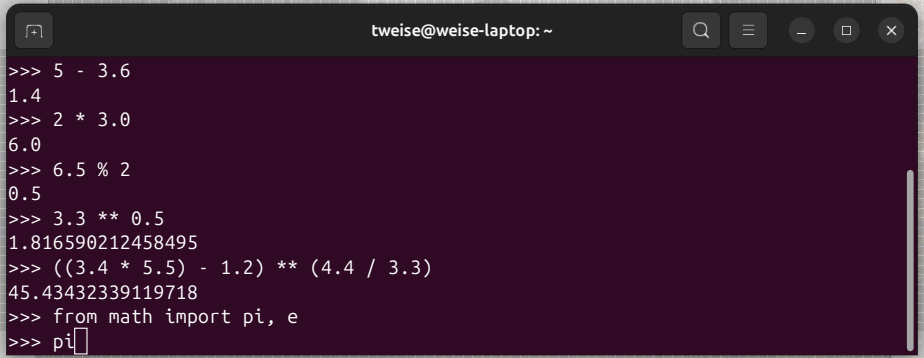
Rechnen mit float

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



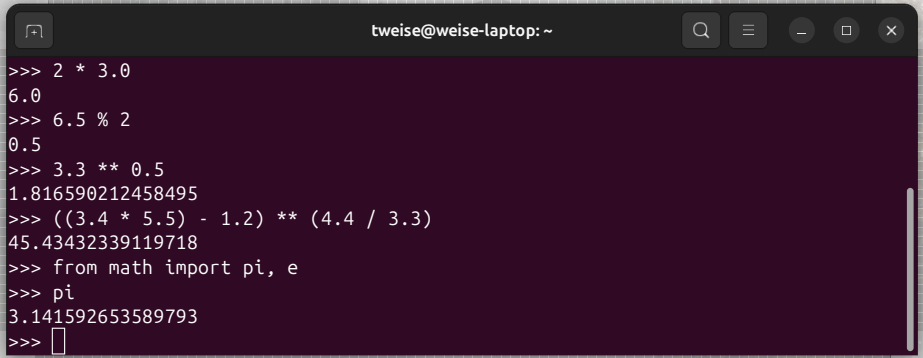
Rechnen mit float

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



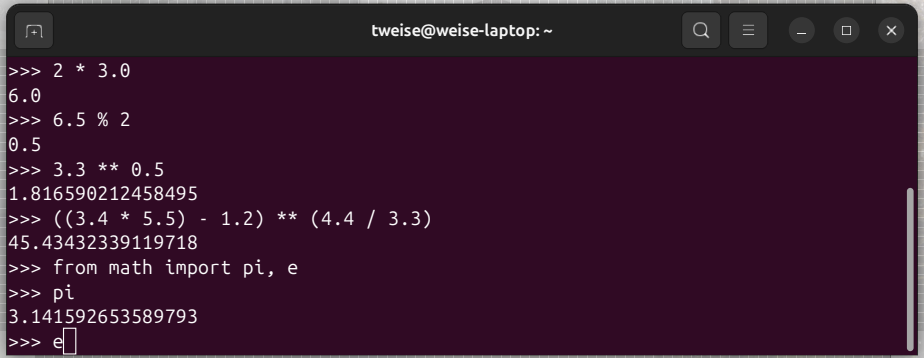
Rechnen mit float

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



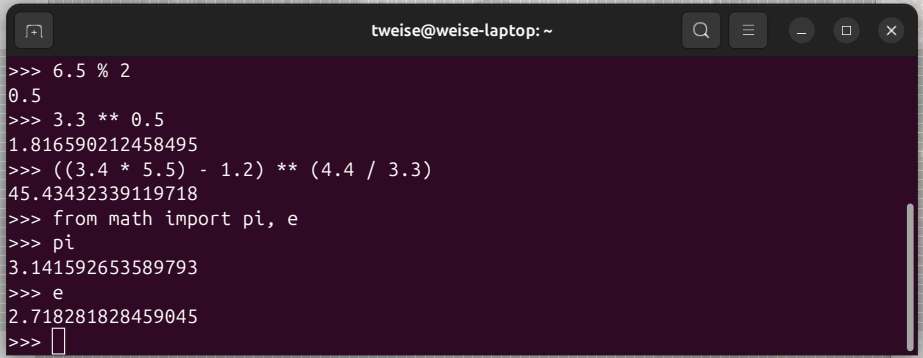
Rechnen mit float

- 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.
- 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 $\frac{1}{10} = 0.1^{12}$.

Rechnen mit float

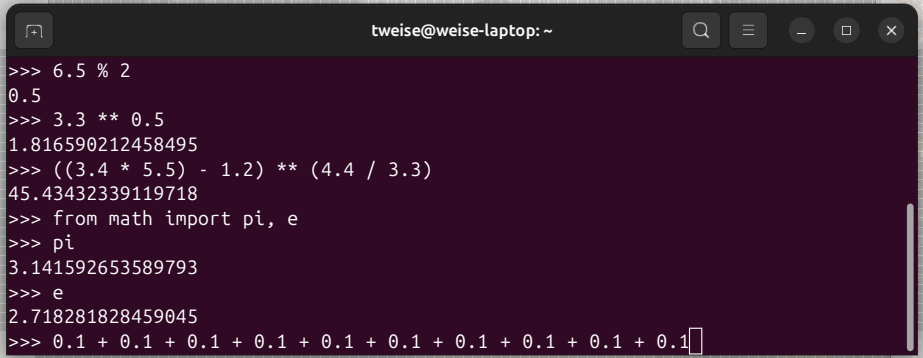
- Stellen Sie sich vor, wir sollen den Bruch $\frac{1}{7}$ als dezimale Kommazahl aufschreiben.
- Das funktioniert nicht. Wir müssen immer irgendwo abbrechen.
- Wir könnten z.B. 0.14285714285714285 schreiben ... aber das ist nicht das Gleiche wie $\frac{1}{7}$.
- Im binären Format sehen wir das Problem schon bei Zahlen wie $\frac{1}{10} = 0.1^{12}$.
- 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 $\frac{1}{10} = 0.1^{12}$.
- 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.

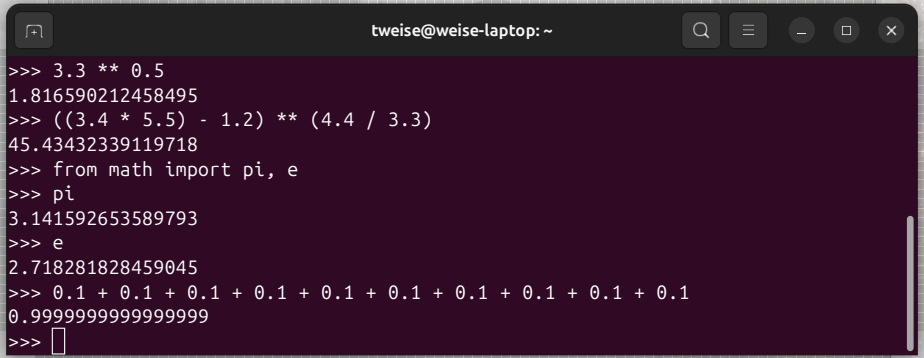
Rechnen mit float

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



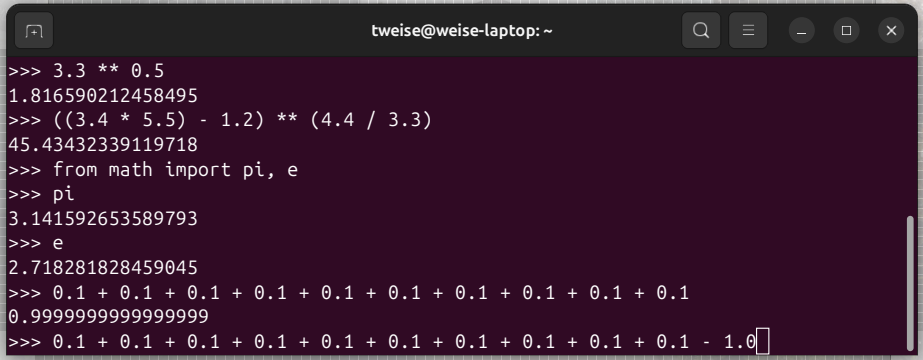
Rechnen mit float

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



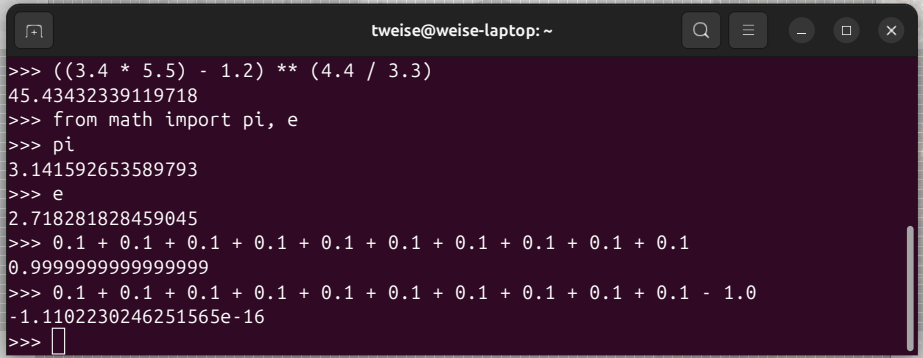
Rechnen mit float

- Ziehen wir `1.0` von "zehn `0.1`" ab...



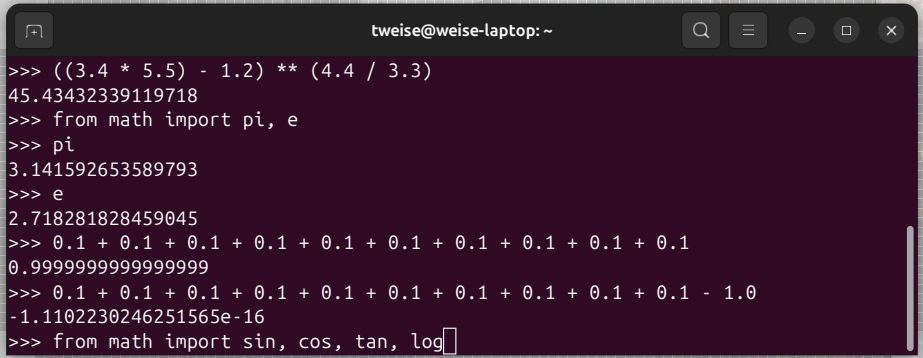
Rechnen mit float

- 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

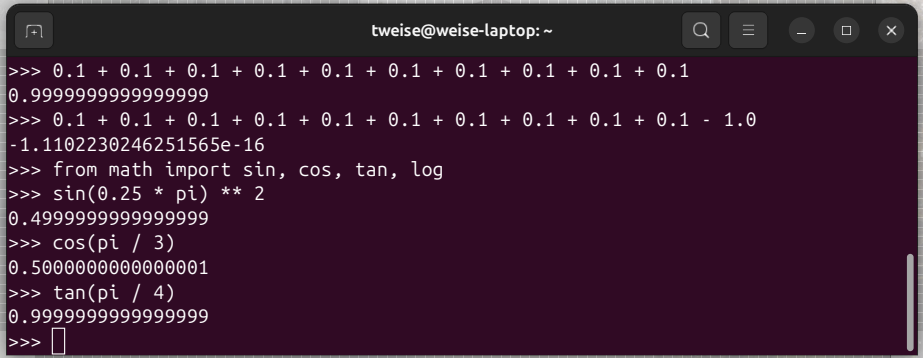
- Aus der Schule wissen wir, dass $\cos\left(\frac{\pi}{3}\right) = \frac{1}{2}$, also berechnen wir `cos(pi / 3)` und das Ergebnis kommt auf etwa 15 Stellen genau hin...

Rechnen mit float

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

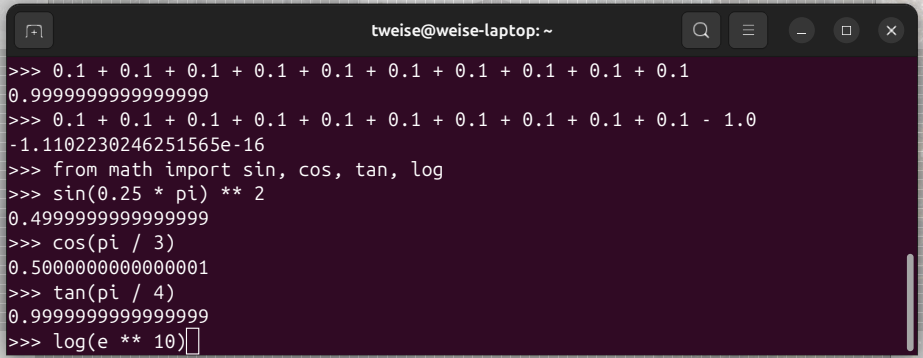
Rechnen mit float

- 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)`...

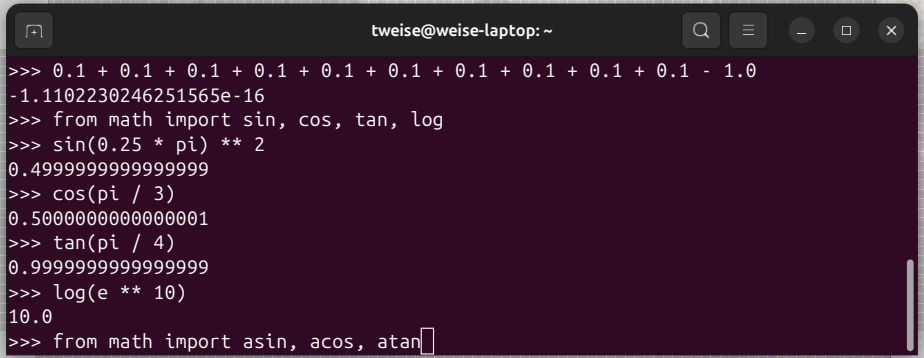


Rechnen mit float

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

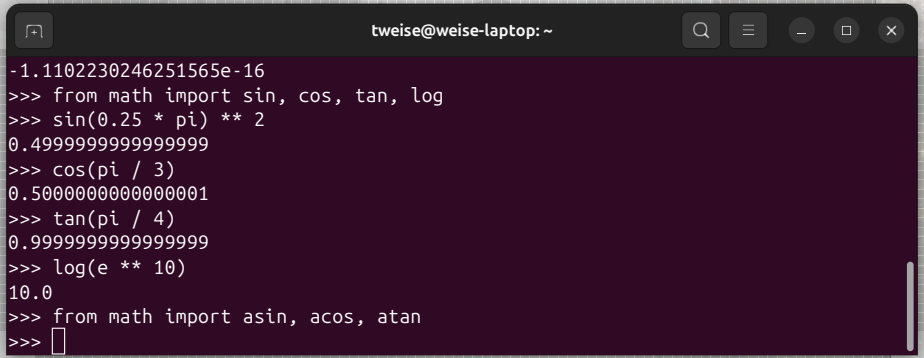
Rechnen mit float

- Python hat natürlich auch die Umkehrfunktionen der trigonometrischen Funktionen.



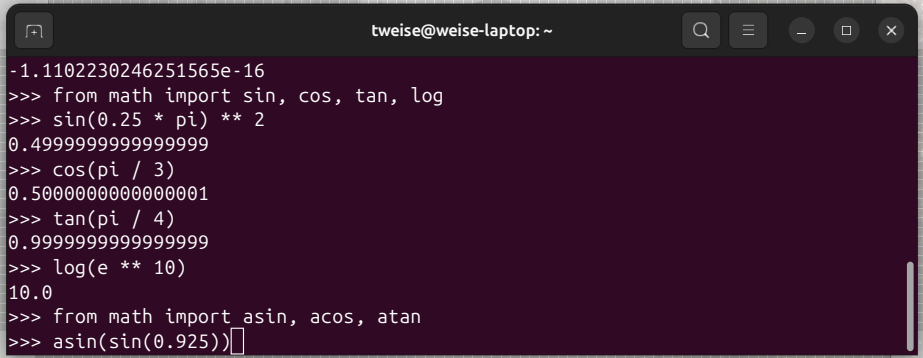
Rechnen mit float

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



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

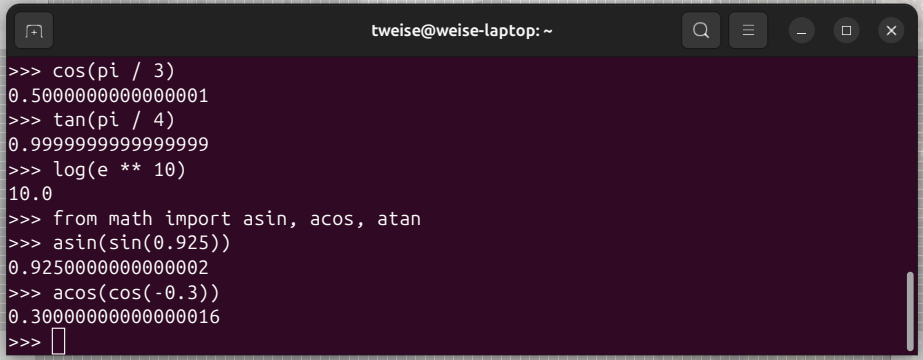
- Aus der Schule wissen wir, dass $\arcsin \sin x = x \; \forall x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, also berechnen wir `asin(sin(0.925))` und das Ergebnis stimmt genau.

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...

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[2,26].

Fließkommazahlen sind ungenau

Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[2,26].

- 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[2,26].

- 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[2,26].

- 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[26].

Fließkommazahlen sind ungenau

Gute Praxis

Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[2,26].

