



Programming with Python

42. Klassen: Kapselung

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

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

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Programming with Python



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

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



Outline



1. Einleitung
2. Beispiel
3. Zusammenfassung





Einleitung



Einleitung

- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.



Einleitung

- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.





Einleitung

- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.



Einleitung

- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*

Einleitung



- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.

Einleitung



- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.

Einleitung



- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.
- Wenn ein Objekt mehrere Attribute hat, dann stehen diese Attribute wahrscheinlich miteinander in einer Beziehung und ihre Werte hängen irgendwie zusammen.

Einleitung



- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.
- Wenn ein Objekt mehrere Attribute hat, dann stehen diese Attribute wahrscheinlich miteinander in einer Beziehung und ihre Werte hängen irgendwie zusammen.
- Wir wollen dann nicht, dass irgendjemand diese Werte beliebig verändern kann.

Einleitung



- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.
- Wenn ein Objekt mehrere Attribute hat, dann stehen diese Attribute wahrscheinlich miteinander in einer Beziehung und ihre Werte hängen irgendwie zusammen.
- Wir wollen dann nicht, dass irgendjemand diese Werte beliebig verändern kann.
- Oftmals wollen wir veränderliche Objekte so entwickeln, dass auf die Werte ihrer veränderlichen Attribute nur über Methoden zugegriffen werden kann.

Einleitung



- Oftmals wollen wir, dass unsere Objekte unveränderlich sind.
- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.
- Wenn ein Objekt mehrere Attribute hat, dann stehen diese Attribute wahrscheinlich miteinander in einer Beziehung und ihre Werte hängen irgendwie zusammen.
- Wir wollen dann nicht, dass irgendjemand diese Werte beliebig verändern kann.
- Oftmals wollen wir veränderliche Objekte so entwickeln, dass auf die Werte ihrer veränderlichen Attribute nur über Methoden zugegriffen werden kann.
- Die Methoden und Attribute einer Klasse formen ja auch eine semantische Einheit.



Einleitung

- Das geht natürlich nicht immer.
- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.
- Wenn ein Objekt mehrere Attribute hat, dann stehen diese Attribute wahrscheinlich miteinander in einer Beziehung und ihre Werte hängen irgendwie zusammen.
- Wir wollen dann nicht, dass irgendjemand diese Werte beliebig verändern kann.
- Oftmals wollen wir veränderliche Objekte so entwickeln, dass auf die Werte ihrer veränderlichen Attribute nur über Methoden zugegriffen werden kann.
- Die Methoden und Attribute einer Klasse formen ja auch eine semantische Einheit.
- Die Methoden werden von den selben Programmierern geschrieben, die die Attribute designed haben.

Einleitung



- Es gibt auch viele Situationen, wo wir wollen, dass sich die Attribute eines Objekts ändern.
- Die Frage ist dann *Wie sollte man den Zustand eines Objekts ändern?*
- Die große Mehrheit unserer Klassen werden mehr als ein Attribut haben.
- Alle Attribute eines Objekts zusammen formen den Zustand des Objekts und bilden eine semantische Einheit.
- Wenn ein Objekt mehrere Attribute hat, dann stehen diese Attribute wahrscheinlich miteinander in einer Beziehung und ihre Werte hängen irgendwie zusammen.
- Wir wollen dann nicht, dass irgendjemand diese Werte beliebig verändern kann.
- Oftmals wollen wir veränderliche Objekte so entwickeln, dass auf die Werte ihrer veränderlichen Attribute nur über Methoden zugegriffen werden kann.
- Die Methoden und Attribute einer Klasse formen ja auch eine semantische Einheit.
- Die Methoden werden von den selben Programmierern geschrieben, die die Attribute designed haben.
- Diese Programmierer wissen genau, wie die Attribute in einer vernünftigen und konsistenten Art verändert werden können.



Beispiel



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.





Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiert.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiertz. B.:
 1. die Methode von LIU Hui (刘徽) zum annähern von π in Einheit 13.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiertz. B.:
 1. die Methode von LIU Hui (刘徽) zum annähern von π in Einheit 13,
 2. Heron's Methode zum annähern der Quadratwurzel in Einheit 25.