- 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[26].
- 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

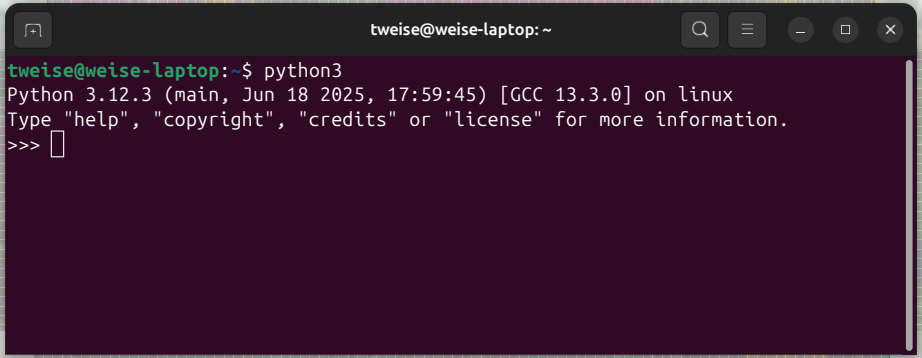
Nehmen Sie immer an, dass `float`-Werte ungenau sind. Nehmen Sie niemals an, dass sie genau sind[2,26].

- 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[26].
- 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[2]...

Runden

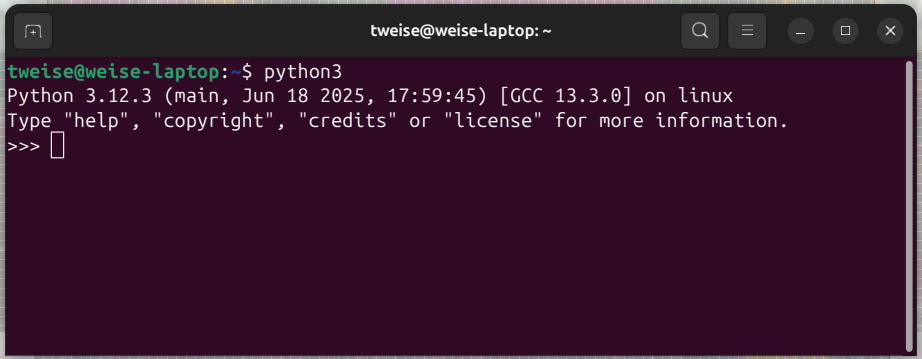
Runden

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



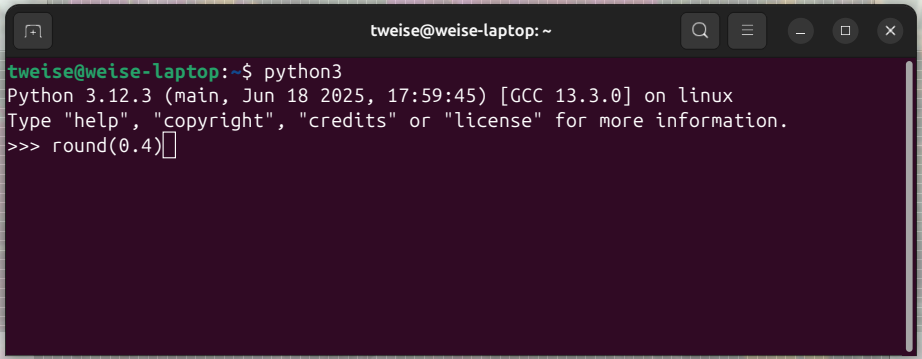
Runden

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



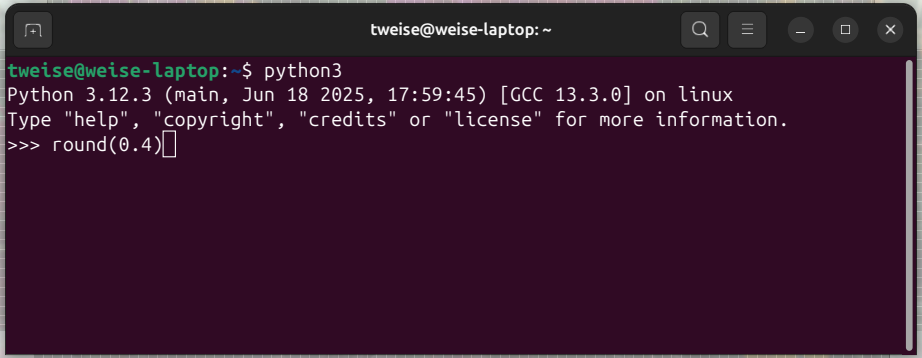
Runden

- Die Funktion `round(x)` führt sogenanntes “Banker's Rounding” durch.



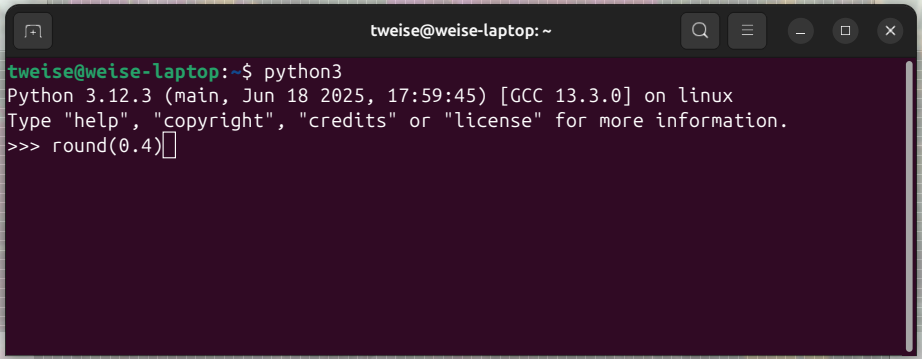
Runden

- Die Funktion `round(x)` führt sogenanntes “Banker's Rounding” durch.
- Dabei wird `x` zur nächsten Ganzzahl gerundet.



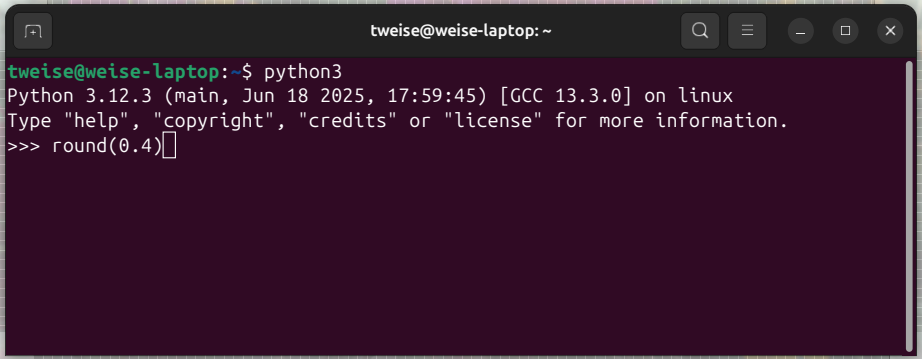
Runden

- 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.[5]



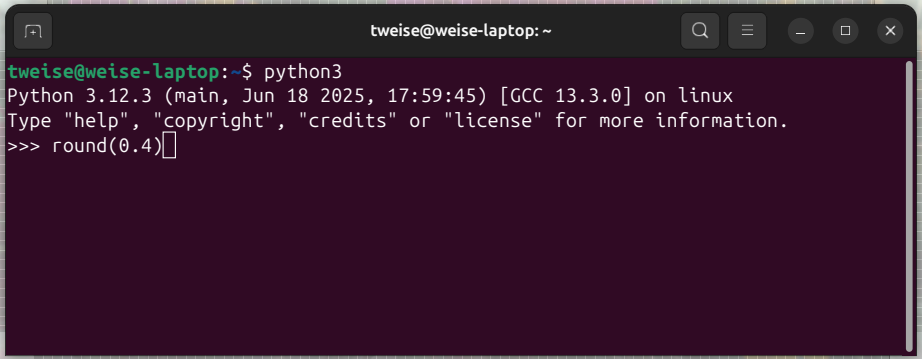
Runden

- 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.[5]
- In der Schule hatte ich gelernt, dass `z.5` zu `z + 1` gerundet wird. Java’s `Math.round` macht das auch so[18] ... aber Python’s `round` macht das nicht!



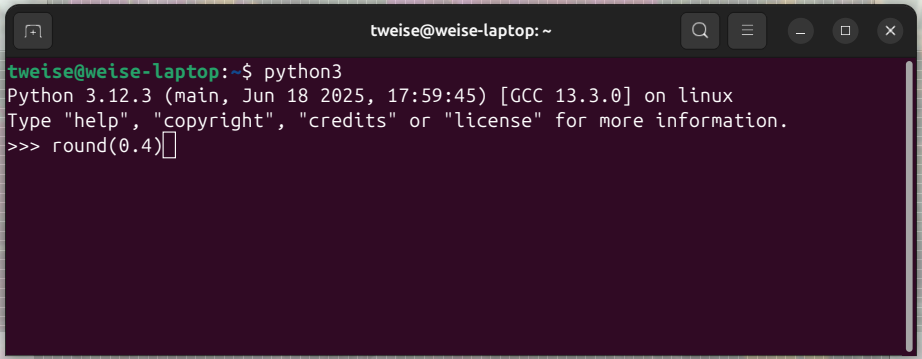
Runden

- 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[15,23].



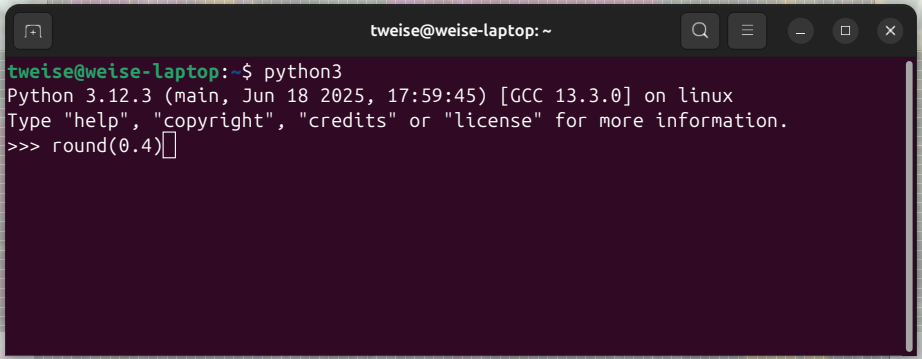
Runden

- Die Funktion `round(x)` führt sogenanntes “Banker's Rounding” durch.
- Deshalb ist es in Python und im IEEE 754 Standard[17] bevorzugt und wird z.B. auch von Alipay+ genutzt[1].



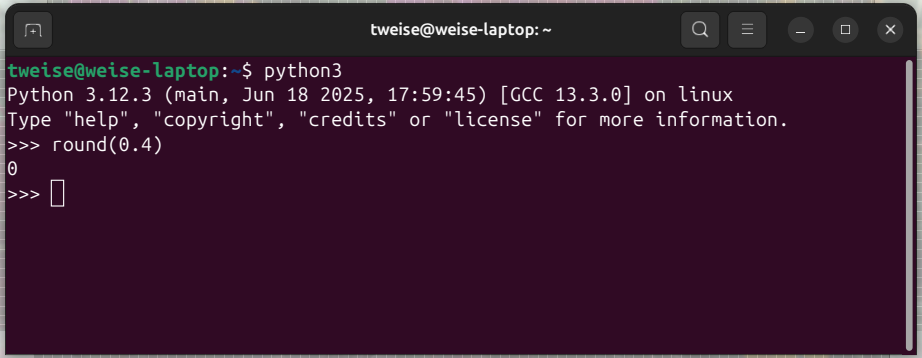
Runden

- Deshalb ist es in Python und im IEEE 754 Standard[17] bevorzugt und wird z.B. auch von Alipay+ genutzt[1].
- Wenn wir `round(0.4)` aufrufen...



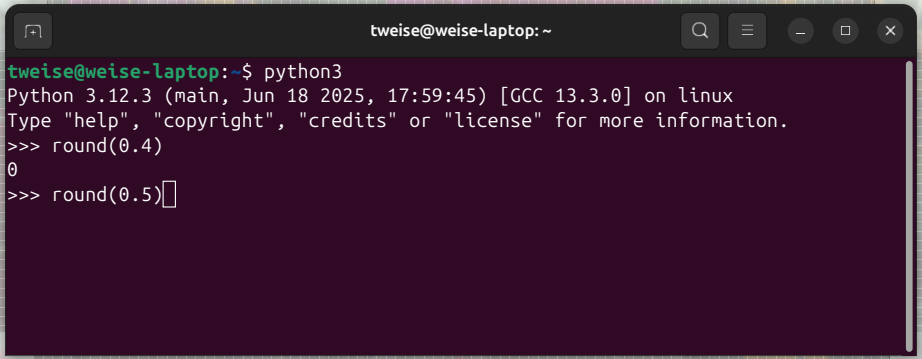
Runden

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



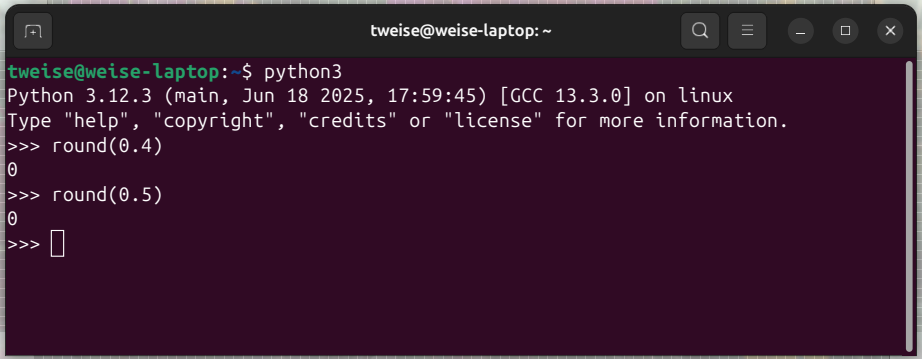
Runden

- Wenn wir `round(0.5)` aufrufen



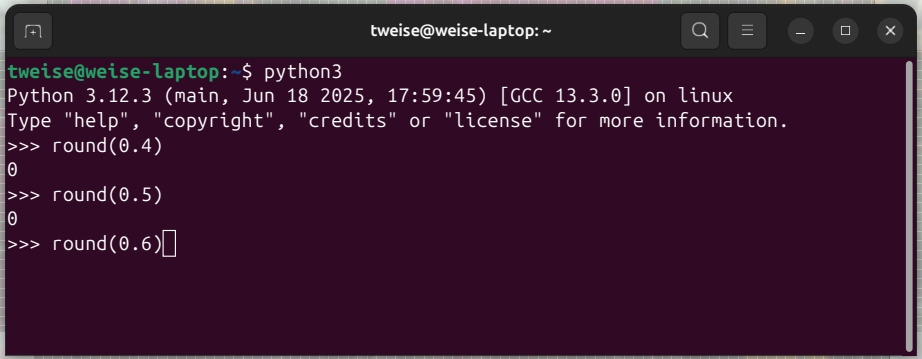
Runden

- 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.



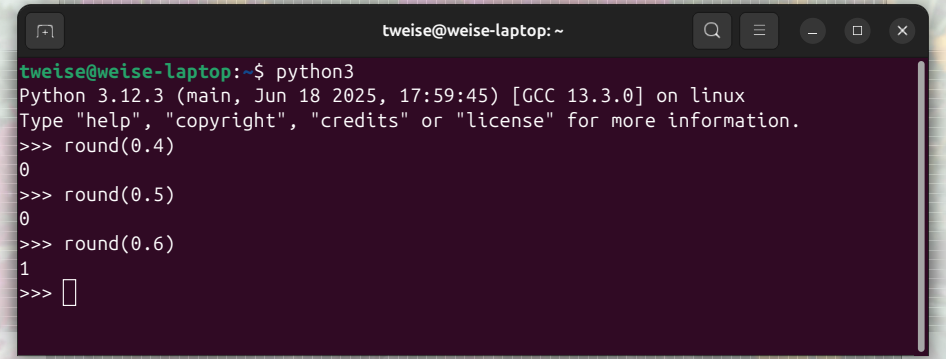
Runden

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



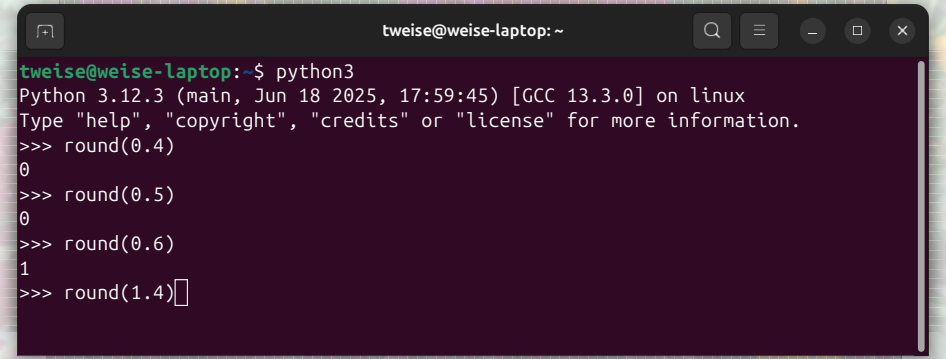
Runden

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



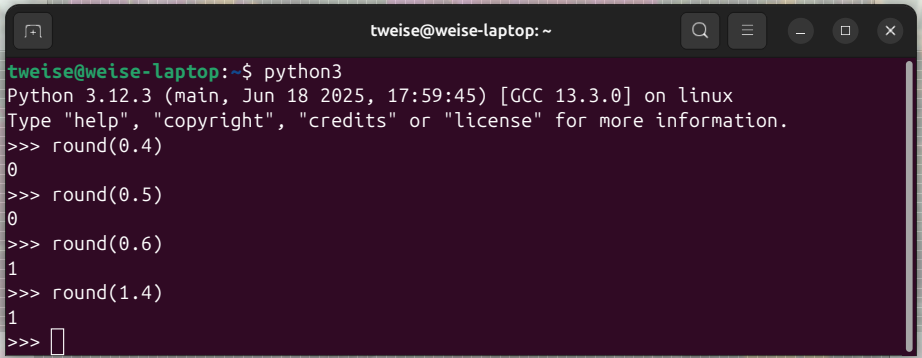
Runden

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



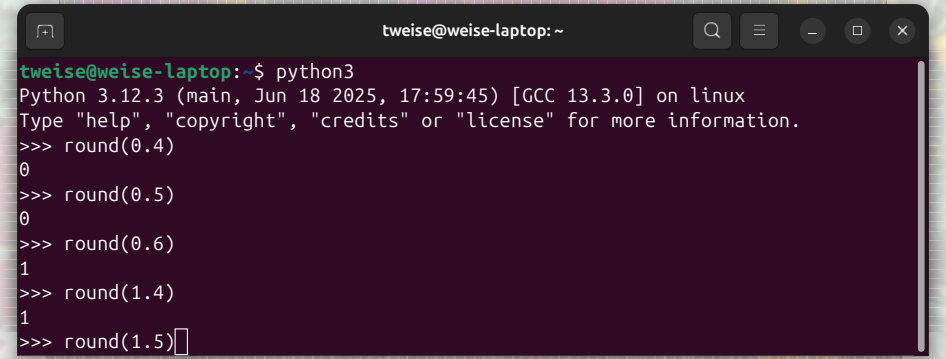
Runden

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



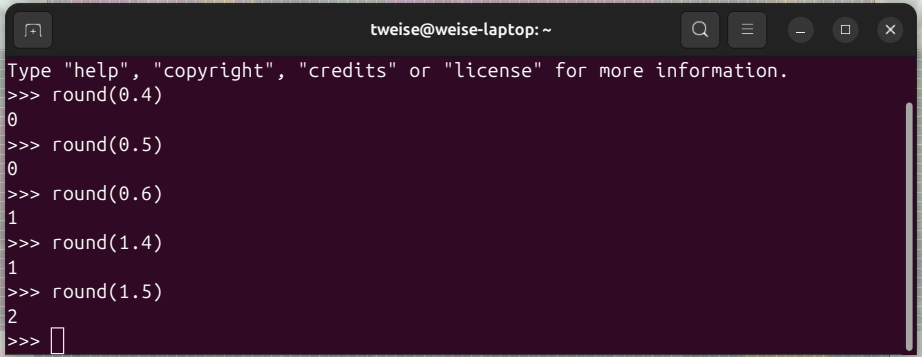
Runden

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



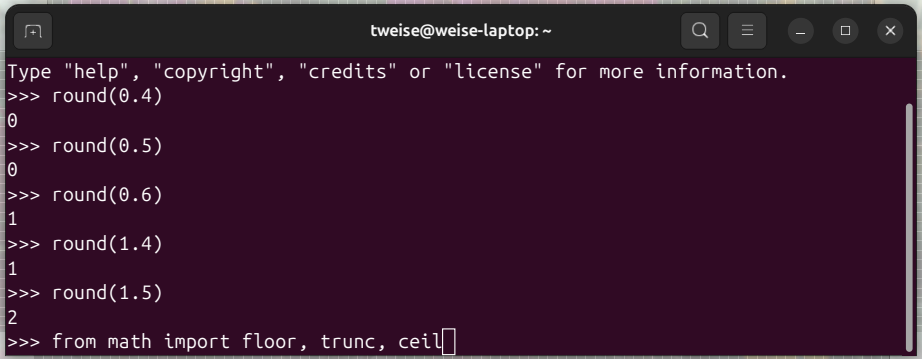
Runden

- 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...

tweise@weise-laptop: ~

```
>>> round(0.5)
0
>>> round(0.6)
1
>>> round(1.4)
1
>>> round(1.5)
2
>>> from math import floor, trunc, ceil
>>> floor(0.4)
0
>>> floor(0.5)
```

Runden

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

tweise@weise-laptop: ~

```
>>> round(0.6)
1
>>> round(1.4)
1
>>> round(1.5)
2
>>> from math import floor, trunc, ceil
>>> floor(0.4)
0
>>> floor(0.5)
0
>>>
```

Runden

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

tweise@weise-laptop: ~

```
>>> round(0.6)
1
>>> round(1.4)
1
>>> round(1.5)
2
>>> from math import floor, trunc, ceil
>>> floor(0.4)
0
>>> floor(0.5)
0
>>> floor(0.6)
```

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.

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.

Runden

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

Runden

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

tweise@weise-laptop: ~

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

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.

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.

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...

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.

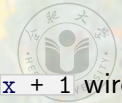
Runden

- 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

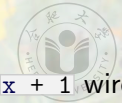
- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Haben wir `x = 12.5`, dann rechnen wir `int(12.5 + 0.5)` und es kommt `13` heraus.

Runden



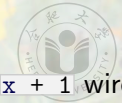
- Wollen wir runden, wie ich es in der Schule gelernt habe, nämlich das `x.5` zu `x + 1` wird, dann können wir einfach `int(x + 0.5)` oder `trunc(x + 0.5)` rechnen.
- Damit sind wir mit dem Thema "runden" fertig.

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[18]...

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[14,17] 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.

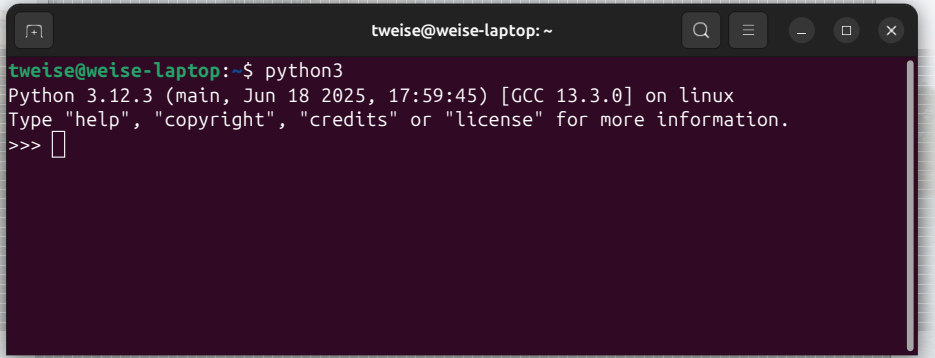
Die Wissenschaftliche Notation: Arbeitsweise

- 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 βeγ.
- 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... $* 10^{-HIJ}$

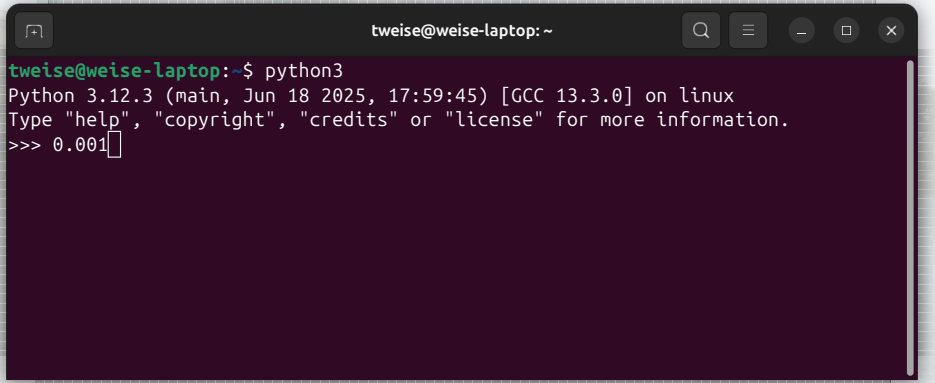
Die Wissenschaftliche Notation: Beispiele

- Probieren wir das mal aus.



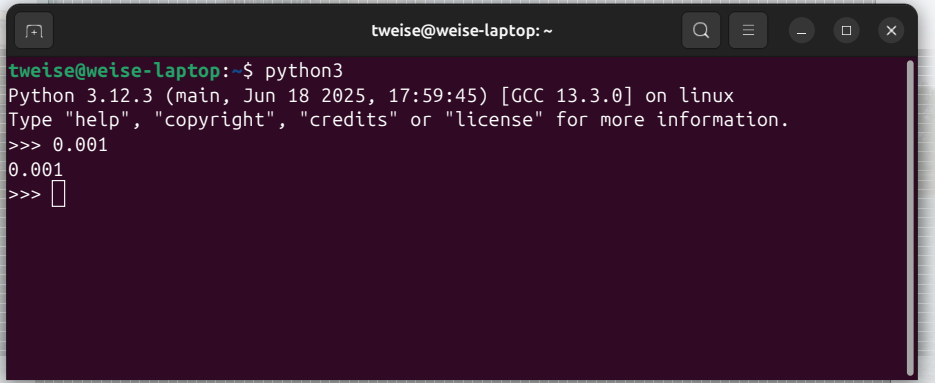
Die Wissenschaftliche Notation: Beispiele

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



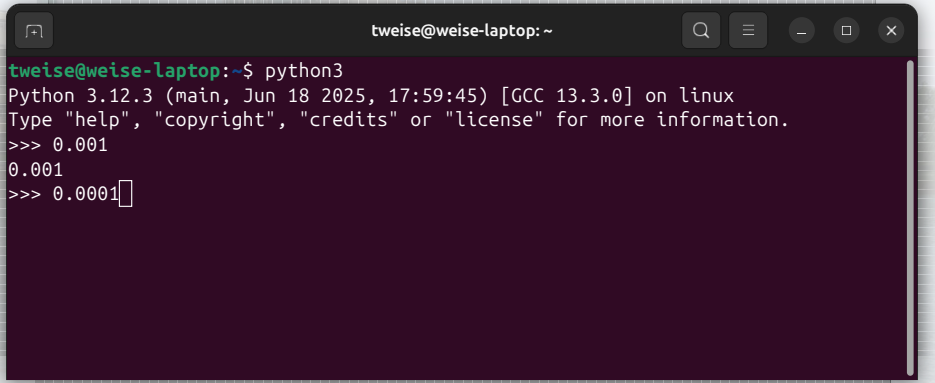
Die Wissenschaftliche Notation: Beispiele

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



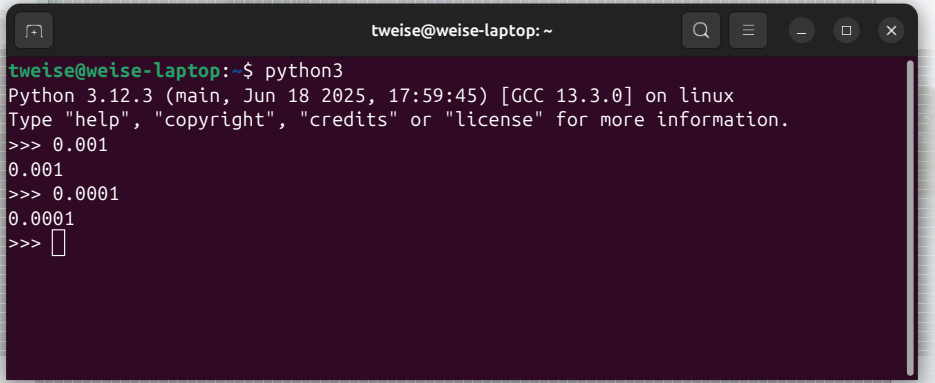
Die Wissenschaftliche Notation: Beispiele

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



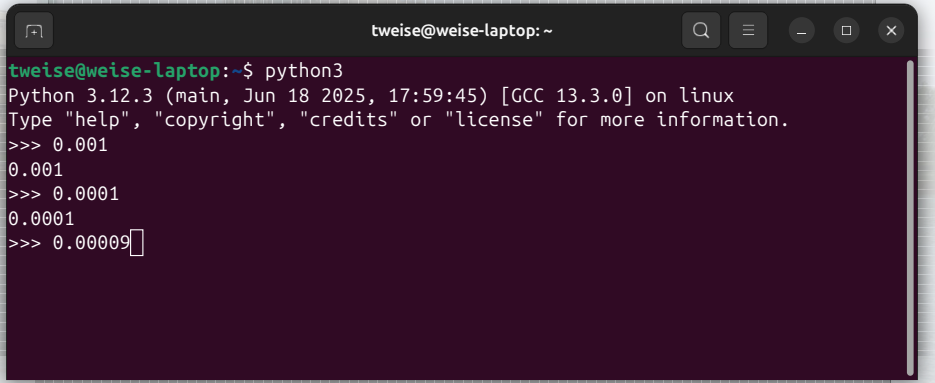
Die Wissenschaftliche Notation: Beispiele

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



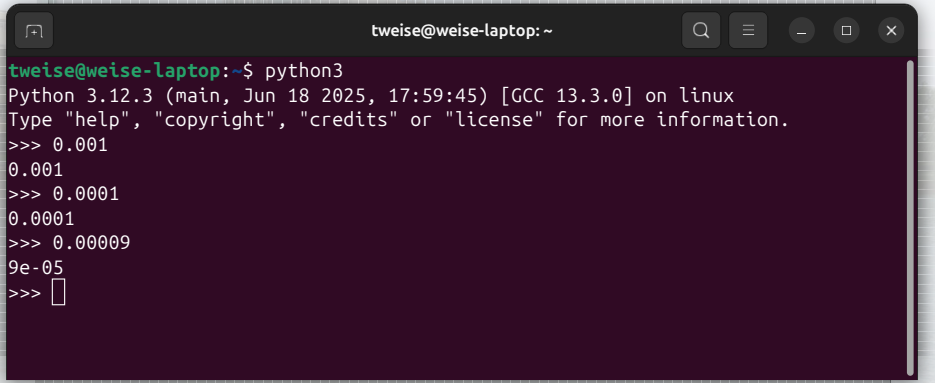
Die Wissenschaftliche Notation: Beispiele

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



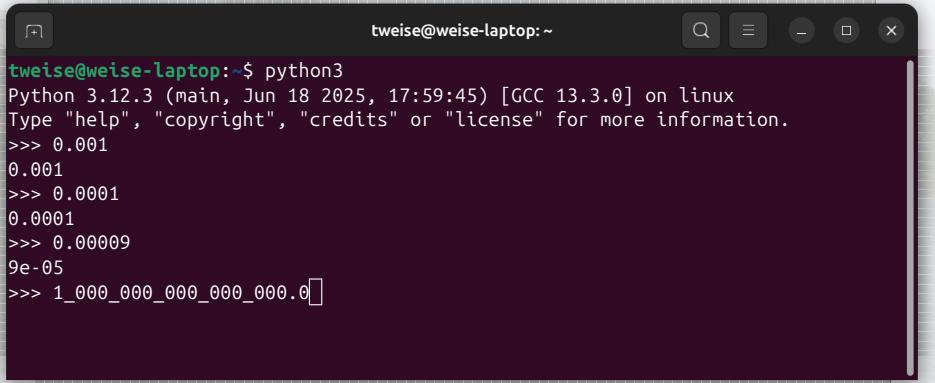
Die Wissenschaftliche Notation: Beispiele

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



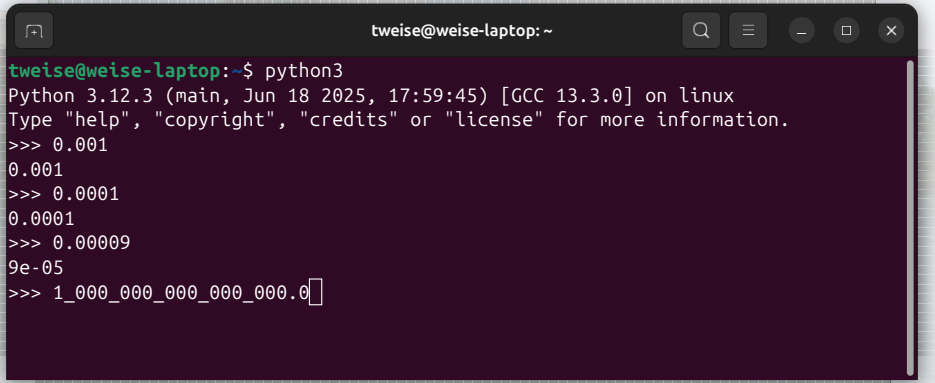
Die Wissenschaftliche Notation: Beispiele

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



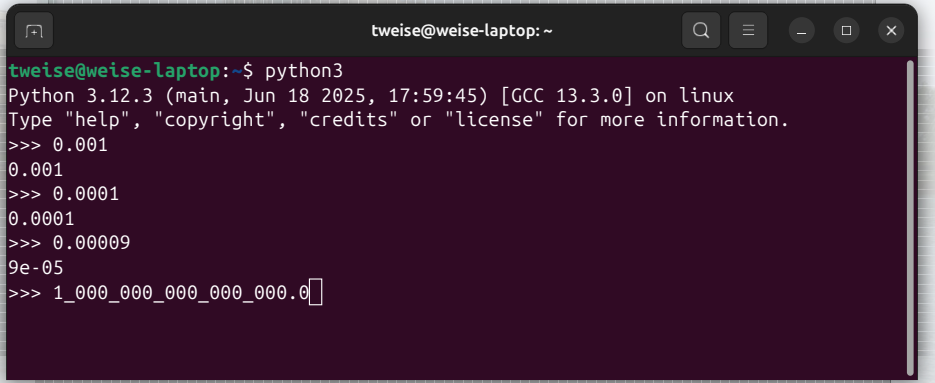
Die Wissenschaftliche Notation: Beispiele

- 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
>>>
```

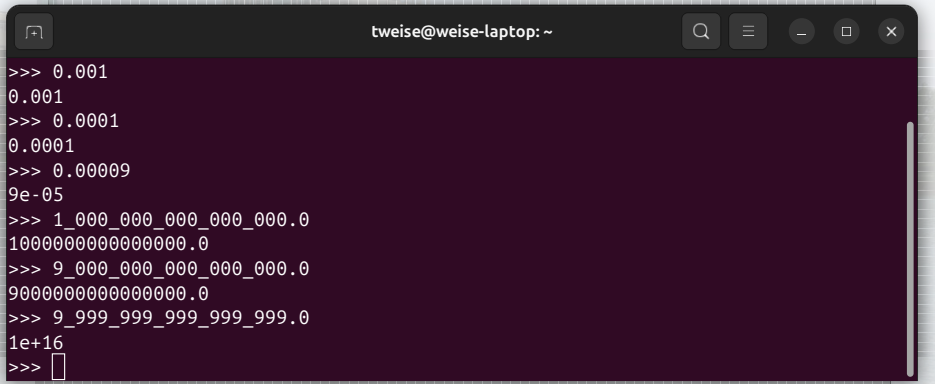
Die Wissenschaftliche Notation: Beispiele

- 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...