Einleitung zum Beispiel

- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiertz. B.:
 1. die Methode von LIU Hui (刘徽) zum annähern von π in Einheit 13,
 2. Heron's Methode zum annähern der Quadratwurzel in Einheit 25 und
 3. Euclids Algorithmus zum berechnen des größten gemeinsamen Teilers in Einheit 26.

Einleitung zum Beispiel



- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiertz. B.:
 1. die Methode von LIU Hui (刘徽) zum annähern von π in Einheit 13,
 2. Heron's Methode zum annähern der Quadratwurzel in Einheit 25 und
 3. Euclids Algorithmus zum berechnen des größten gemeinsamen Teilers in Einheit 26.
- Diese Algorithmen kennen Sie wahrscheinlich aus der Schule.

Einleitung zum Beispiel



- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiertz. B.:
 1. die Methode von LIU Hui (刘徽) zum annähern von π in Einheit 13,
 2. Heron's Methode zum annähern der Quadratwurzel in Einheit 25 und
 3. Euclids Algorithmus zum berechnen des größten gemeinsamen Teilers in Einheit 26.
- Diese Algorithmen kennen Sie wahrscheinlich aus der Schule.
- Implementieren wir nun einen Algorithmus, von dem Sie wahrscheinlich noch *nie* gehört haben.

Einleitung zum Beispiel



- Wir schauen uns nun ein Szenario an.
- Wir haben eine Klasse deren Attribute eng zusammenhängen, so dass es gar keinen Sinn ergeben würde, zu erlauben, dass ein Benutzer sie selbst verändern könnte.
- Stattdessen implementieren wir Methoden die die Attribute auf eine konsistente Art ändern und Information vom Objekte abrufen können.
- Natürlich nehmen wir als Beispiel wieder einen mathematischen Algorithmus.
- Wir haben ja schon einige interessante mathematische Algorithmen implementiertz. B.:
 1. die Methode von LIU Hui (刘徽) zum annähern von π in Einheit 13,
 2. Heron's Methode zum annähern der Quadratwurzel in Einheit 25 und
 3. Euclids Algorithmus zum berechnen des größten gemeinsamen Teilers in Einheit 26.
- Diese Algorithmen kennen Sie wahrscheinlich aus der Schule.
- Implementieren wir nun einen Algorithmus, von dem Sie wahrscheinlich noch *nie* gehört haben.
- Eine praktische Methode, um die Beschränkungen des Datentyps `float` zu umgehen, die wir vor langer Zeit in Einheit 8 diskutiert haben.

Grenzen des Datentyps float

- Wir wissen, dass der Datentyp **float**

Zahlen auf 15 bis 16 Ziffern genau darstellen kann.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wir wissen, dass der Datentyp `float`

Zahlen auf 15 bis 16 Ziffern genau darstellen kann.

- Wenn wir `1` zu $10^{15} = 1\text{e}15$ addieren, dann ist das korrekte Ergebnis `1000000000000001.0`.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 1000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wir wissen, dass der Datentyp `float`

Zahlen auf 15 bis 16 Ziffern genau darstellen kann.

- Wenn wir `1` zu $10^{15} = 1\text{e}15$ addieren, dann ist das korrekte Ergebnis `1000000000000001.0`.