```
tweise@weise-laptop: ~
>>> 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
1e+16
>>> 10_000_000_000_000_000.0
```

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

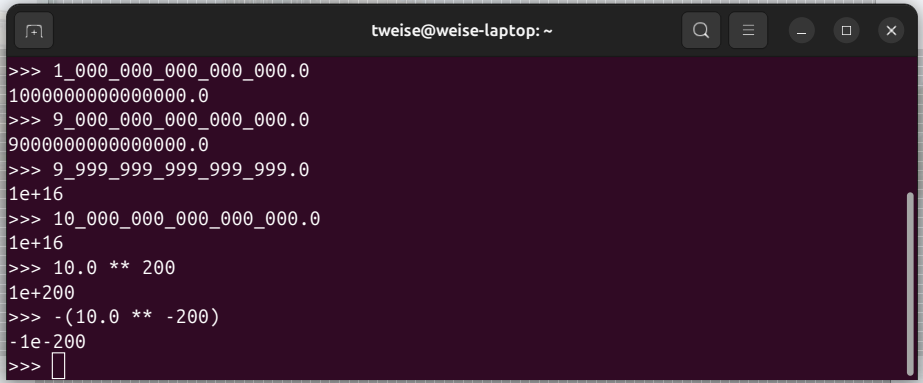
- Berechnen wir 10^{200} als `float` in dem wir schreiben `10.0 ** 200`, dann schreibt es `1e+200` in der wissenschaftlichen Notation.

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

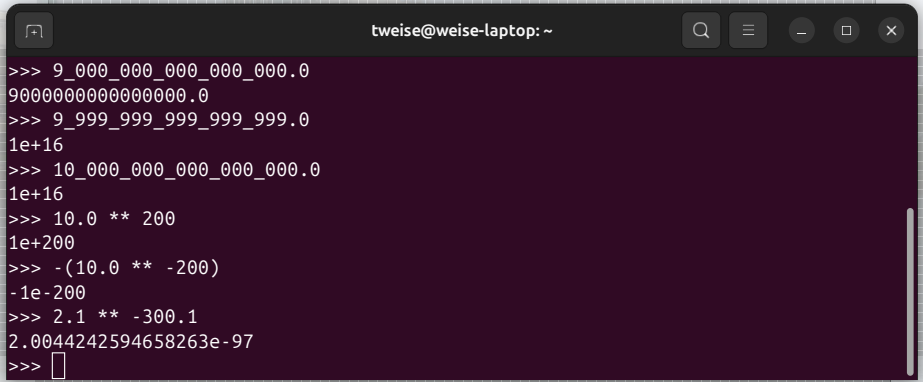
- Berechnen wir $2.1^{-300.1}$ als `float` in dem wir schreiben `2.1 ** -300.1`...

tweise@weise-laptop: ~

```
>>> 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
1e+16
>>> 10_000_000_000_000_000.0
1e+16
>>> 10.0 ** 200
1e+200
>>> -(10.0 ** -200)
-1e-200
>>> 2.1 ** -300.1
```

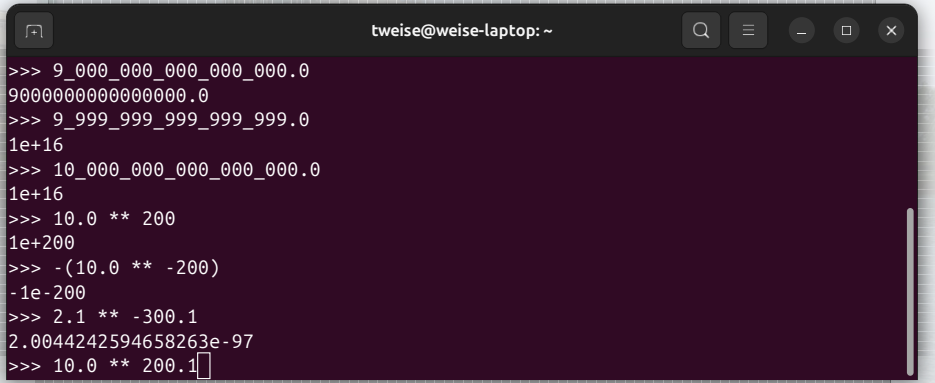
Die Wissenschaftliche Notation: Beispiele

- 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.



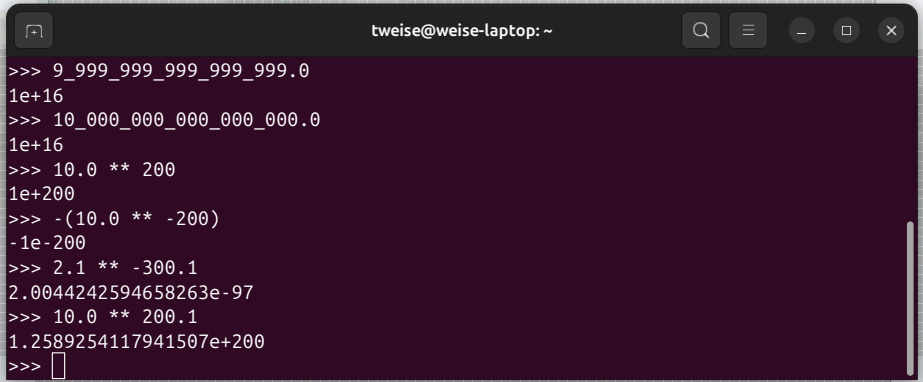
Die Wissenschaftliche Notation: Beispiele

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



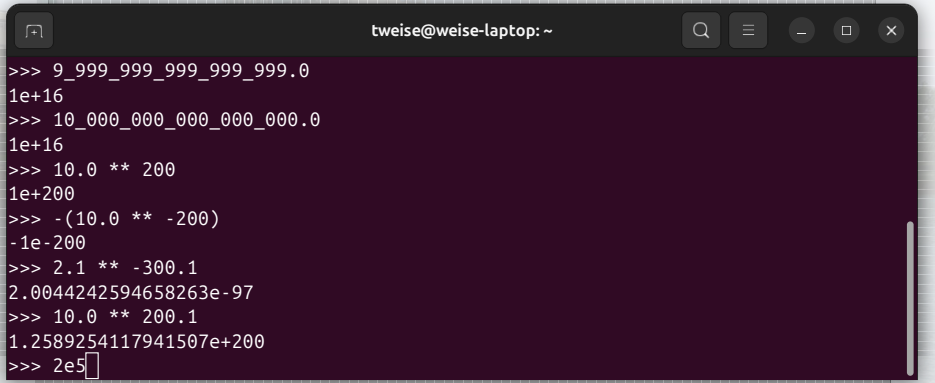
Die Wissenschaftliche Notation: Beispiele

- 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.
- Genauso, wie es Python egal ist, ob ein `int`-Wert in hexadezimaler, dezimaler, oktaler, oder binärer Schreibweise eingegeben wurde, ist es Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder "normal".

Die Wissenschaftliche Notation: Beispiele

- Damit haben wie die wissenschaftliche Notation verstanden.
- Genauso, wie es Python egal ist, ob ein `int`-Wert in hexadezimaler, dezimaler, oktaler, oder binärer Schreibweise eingegeben wurde, ist es Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder "normal".
- 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.
- Genauso, wie es Python egal ist, ob ein `int`-Wert in hexadezimaler, dezimaler, oktaler, oder binärer Schreibweise eingegeben wurde, ist es Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder "normal".
- 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.
- Genauso, wie es Python egal ist, ob ein `int`-Wert in hexadezimaler, dezimaler, oktaler, oder binärer Schreibweise eingegeben wurde, ist es Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder "normal".
- 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.
- Genauso, wie es Python egal ist, ob ein `int`-Wert in hexadezimaler, dezimaler, oktaler, oder binärer Schreibweise eingegeben wurde, ist es Python egal, ob wir `float`-Werte in wissenschaftlicher Notation schreiben oder "normal".
- 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 Zifferen zu trennen[4]. `37_859_378` ist z.B. viel leicher zu lesen als is much easier to read than `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[17] für seinen Datentyp `double`.

Sehr kleine Zahlen

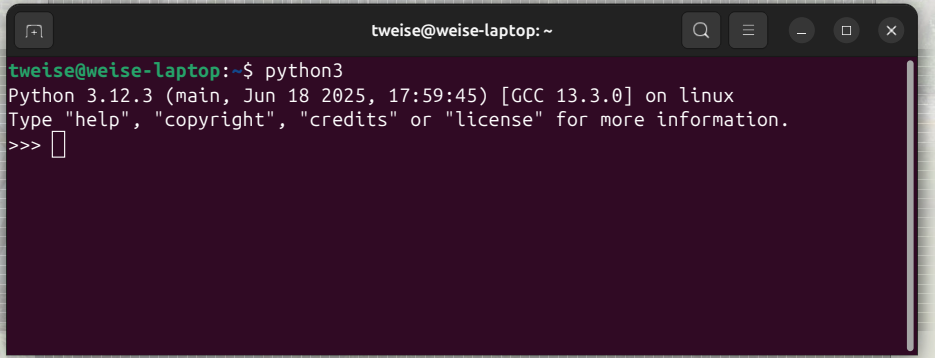
- Wir wissen, dass der Fließkommatyp `float` sowohl sehr kleine als auch sehr große Zahlen darstellen kann.
- Intern ist er 64 Bits groß, also ist sein Wertebereich natürlich begrenzt.
- Was passiert, wenn wir diese Grenzen überschreiten?
- Python kann sehr kleine Zahlen, z.B. 10^{-300}, als `floats` speichern.
- Aber wie klein *genau*?
- Java benutzt den selben IEEE 754 Standard[17] für seinen Datentyp `double`.
- In der entsprechenden Dokumentation[6] 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[17] für seinen Datentyp `double`.
- In der entsprechenden Dokumentation[6] 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.

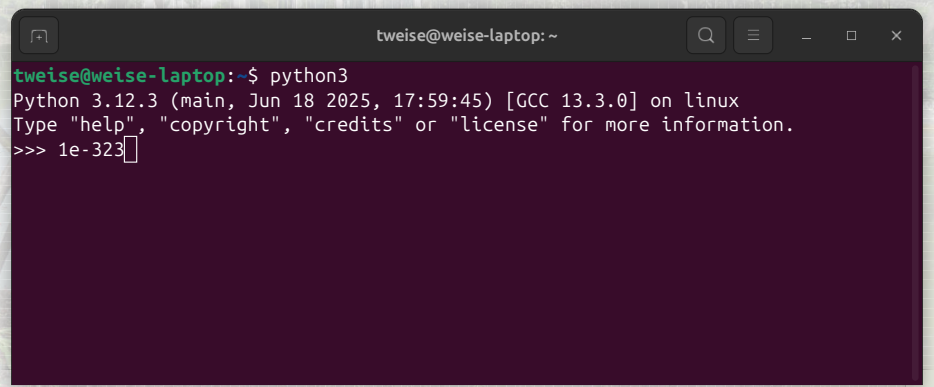
Sehr kleine Zahlen: Ausprobieren

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



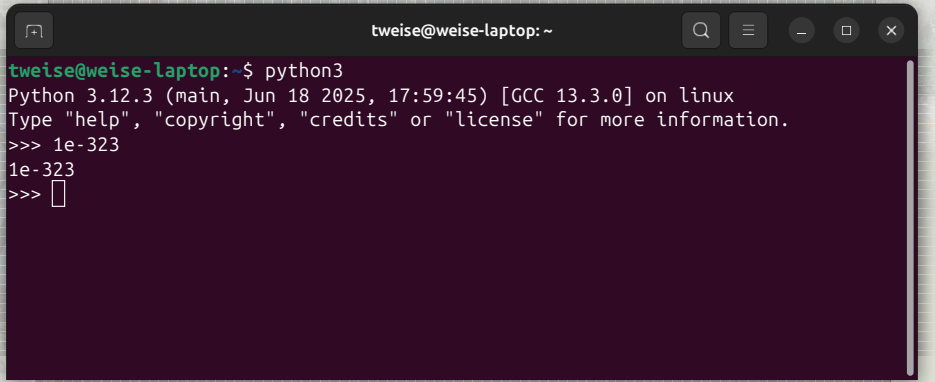
Sehr kleine Zahlen: Ausprobieren

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



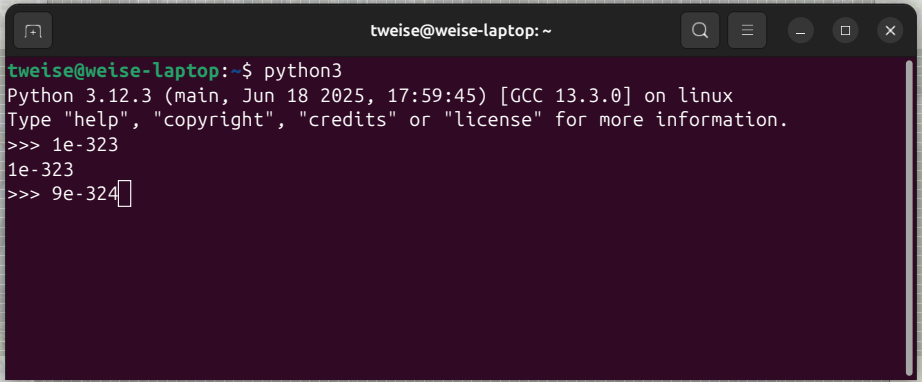
Sehr kleine Zahlen: Ausprobieren

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



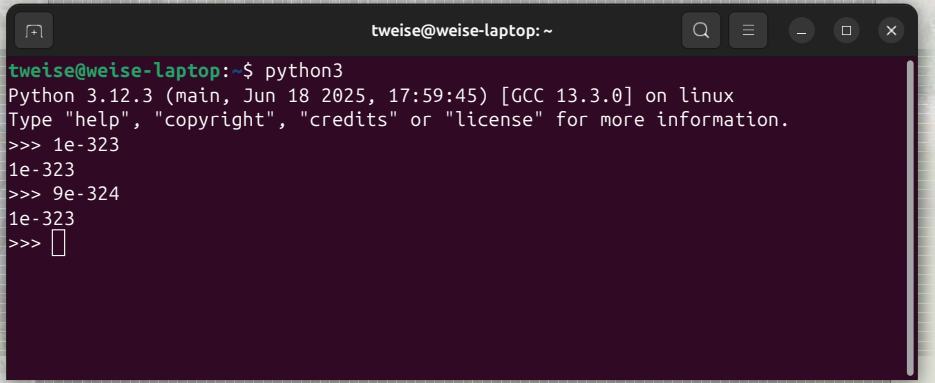
Sehr kleine Zahlen: Ausprobieren

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



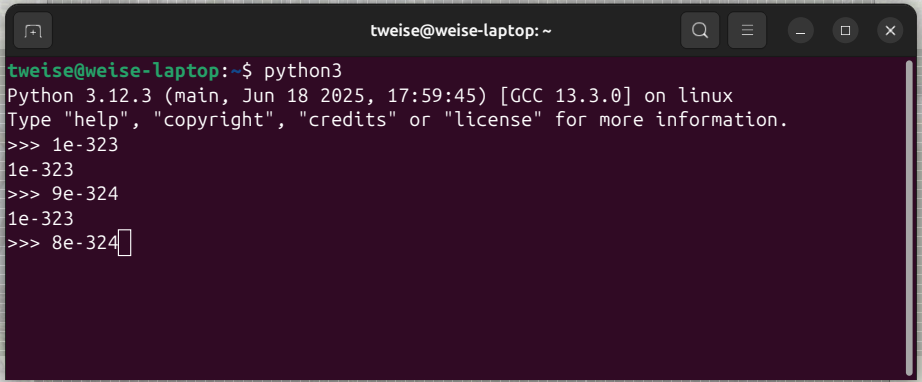
Sehr kleine Zahlen: Ausprobieren

- 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 Significand genutzt werden und viele Zahlen zusammenfallen.



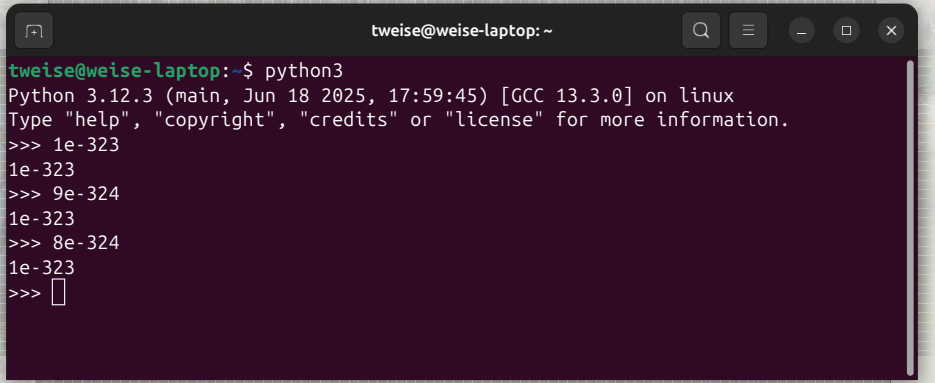
Sehr kleine Zahlen: Ausprobieren

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



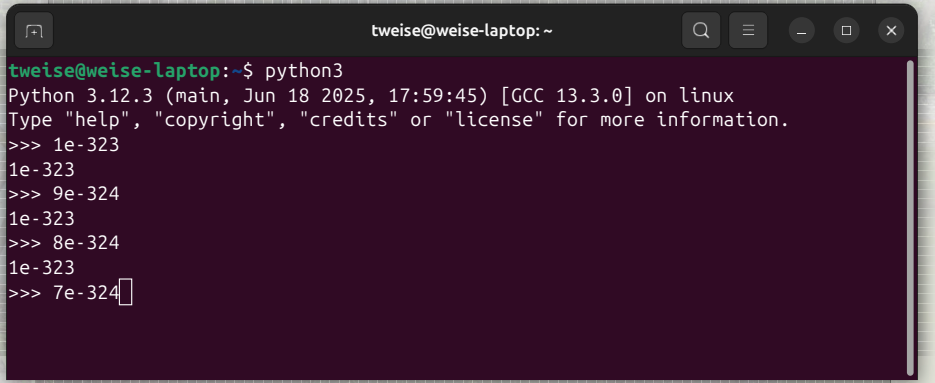
Sehr kleine Zahlen: Ausprobieren

- 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 Significand genutzt werden und viele Zahlen zusammenfallen.



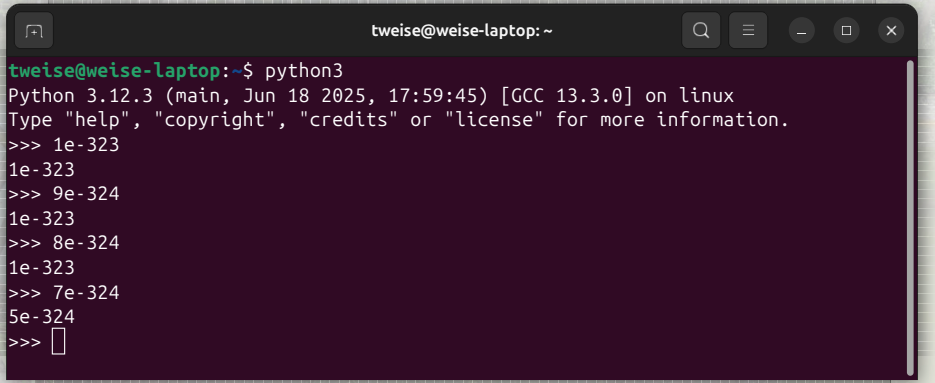
Sehr kleine Zahlen: Ausprobieren

- Wir geben `7e-324` ein



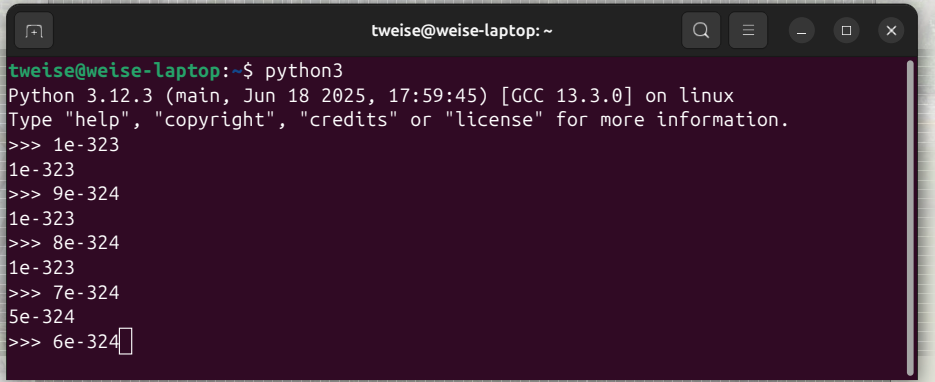
Sehr kleine Zahlen: Ausprobieren

- 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 Significand genutzt werden und viele Zahlen zusammenfallen.



Sehr kleine Zahlen: Ausprobieren

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

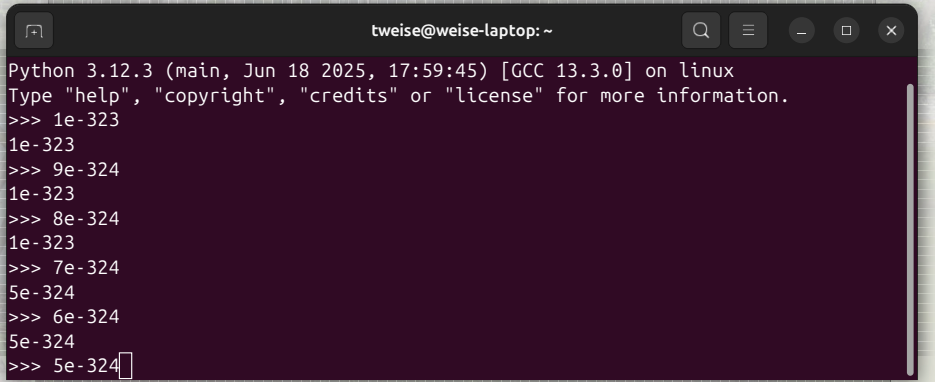


Sehr kleine Zahlen: Ausprobieren

- 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 Significand 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 kleinse `double`-Zahl in Java[6]...

Sehr kleine Zahlen: Ausprobieren

- Wir geben `4.94065645841246544e-324` ein, nämlich die kleinse `double`-Zahl in Java[6] und Python gibt uns wieder `5e-324` aus.

```
tweise@weise-laptop: ~
>>> 9e-324
1e-323
>>> 8e-324
1e-323
>>> 7e-324
5e-324
>>> 6e-324
5e-324
>>> 5e-324
5e-324
>>> 4.94065645841246544e-324
5e-324
>>>
```

Sehr kleine Zahlen: Ausprobieren

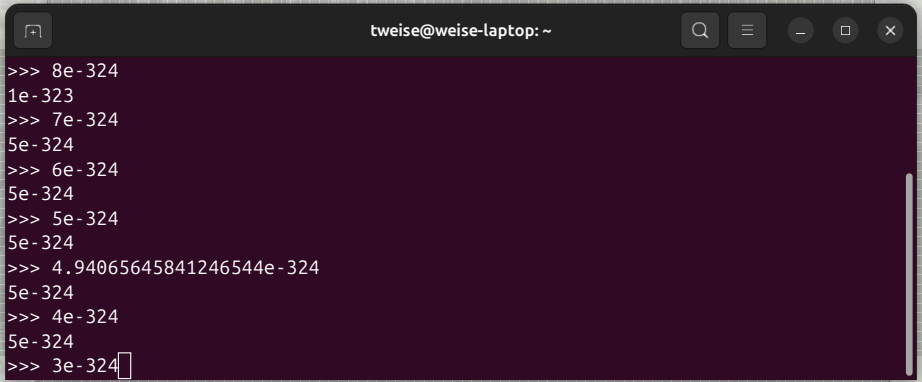
- Wir geben `4e-324` ein...

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...



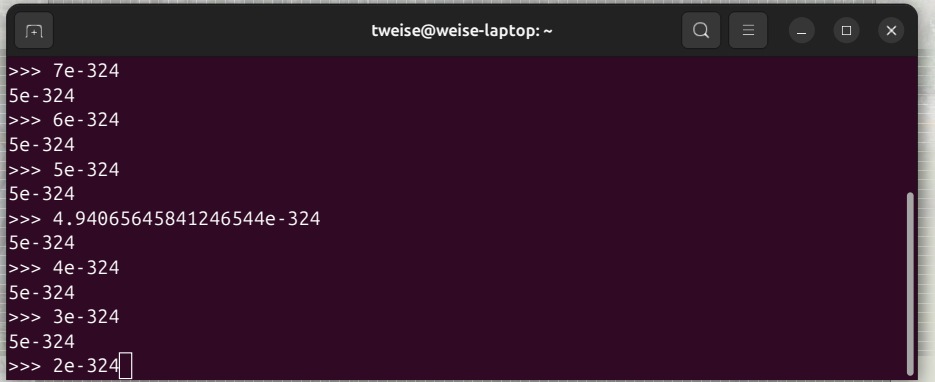
Sehr kleine Zahlen: Ausprobieren

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

```
tweise@weise-laptop: ~
>>> 7e-324
5e-324
>>> 6e-324
5e-324
>>> 5e-324
5e-324
>>> 4.94065645841246544e-324
5e-324
>>> 4e-324
5e-324
>>> 3e-324
5e-324
>>>
```

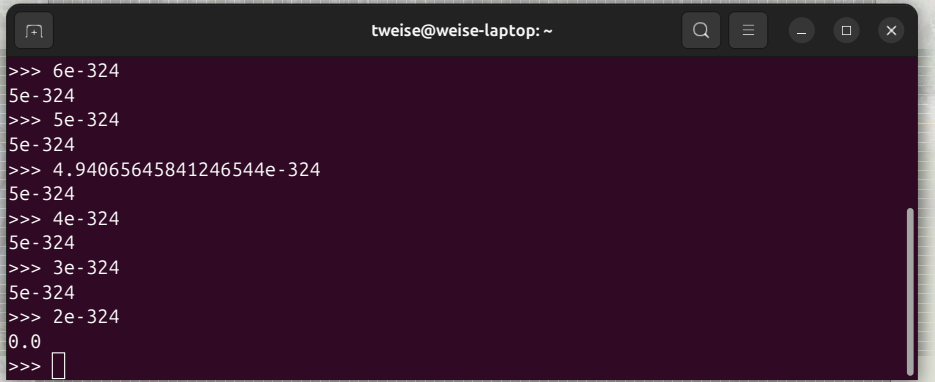
Sehr kleine Zahlen: Ausprobieren

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



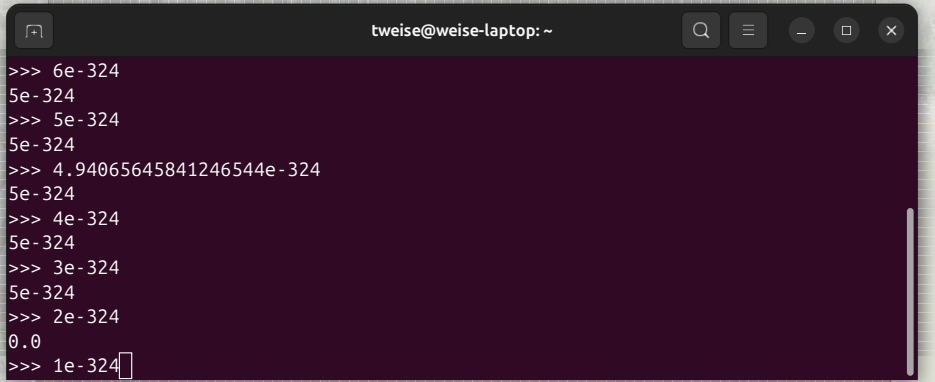
Sehr kleine Zahlen: Ausprobieren

- 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

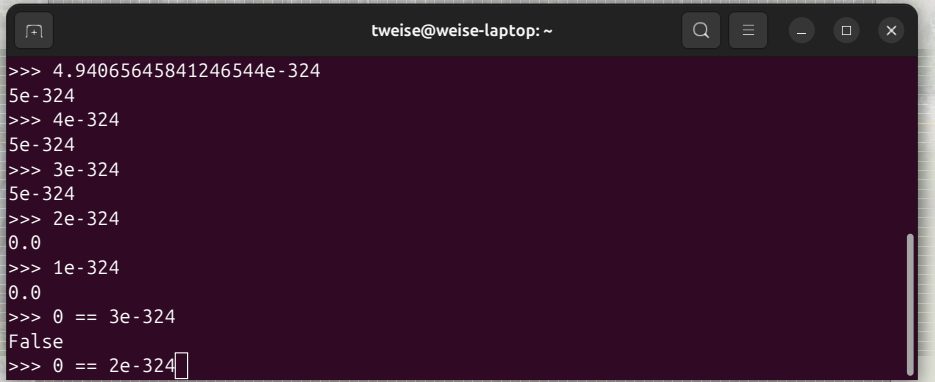
- Wir geben `0 == 3e-324` ein...

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 Significand 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

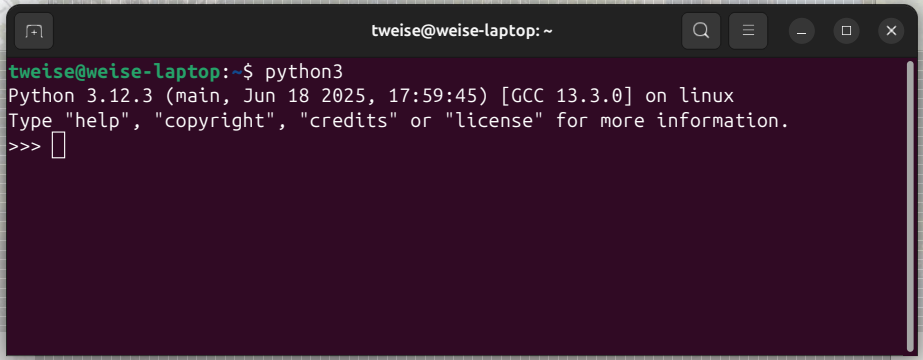
- Für kleine Zahlen gibt es ein natürliches Ende: sehr sehr kleine Zahlen sind einfach 0.
- Was passiert mit sehr großen Zahlen?
- Die Java-Dokumentation[6] nennt $(2 - 2^{-52}) * 2^{1023} \approx 1.797\,693\,134\,862\,315\,708 \cdots * 10^{308}$ als die größte Zahl, die mit 64-Bit Fließkommazahlen dargstellt 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[6] 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?

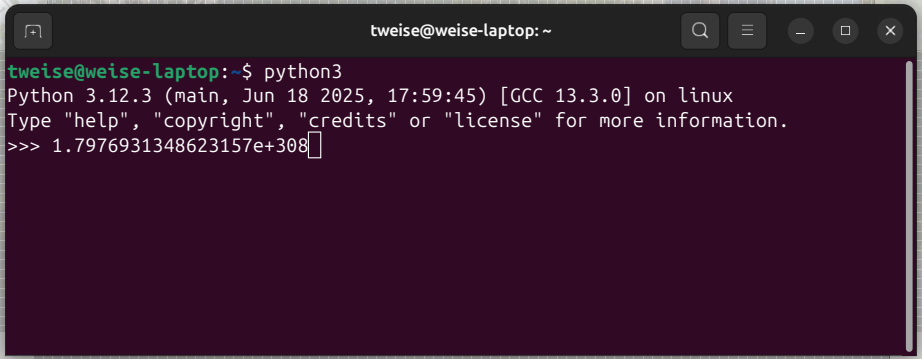
Sehr große Zahlen: Ausprobieren

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



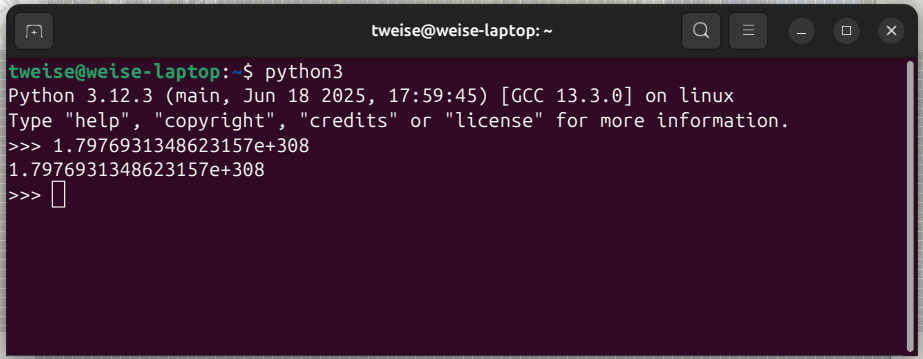
Sehr große Zahlen: Ausprobieren

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



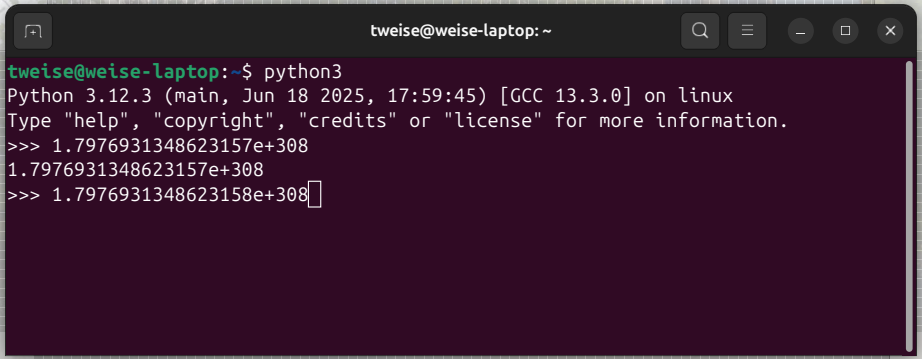
Sehr große Zahlen: Ausprobieren

- 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.



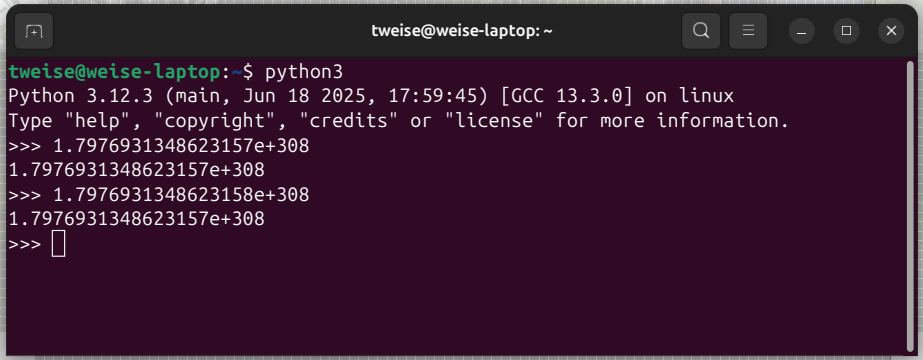
Sehr große Zahlen: Ausprobieren

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



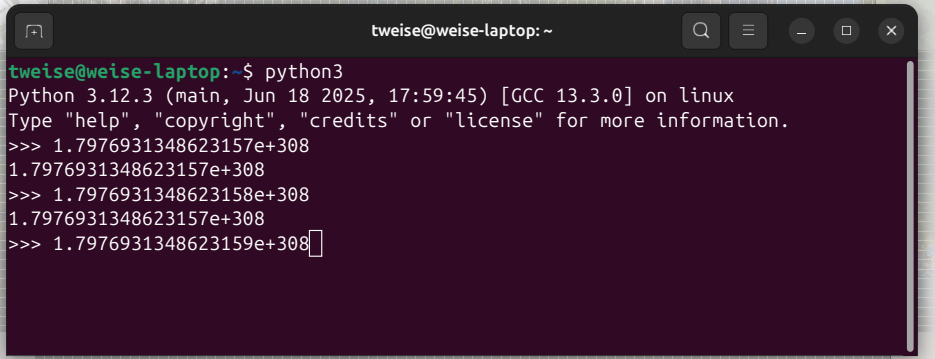
Sehr große Zahlen: Ausprobieren

- Wir geben `1.7976931348623158e+308` ein, also 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.



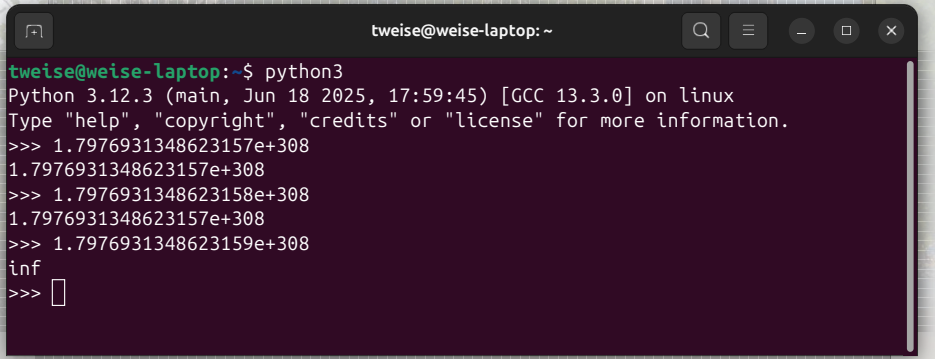
Sehr große Zahlen: Ausprobieren

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



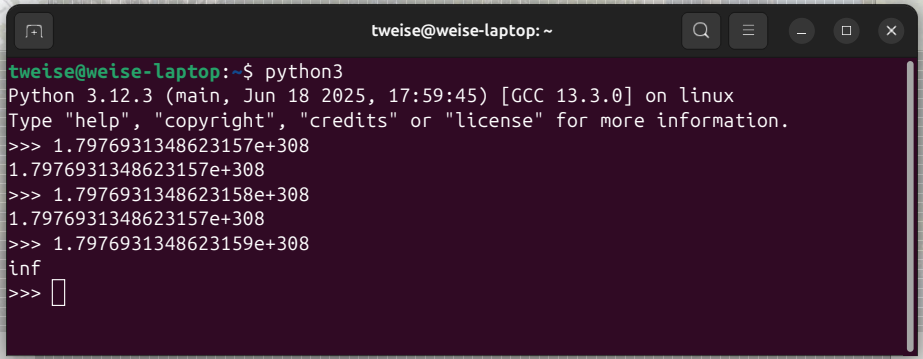
Sehr große Zahlen: Ausprobieren

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



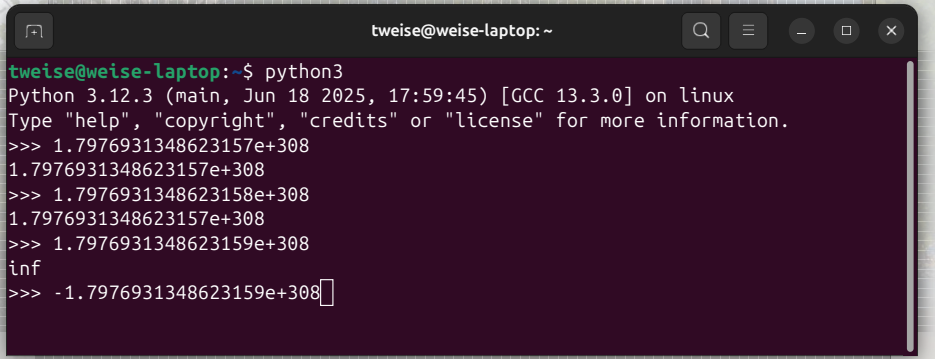
Sehr große Zahlen: Ausprobieren

- 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`.



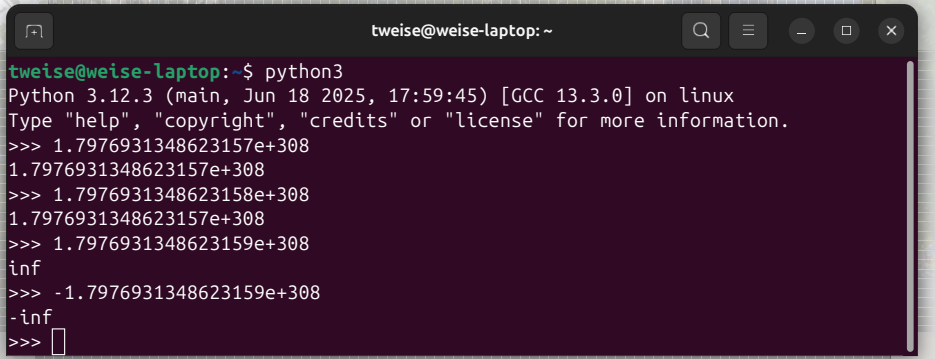
Sehr große Zahlen: Ausprobieren

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



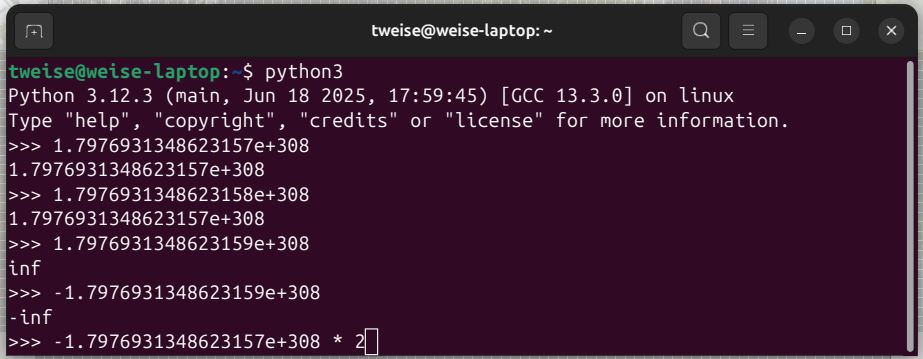
Sehr große Zahlen: Ausprobieren

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



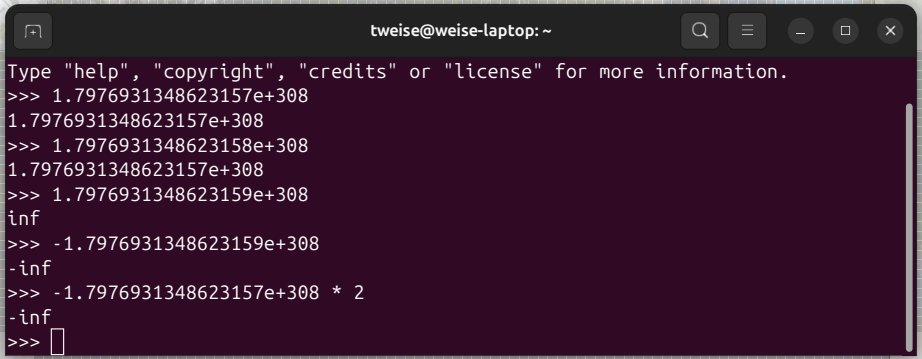
Sehr große Zahlen: Ausprobieren

- Wenn wir rechnen `-1.7976931348623157e+308 * 3`...



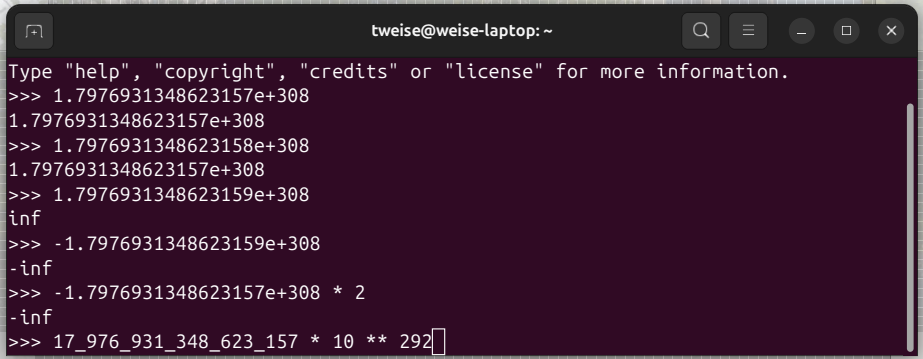
Sehr große Zahlen: Ausprobieren

- Wenn wir rechnen `-1.7976931348623157e+308 * 3`, 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.