- Wie Sie selbst zählen können, finden wir 16 Ziffern vor dem Dezimalpunkt, zwei `1`en und 14 `0`en.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 1000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wir wissen, dass der Datentyp `float` Zahlen auf 15 bis 16 Ziffern genau darstellen kann.
- Wenn wir `1` zu $10^{15} = 1\text{e}15$ addieren, dann ist das korrekte Ergebnis `1000000000000001.0`.
- Wie Sie selbst zählen können, finden wir 16 Ziffern vor dem Dezimalpunkt, zwei `1`en und 14 `0`en.
- Daher würde die Addition von `1` zu $10^{16} = 1\text{e}16$ 17 Ziffern erfordern.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 1000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wir wissen, dass der Datentyp `float` Zahlen auf 15 bis 16 Ziffern genau darstellen kann.
- Wenn wir `1` zu $10^{15} = 1\text{e}15$ addieren, dann ist das korrekte Ergebnis `1000000000000001.0`.
- Wie Sie selbst zählen können, finden wir 16 Ziffern vor dem Dezimalpunkt, zwei `1`en und 14 `0`en.
- Daher würde die Addition von `1` zu $10^{16} = 1\text{e}16$ 17 Ziffern erfordern.
- Es überschreitet die Kapazität des Datentyps `floats`.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 1000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wenn wir 1 zu $10^{15} = 1\text{e}15$ addieren, dann ist das korrekte Ergebnis 1000000000000001.0.
- Wie Sie selbst zählen können, finden wir 16 Ziffern vor dem Dezimalpunkt, zwei 1en und 14 0en.
- Daher würde die Addition von 1 zu $10^{16} = 1\text{e}16$ 17 Ziffern erfordern.
- Es überschreitet die Kapazität des Datentyps floats.
- Daher geht die niedrigstwertige Ziffer „verloren“.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 1000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wie Sie selbst zählen können, finden wir 16 Ziffern vor dem Dezimalpunkt, zwei 1en und 14 0en.
- Daher würde die Addition von 1 zu $10^{16} = 1e16$ 17 Ziffern erfordern.
- Es überschreitet die Kapazität des Datentyps `floats`.
- Daher geht die niedrigstwertige Ziffer „verloren“.
- Das Ergebnis von `1e16 + 1`, berechnet mit `floats`, ist immer noch `1e16`.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Daher würde die Addition von `1` zu $10^{16} = 1\text{e}16$ 17 Ziffern erfordern.
- Es überschreitet die Kapazität des Datentyps `floats`.
- Daher geht die niedrigstwertige Ziffer „verloren“.
- Das Ergebnis von `1e16 + 1`, berechnet mit `floats`, ist immer noch `1e16`.
- Es ist nicht möglich, die Zahl $1 + 10^{16}$ exakt mit den 64 Bit Double Precision Floating Point Numbers^{28,37,39} die Python bietet, darzustellen.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Es überschreitet die Kapazität des Datentyps `floats`.
- Daher geht die niedrigstwertige Ziffer „verloren“.
- Das Ergebnis von `1e16 + 1`, berechnet mit `floats`, ist immer noch `1e16`.
- Es ist nicht möglich, die Zahl $1 + 10^{16}$ exakt mit den 64 Bit Double Precision Floating Point Numbers^{28,37,39} die Python bietet, darzustellen.
- Normalerweise ist das auch OK.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Daher geht die niedrigstwertige Ziffer „verloren“.
- Das Ergebnis von `1e16 + 1`, berechnet mit `floats`, ist immer noch `1e16`.
- Es ist nicht möglich, die Zahl $1 + 10^{16}$ exakt mit den 64 Bit Double Precision Floating Point Numbers^{28,37,39} die Python bietet, darzustellen.
- Normalerweise ist das auch OK.
- Es gibt sehr wenige Anwendungen, bei denen wir wirklich mehr als 15 Ziffern Genauigkeit brauchen.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Das Ergebnis von `1e16 + 1`, berechnet mit `floats`, ist immer noch `1e16`.
- Es ist nicht möglich, die Zahl $1 + 10^{16}$ exakt mit den 64 Bit Double Precision Floating Point Numbers^{28,37,39} die Python bietet, darzustellen.
- Normalerweise ist das auch OK.
- Es gibt sehr wenige Anwendungen, bei denen wir wirklich mehr als 15 Ziffern Genauigkeit brauchen.
- Machen wir mit dem Beispiel weiter.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Es ist nicht möglich, die Zahl $1 + 10^{16}$ exakt mit den 64 Bit Double Precision Floating Point Numbers^{28,37,39} die Python bietet, darzustellen.
- Normalerweise ist das auch OK.
- Es gibt sehr wenige Anwendungen, bei denen wir wirklich mehr als 15 Ziffern Genauigkeit brauchen.
- Machen wir mit dem Beispiel weiter.
- Was passiert, wenn wir $10^{16} + 1$ berechnen, und dann 10^{16} von der Summe abziehen?