```
tweise@weise-laptop: ~
>>> 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…

```
tweise@weise-laptop: ~
>>> 1.7976931348623159e+308
inf
>>> -1.7976931348623159e+308
-inf
>>> -1.7976931348623157e+308 * 2
-inf
>>> 17_976_931_348_623_157 * 10 ** 292
179769313486231570000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
```

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.

```
>>> -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...

```
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
>>> 17_976_931_348_623_157 * 10 ** 293
```

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.

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.)

```
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
179769313486231570000000000000000000000000000000000000000000000000000000000000000
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
>>>
```

Sehr große Zahlen: Ausprobieren

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

tweise@weise-laptop: ~

```
>>> (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.

```
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. . .

```
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
>>> 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.

```
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
>>> 
```

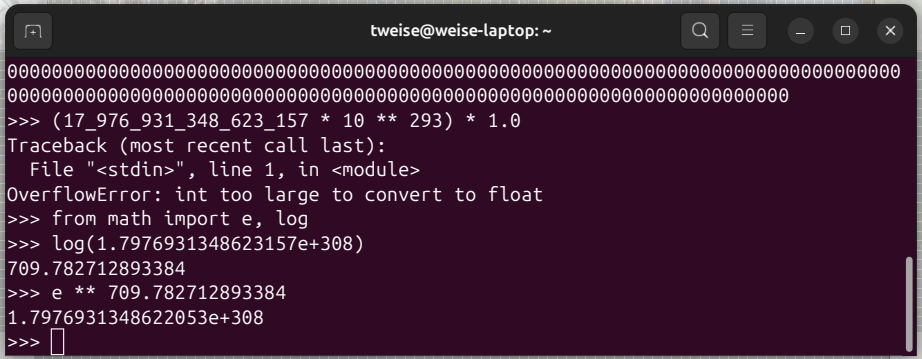
Sehr große Zahlen: Ausprobieren

- `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
```

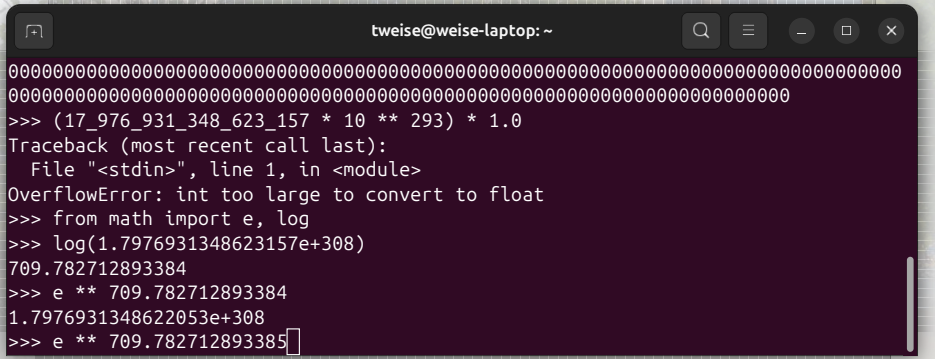
Sehr große Zahlen: Ausprobieren

- `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`.



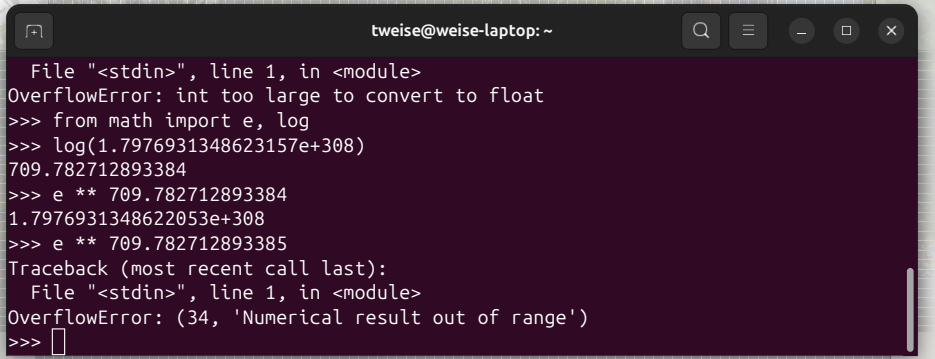
Sehr große Zahlen: Ausprobieren

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



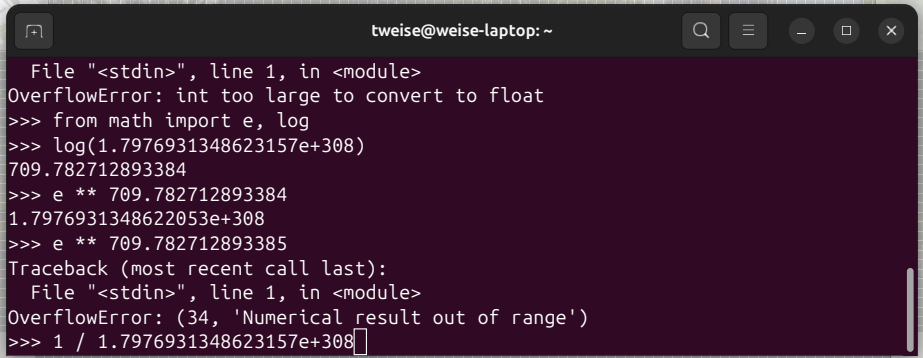
Sehr große Zahlen: Ausprobieren

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



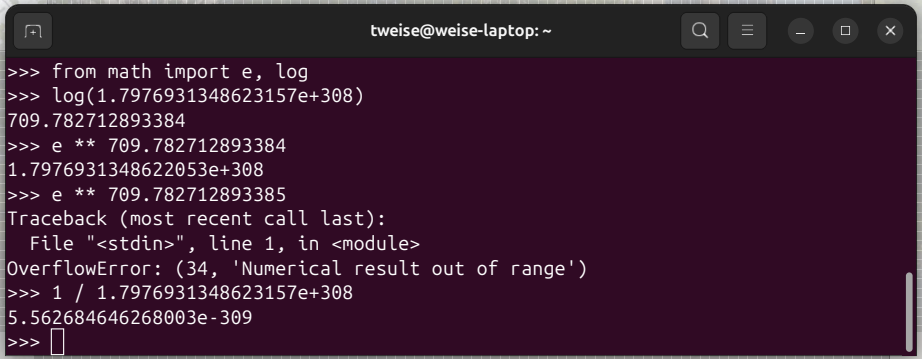
Sehr große Zahlen: Ausprobieren

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



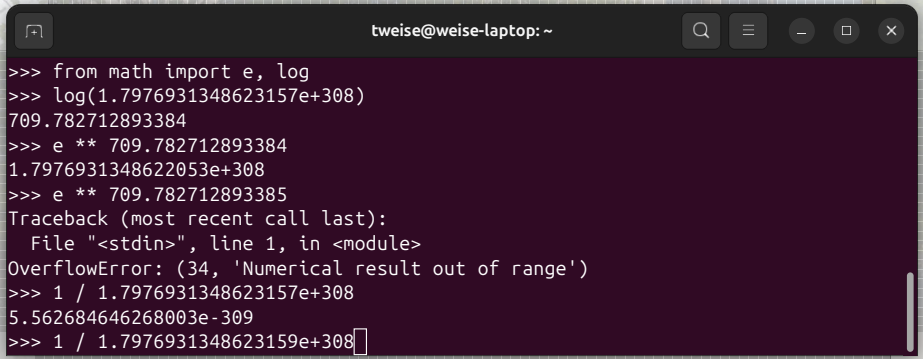
Sehr große Zahlen: Ausprobieren

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



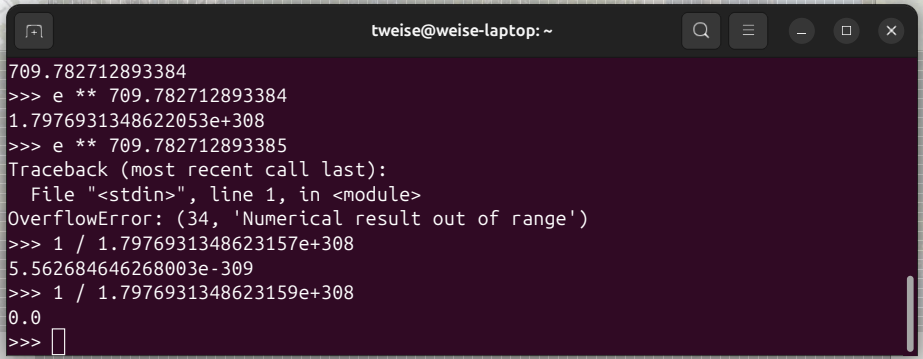
Sehr große Zahlen: Ausprobieren

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



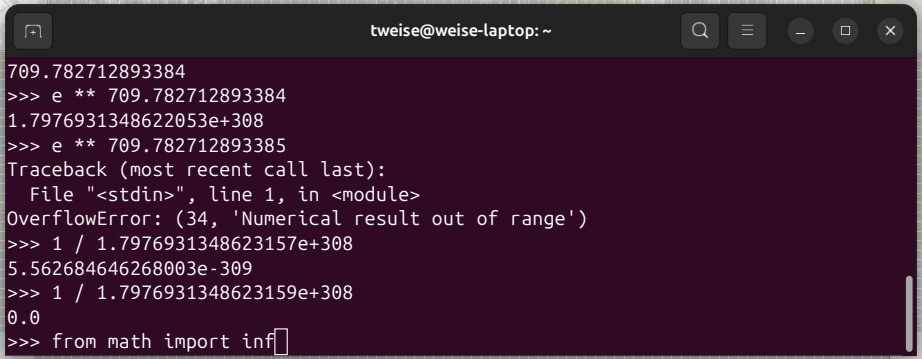
Sehr große Zahlen: Ausprobieren

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



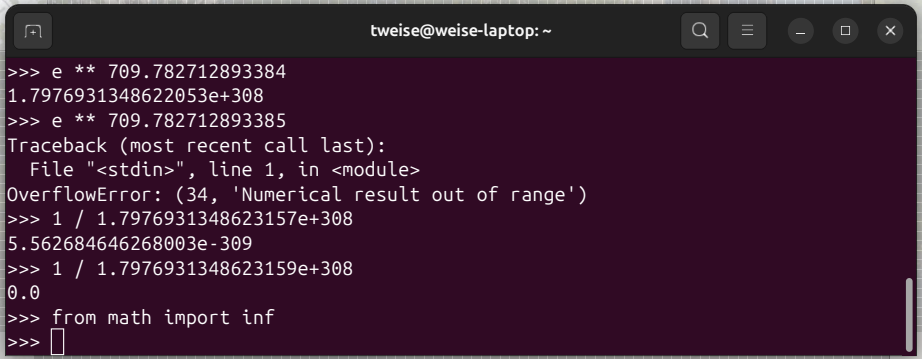
Sehr große Zahlen: Ausprobieren

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



Sehr große Zahlen: Ausprobieren

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

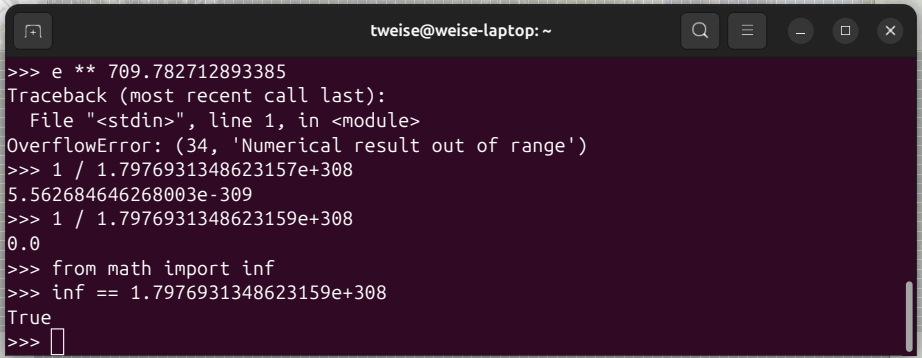


Sehr große Zahlen: Ausprobieren

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

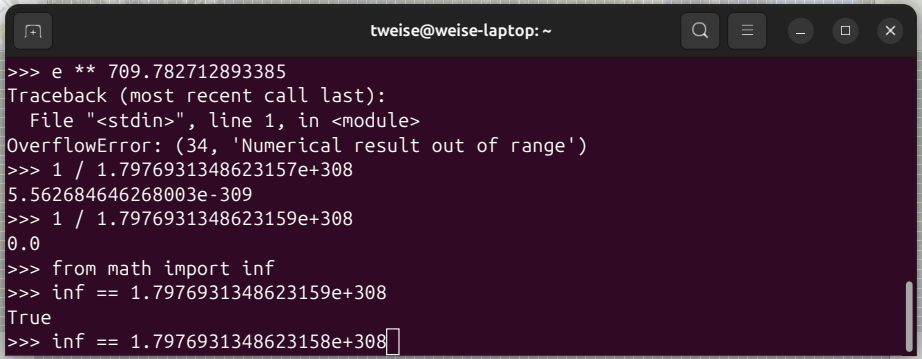
Sehr große Zahlen: Ausprobieren

- 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 Kontante 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 Kontante 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 Kontante 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 Kontante 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 Kontante 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 Kontante 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 Berrechnung direkt stopp.

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 Kontante 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 Berrechnung direkt stopp.
- 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 Kontante 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 Berrechnung direkt stopp.
- 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!

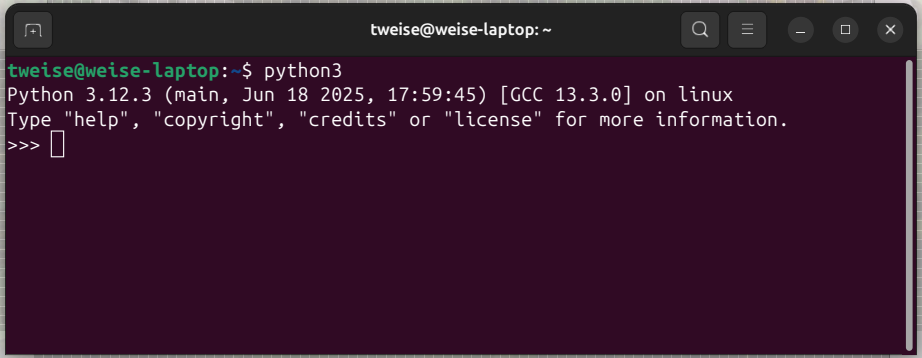
- Also gut, wir wissen jetzt, dass sehr große Zahlen irgendwann den Wertebereich von `float` zum überlaufen bringen und zu `inf` werden.
- Nun haben wir gesagt, dass `inf` praktisch alle Zahlen umfasst, die zu groß für den Wertebereich von `float` sind.
- Das sind sehr viele Zahlen.
- Es schließt auch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

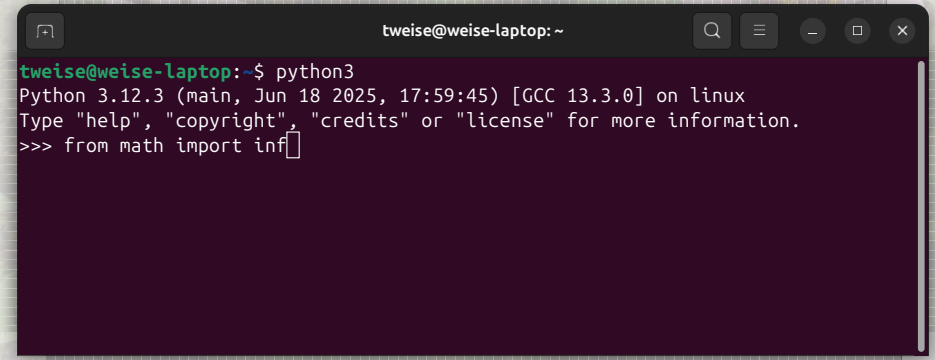
Das ist keine Zahl! – Ausprobieren

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



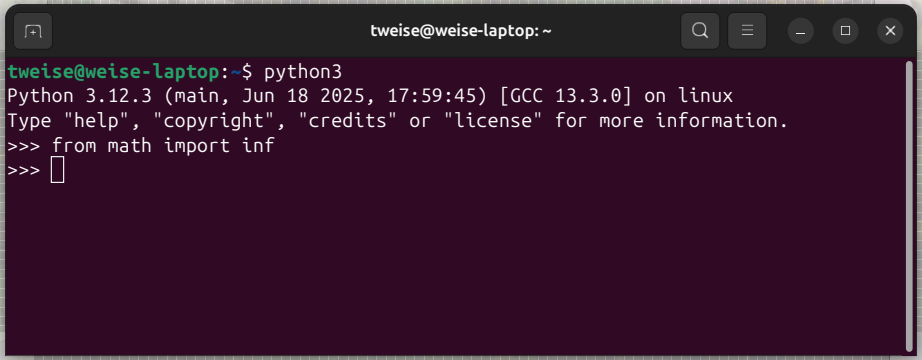
Das ist keine Zahl! – Ausprobieren

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



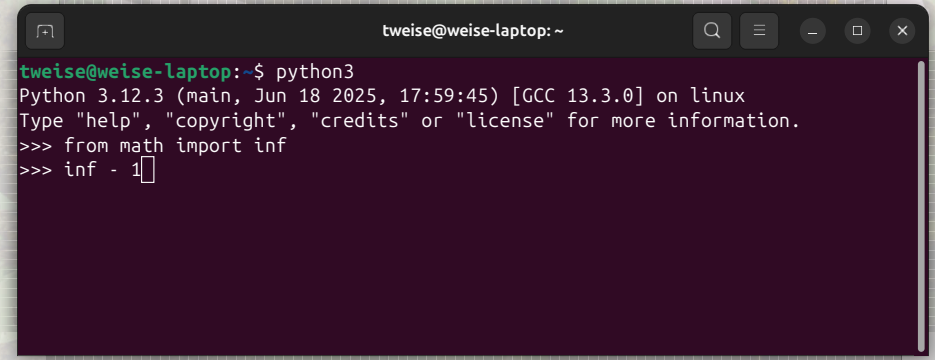