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Normalerweise ist das auch OK.
- Es gibt sehr wenige Anwendungen, bei denen wir wirklich mehr als 15 Ziffern Genauigkeit brauchen.
- Machen wir mit dem Beispiel weiter.
- Was passiert, wenn wir $10^{16} + 1$ berechnen, und dann 10^{16} von der Summe abziehen?
- Offensichtlich, in einer idealen Welt, wäre das Ergebnis 1.0.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Es gibt sehr wenige Anwendungen, bei denen wir wirklich mehr als 15 Ziffern Genauigkeit brauchen.
- Machen wir mit dem Beispiel weiter.
- Was passiert, wenn wir $10^{16} + 1$ berechnen, und dann 10^{16} von der Summe abziehen?
- Offensichtlich, in einer idealen Welt, wäre das Ergebnis 1.0.
- Nun wäre `1e16 + 1` ja `10_000_000_000_001.0`, was nicht in einen `float` passt.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Machen wir mit dem Beispiel weiter.
- Was passiert, wenn wir $10^{16} + 1$ berechnen, und dann 10^{16} von der Summe abziehen?
- Offensichtlich, in einer idealen Welt, wäre das Ergebnis 1.0.
- Nun wäre `1e16 + 1` ja `10_000_000_000_001.0`, was nicht in einen `float` passt.
- 1.0 ist aber eine Zahl, die wir sehr wohl genau als `float` darstellen können.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Was passiert, wenn wir $10^{16} + 1$ berechnen, und dann 10^{16} von der Summe abziehen?
- Offensichtlich, in einer idealen Welt, wäre das Ergebnis `1.0`.
- Nun wäre `1e16 + 1` ja `10_000_000_000_001.0`, was nicht in einen `float` passt.
- `1.0` ist aber eine Zahl, die wir sehr wohl genau als `float` darstellen können.
- Das wirkliche Ergebnis der Berechnung in Python ist jedoch `0.0`.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Offensichtlich, in einer idealen Welt, wäre das Ergebnis `1.0`.
- Nun wäre `1e16 + 1` ja `10_000_000_000_000_001.0`, was nicht in einen `float` passt.
- `1.0` ist aber eine Zahl, die wir sehr wohl genau als `float` darstellen können.
- Das wirkliche Ergebnis der Berechnung in Python ist jedoch `0.0`.
- Der Grund ist, dass das Zwischenergebnis `1e16 + 1 == 1e16` und dann `1e16 - 1e16 == 0` erfolgt.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- `1.0` ist aber eine Zahl, die wir sehr wohl genau als `float` darstellen können.
- Das wirkliche Ergebnis der Berechnung in Python ist jedoch `0.0`.
- Der Grund ist, dass das Zwischenergebnis `1e16 + 1 == 1e16` und dann `1e16 - 1e16 == 0` erfolgt.
- Ähnliches passiert, wenn wir `1e18 + 1 + 1e36 - 1e36 - 1e18` berechnen, was dann `-1e18` ergibt, wohingegen das „korrekte“ Ergebnis wieder `1.0` wäre.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- `1.0` ist aber eine Zahl, die wir sehr wohl genau als `float` darstellen können.
- Das wirkliche Ergebnis der Berechnung in Python ist jedoch `0.0`.
- Der Grund ist, dass das Zwischenergebnis `1e16 + 1 == 1e16` und dann `1e16 - 1e16 == 0` erfolgt.
- Ähnliches passiert, wenn wir `1e18 + 1 + 1e36 - 1e36 - 1e18` berechnen, was dann `-1e18` ergibt, wohingegen das „korrekte“ Ergebnis wieder `1.0` wäre.
- Der Grund ist dass zuerst `1e18 + 1 == 1e18` berechnet wird.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Das wirkliche Ergebnis der Berechnung in Python ist jedoch 0.0.

- Der Grund ist, dass das Zwischenergebnis

$1e16 + 1 == 1e16$ und dann
 $1e16 - 1e16 == 0$ erfolgt.

- Ähnliches passiert, wenn wir

$1e18 + 1 + 1e36 - 1e36 - 1e18$

berechnen, was dann $-1e18$ ergibt, wohingegen das „korrekte“ Ergebnis wieder 1.0 wäre.

- Der Grund ist dass zuerst

$1e18 + 1 == 1e18$ berechnet wird.

- Danach ergibt $1e18 + 1e36$ dann $1e36$, wovon wir wiederum $1e36$ abziehen und 0.0 bekommen.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16              # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36          # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Der Grund ist, dass das Zwischenergebnis

`1e16 + 1 == 1e16` und dann

`1e16 - 1e16 == 0` erfolgt.

- Ähnliches passiert, wenn wir

`1e18 + 1 + 1e36 - 1e36 - 1e18`

berechnen, was dann `-1e18` ergibt, wohingegen das „korrekte“ Ergebnis wieder `1.0` wäre.

- Der Grund ist dass zuerst

`1e18 + 1 == 1e18` berechnet wird.

- Danach ergibt `1e18 + 1e36` dann

`1e36`, wovon wir wiederum `1e36` abziehen und `0.0` bekommen.

- Die letzte Subtraktion von `1e18` liefert dann `1e-18`.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Ähnliches passiert, wenn wir

`1e18 + 1 + 1e36 - 1e36 - 1e18`

berechnen, was dann `-1e18` ergibt, wohingegen das „korrekte“ Ergebnis wieder `1.0` wäre.

- Der Grund ist dass zuerst

`1e18 + 1 == 1e18` berechnet wird.

- Danach ergibt `1e18 + 1e36` dann

`1e36`, wovon wir wiederum `1e36` abziehen und `0.0` bekommen.

- Die letzte Subtraktion von `1e18` liefert dann `1e-18`.

- Hätten wir unendliche Genauigkeit, dann würde der erste Rechenschritt $10^{18} + 1 = 1\ 000\ 000\ 000\ 000\ 000\ 001$ ergeben.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Der Grund ist dass zuerst $1e18 + 1 == 1e18$ berechnet wird.
- Danach ergibt $1e18 + 1e36$ dann $1e36$, wovon wir wiederum $1e36$ abziehen und 0.0 bekommen.
- Die letzte Subtraktion von $1e18$ liefert dann $1e-18$.
- Hätten wir unendliche Genauigkeit, dann würde der erste Rechenschritt $10^{18} + 1 = 1\ 000\ 000\ 000\ 000\ 000\ 001$ ergeben.
- Wenn wir $1e36$ zu dieser Zahl addieren, dann sollte das Ergebnis $1000\ 000\ 000\ 000\ 000\ 001\ 000\ 000\ 000\ 000\ 001$ sein.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Die letzte Subtraktion von `1e18` liefert dann `1e-18`.
- Hätten wir unendliche Genauigkeit, dann würde der erste Rechenschritt $10^{18} + 1 = 1\ 000\ 000\ 000\ 000\ 000\ 001$ ergeben.
- Wenn wir `1e36` zu dieser Zahl addieren, dann sollte das Ergebnis $1000\ 000\ 000\ 000\ 000\ 001\ 000\ 000\ 000\ 000\ 001$ sein.
- Das Abziehen von `1e36` von dieser gigantischen Zahl würde uns wieder zu $10^{18} + 1$ bringen und die letzte Subtraktion von `1e18` ergäbe dann `1.0`.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Hätten wir unendliche Genauigkeit, dann würde der erste Rechenschritt $10^{18} + 1 = 1\ 000\ 000\ 000\ 000\ 000\ 001$ ergeben.
- Wenn wir `1e36` zu dieser Zahl addieren, dann sollte das Ergebnis $1000\ 000\ 000\ 000\ 000\ 001\ 000\ 000\ 000\ 000\ 001$ sein.
- Das Abziehen von `1e36` von dieser gigantischen Zahl würde uns wieder zu $10^{18} + 1$ bringen und die letzte Subtraktion von `1e18` ergäbe dann `1.0`.
- Es ist leicht zu sehen, warum das nicht funktioniert.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Wenn wir `1e36` zu dieser Zahl addieren, dann sollte das Ergebnis `1000 000 000 000 000 001 000 000 000 000 001` sein.
- Das Abziehen von `1e36` von dieser gigantischen Zahl würde uns wieder zu $10^{18} + 1$ bringen und die letzte Subtraktion von `1e18` ergäbe dann `1.0`.
- Es ist leicht zu sehen, warum das nicht funktioniert.
- Die Entwickler der Fließkommaarithmetikeinheit von CPUs mussten eine bestimmte, begrenzte Anzahl von Bits für den Datentyp `float` festlegen.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Es ist leicht zu sehen, warum das nicht funktioniert.
- Die Entwickler der Fließkommaarithmetikeinheit von CPUs mussten eine bestimmte, begrenzte Anzahl von Bits für den Datentyp `float` festlegen.
- Sie haben sich überlegt, dass 52 Bits für den Signifikand, die uns $52/\log_2 10 \approx 15.7$ Ziffern geben, vernünftig sind und auch gut in acht Bytes Speicher passen.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Die Entwickler der Fließkommaarithmetikeinheit von CPUs mussten eine bestimmte, begrenzte Anzahl von Bits für den Datentyp `float` festlegen.
- Sie haben sich überlegt, dass 52 Bits für den Signifikand, die uns $52 / \log_2 10 \approx 15.7$ Ziffern geben, vernünftig sind und auch gut in acht Bytes Speicher passen.
- Dagegen benötigen 36 Ziffern einen Signifikand von $36 \log_2 10 \approx 120$ Bits, was den Datentyp viel größer macht und wahrscheinlich auch nur sehr selten benötigt werden würde.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Sie haben sich überlegt, dass 52 Bits für den Signifikand, die uns $52 / \log_2 10 \approx 15.7$ Ziffern geben, vernünftig sind und auch gut in acht Bytes Speicher passen.
- Dagegen benötigen 36 Ziffern einen Signifikand von $36 \log_2 10 \approx 120$ Bits, was den Datentyp viel größer macht und wahrscheinlich auch nur sehr selten benötigt werden würde.
- Naja, es sei denn, man addiert große und kleine Zahlen zusammen...