Das ist keine Zahl! – Ausprobieren

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



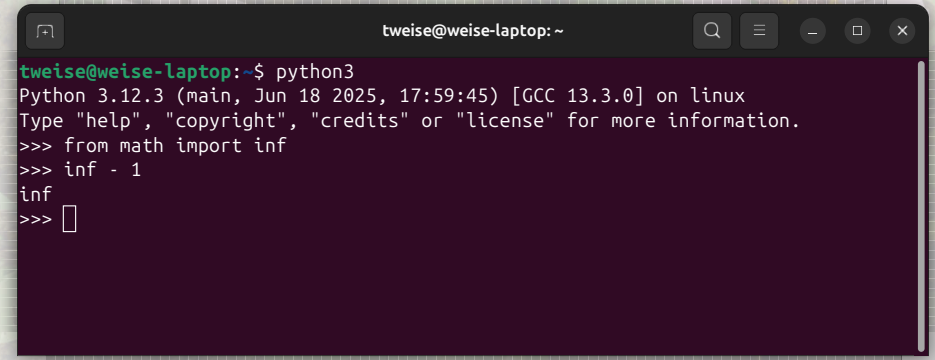
Das ist keine Zahl! – Ausprobieren

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



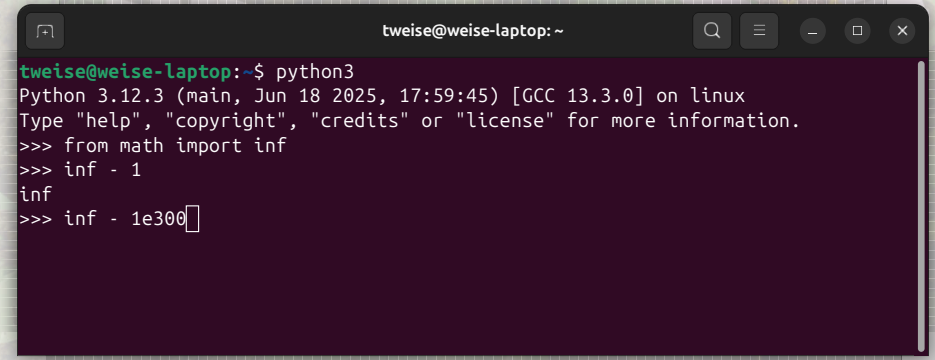
Das ist keine Zahl! – Ausprobieren

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



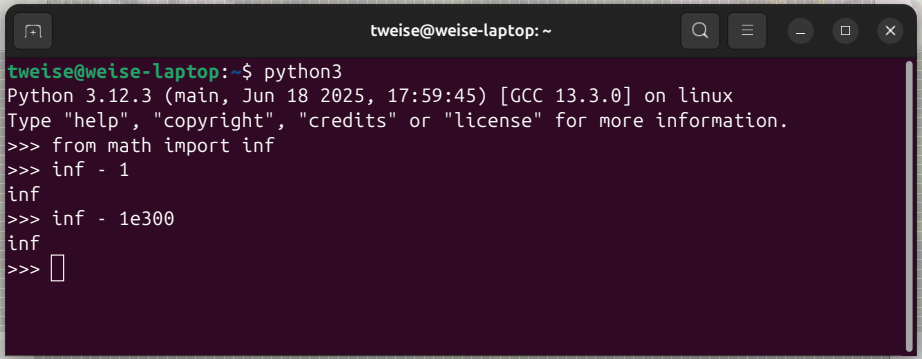
Das ist keine Zahl! – Ausprobieren

- `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...



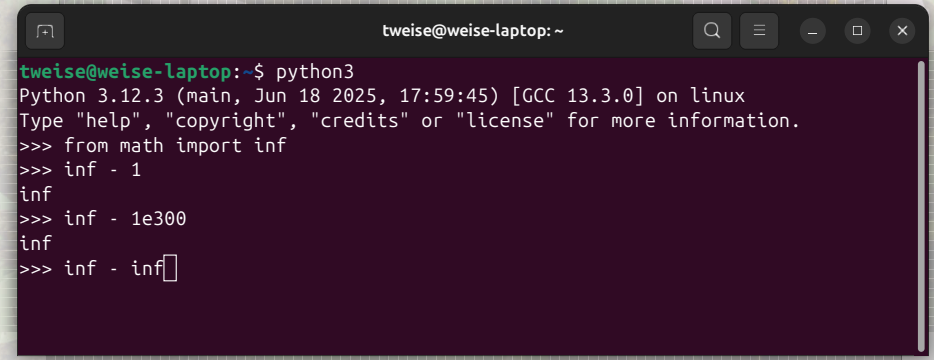
Das ist keine Zahl! – Ausprobieren

- `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 $+\infty$.



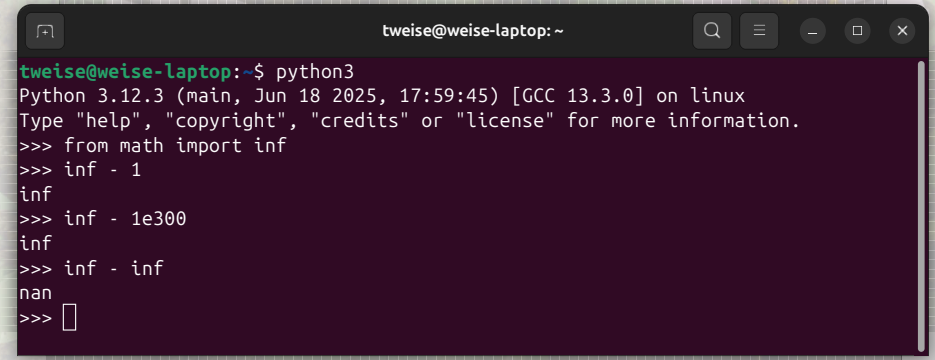
Das ist keine Zahl! – Ausprobieren

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



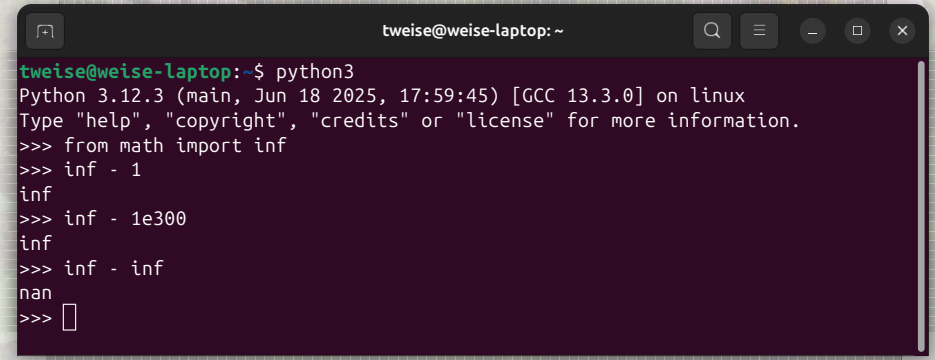
Das ist keine Zahl! – Ausprobieren

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



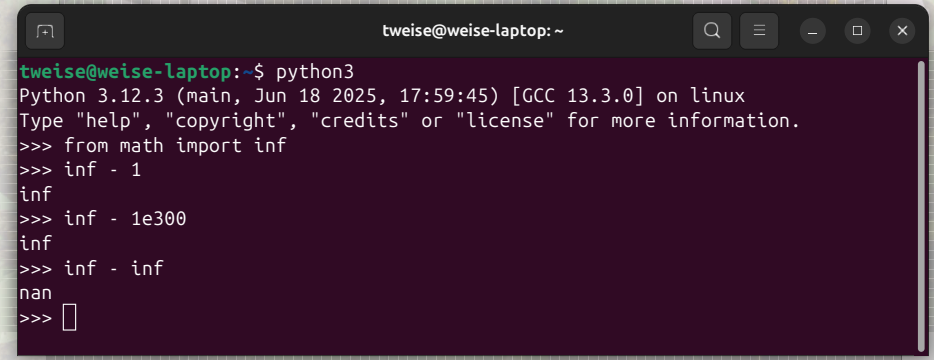
Das ist keine Zahl! – Ausprobieren

- 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!"*.



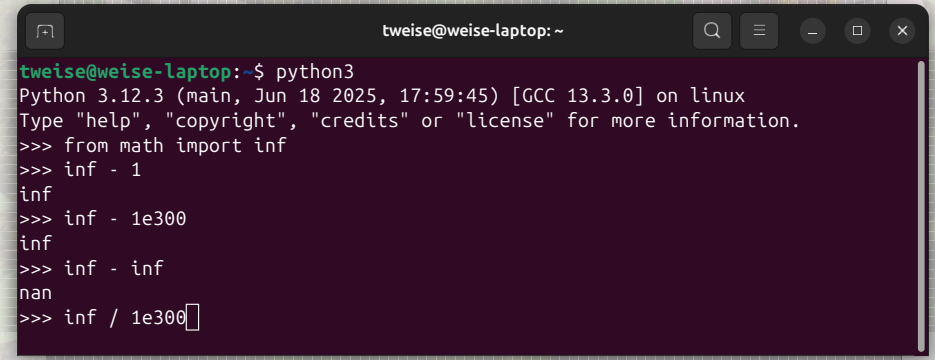
Das ist keine Zahl! – Ausprobieren

- 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!"*.
- Kommt irgendwo in einer Rechnung `nan` vor, dann ist das Ergebnis immer auch `nan`.



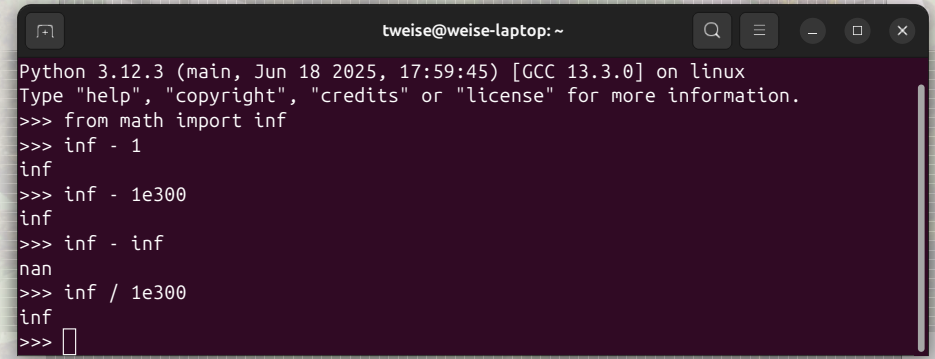
Das ist keine Zahl! – Ausprobieren

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



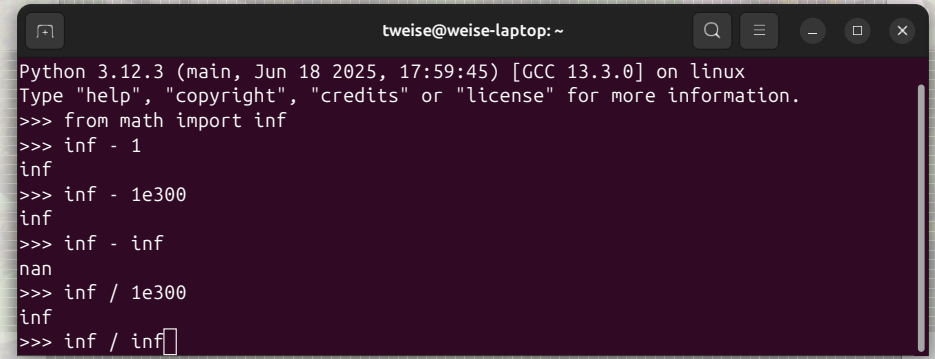
Das ist keine Zahl! – Ausprobieren

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



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`.

tweise@weise-laptop: ~

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

Das ist keine Zahl! – Ausprobieren

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

Das ist keine Zahl! – Ausprobieren

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

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`?

Das ist keine Zahl! – Ausprobieren

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

Das ist keine Zahl! – Ausprobieren

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

Das ist keine Zahl! – Ausprobieren

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

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?

Das ist keine Zahl! – Ausprobieren

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

Das ist keine Zahl! – Ausprobieren

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

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...

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 `ints` 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 `ints` 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 `ints` 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 `info` 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 `info` 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 `info` 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 `info` 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 `info` 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 `info` 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[7].

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 `info` 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[7].

Wo kommen `inf` und `nan` her?

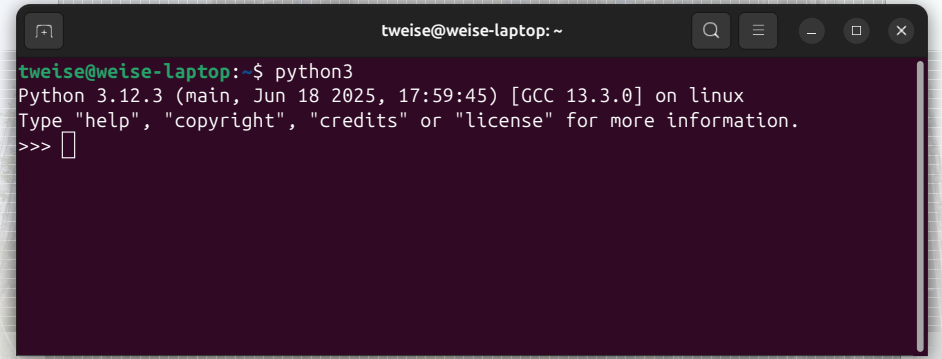
- OK, das Ergebnis einer Berechnung könnte also `inf` oder `nan` sein.
- Wir mussten einige komische Verrenkungen machen, um diese Werte zu produzieren.
- Es gibt vier grundlegende Szenarios, weshalb `info` oder `nan` in unseren Berechnungen vorkommen können.:
  1. Weil es so seien soll. Vielleicht sind 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[7].
- 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 `info` 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[7].
- 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`.

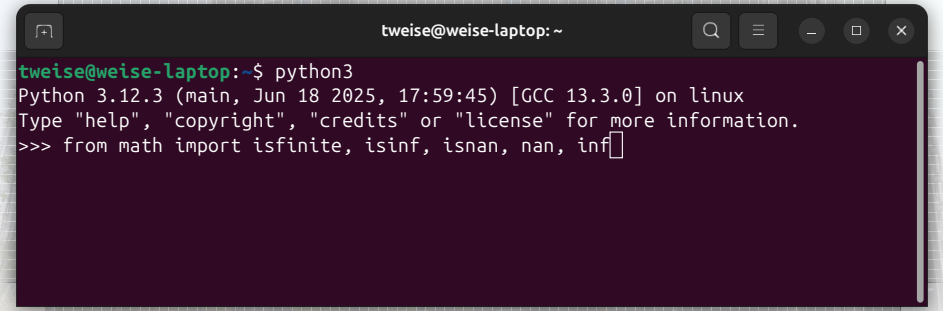
Testen auf inf und nan: Ausprobieren

- Probieren wir das mal aus.



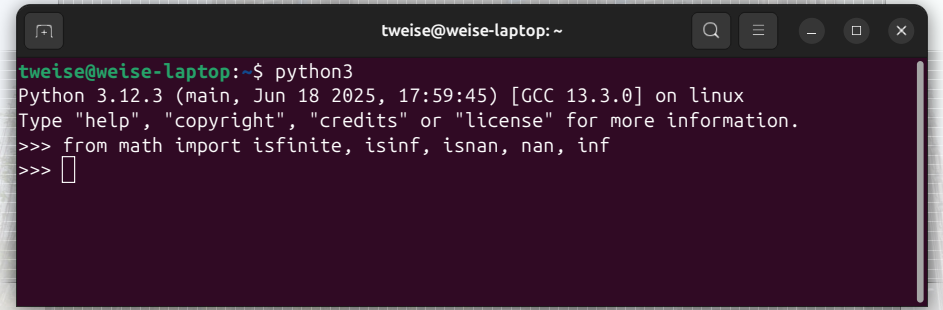
Testen auf inf und nan: Ausprobieren

- 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

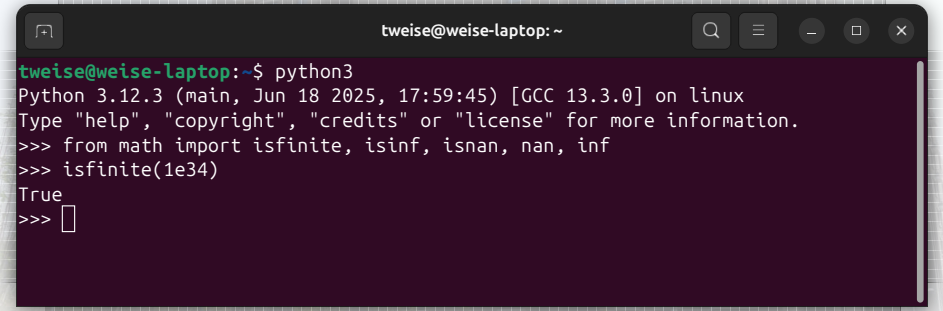
- Mit `isfinite` prüfen wir, ob eine Zahl finite ist, also weder unendlich noch undefiniert...

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 isfinite, isinf, isnan, nan, inf
>>> isfinite(1e34)
```

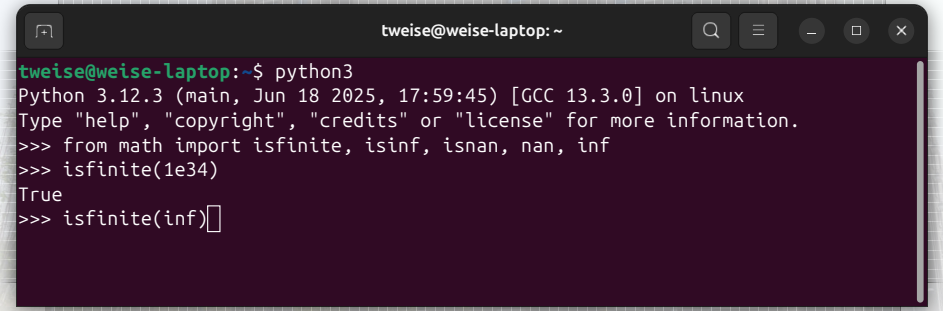
Testen auf inf und nan: Ausprobieren

- Mit `isfinite` prüfen wir, ob eine Zahl finite ist, also weder unendlich noch undefiniert. `isfinite(0.3)` ist daher `True`.



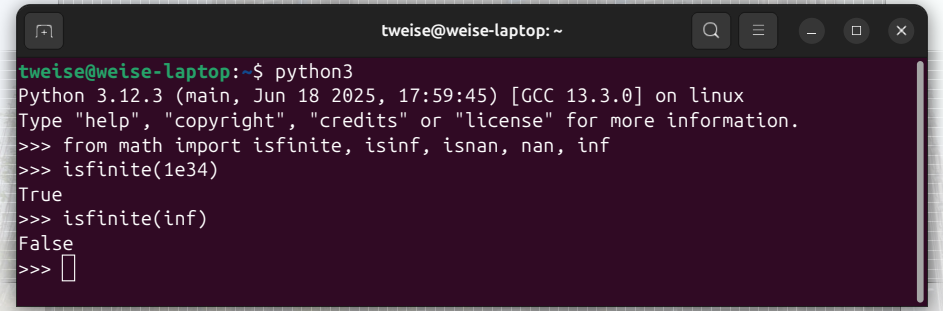
Testen auf inf und nan: Ausprobieren

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



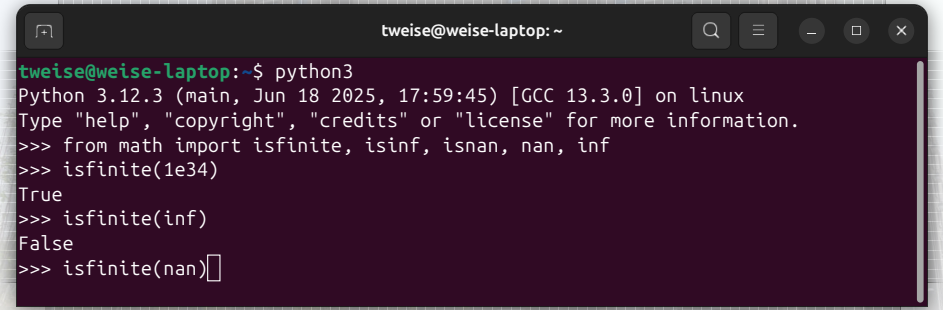
Testen auf inf und nan: Ausprobieren

- 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`.

tweise@weise-laptop: ~

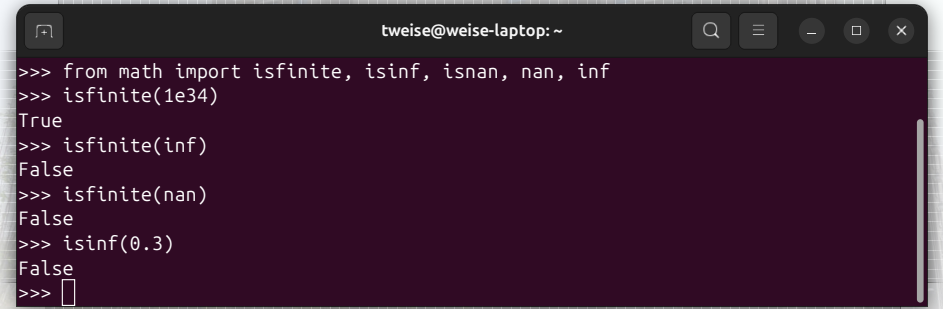
```
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import isfinite, isinf, isnan, nan, inf
>>> isfinite(1e34)
True
>>> isfinite(inf)
False
>>> isfinite(nan)
False
>>>
```

Testen auf inf und nan: Ausprobieren

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

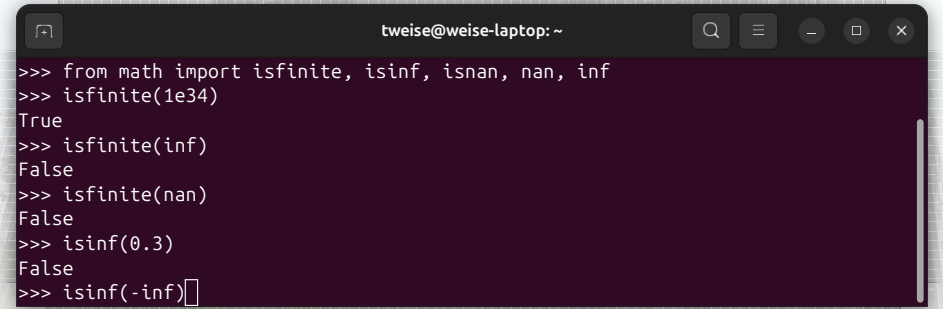
Testen auf inf und nan: Ausprobieren

- `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)` ...

tweise@weise-laptop: ~

```
False
>>> isfinite(nan)
False
>>> isinf(0.3)
False
>>> isinf(-inf)
True
>>> isinf(nan)
False
>>> 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` fpr 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` fpr 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

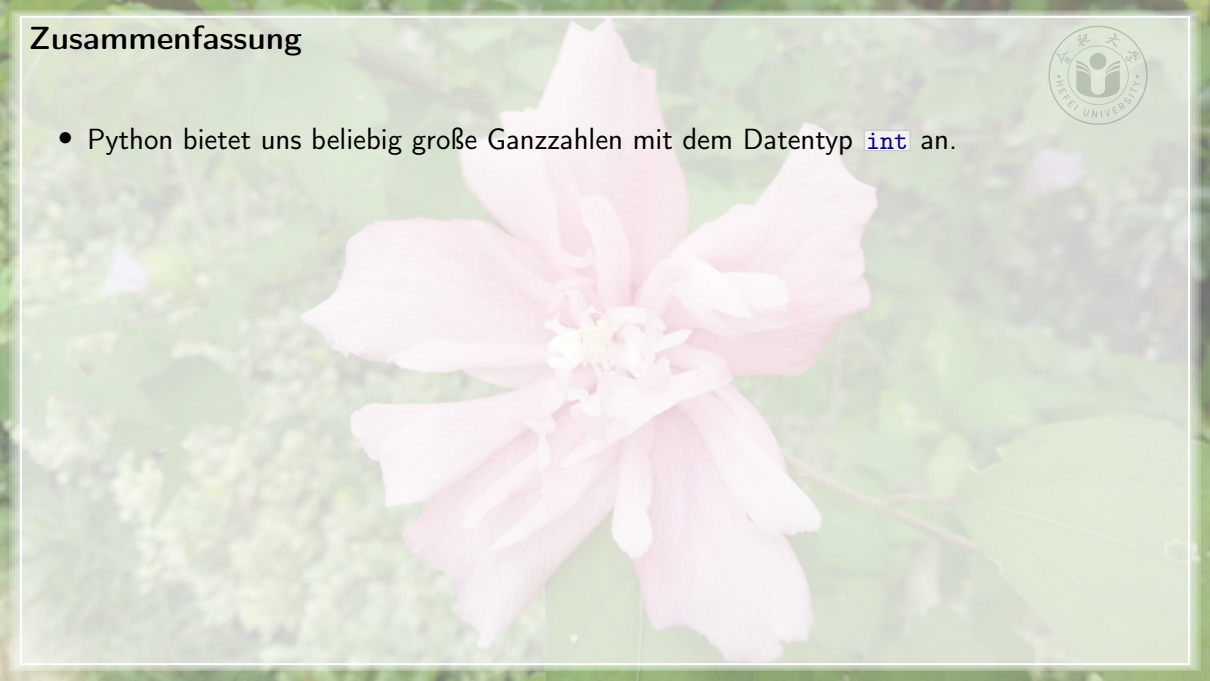
- Es ist wichtig, dass wir so eigenartige Werte wie `inf`, `-inf` und `nan` abfangen können.
- Manchmal würden wir das bereits machen, wenn wir Eingabewerte einlesen.
- Manchmal machen wir das während unserer Berechnungen.
- Was wir in der Regel nicht wollen, ist Kode der solche Werte ausgibt.
- Ein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

Zusammenfassung

- Python bietet uns beliebig große Ganzzahlen mit dem Datentyp `int` an.
- Für reelle Zahlen können wir den Datentype `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 Datentype `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 Datentype `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 Datentype `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 Datentype `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] "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. 118, 119).

[2] 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. 104–109).

[3] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: 978-0-321-35668-0 (siehe S. 415).

[4] 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. 186–226).

[5] "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. 115, 116).

[6] "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. 228–235, 249, 250, 266–269).

[7] 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. 363–373).

[8] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 415).

[9] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (besucht am 2024-09-24). Translation of[10]. (Siehe S. 412).

References II

[10] 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[9] for a translation. (Siehe S. 411, 415).

[11] 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, 415).

[12] "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. 38–46, 48–80).

[13] 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. 415).

[14] 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, 174–181).

[15] ."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) (siehe S. 117).

[16] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 415).

[17] *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. 16–37, 118, 119, 174–181, 228–235).

[18] *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. 111–172).

References III

[19] 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, 415).

[20] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 415).

[21] 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. 415).

[22] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 415).

[23] 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) (siehe S. 117).

[24] 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, 415).

[25] "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).

[26] 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 & Assessment, 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. 104–109).

[27] *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. 415).

References IV

[28] *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).

[29] 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 und Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 415).

Glossary (in English) I

C is a programming language, which is very successful in system programming situations[8,27].

Java is another very successful programming language, with roots in the C family of languages[3,21].

Python The Python programming language[16,20,22,29], i.e., what you will learn about in our book[29]. Learn more at `https://python.org`.

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[13].

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[11,19,24], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. In Python, it is provided by the `math` module as constant `pi` with value `3.141592653589793`.

e is Euler's number[10], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number[11,19], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as constant `e` with value `2.718281828459045`.

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