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Sie haben sich überlegt, dass 52 Bits für den Signifikand, die uns $52 / \log_2 10 \approx 15.7$ Ziffern geben, vernünftig sind und auch gut in acht Bytes Speicher passen.
- Dagegen benötigen 36 Ziffern einen Signifikand von $36 \log_2 10 \approx 120$ Bits, was den Datentyp viel größer macht und wahrscheinlich auch nur sehr selten benötigt werden würde.
- Naja, es sei denn, man addiert große und kleine Zahlen zusammen...
- Die interessante Frage, die wir basierend auf dem Beispiel stellen können ist...

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Dagegen benötigen 36 Ziffern einen Signifikand von $36 \log_2 10 \approx 120$ Bits, was den Datentyp viel größer macht und wahrscheinlich auch nur sehr selten benötigt werden würde.
- Naja, es sei denn, man addiert große und kleine Zahlen zusammen...
- Die interessante Frage, die wir basierend auf dem Beispiel stellen können ist:
Gibt es eine Möglichkeit, große und kleine Zahlen genauer zu addieren?

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Dagegen benötigen 36 Ziffern einen Signifikand von $36 \log_2 10 \approx 120$ Bits, was den Datentyp viel größer macht und wahrscheinlich auch nur sehr selten benötigt werden würde.
- Naja, es sei denn, man addiert große und kleine Zahlen zusammen...
- Die interessante Frage, die wir basierend auf dem Beispiel stellen können ist:
- *Gibt es eine Möglichkeit, große und kleine Zahlen genauer zu addieren?*
- Natürlich können wir `1e16 + 1` niemals genau mit einem `float` repräsentieren.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Naja, es sei denn, man addiert große und kleine Zahlen zusammen...
- Die interessante Frage, die wir basierend auf dem Beispiel stellen können ist:
- Gibt es eine Möglichkeit, große und kleine Zahlen genauer zu addieren?
- Natürlich können wir `1e16 + 1` niemals genau mit einem `float` repräsentieren.
- Trotzdem wollen wir das Endergebnis eine Summe doch so genau wie möglich darstellen.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Die interessante Frage, die wir basierend auf dem Beispiel stellen können ist:
- Gibt es eine Möglichkeit, große und kleine Zahlen genauer zu addieren?
- Natürlich können wir `1e16 + 1` niemals genau mit einem `float` repräsentieren.
- Trotzdem wollen wir das Endergebnis eine Summe doch so genau wie möglich darstellen.
- Wir wollen eine Möglichkeit, die Summe `(1e18 + 1) + -1e18` so zu berechnen, dass am Ende `1.0` herauskommt.

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16               # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36                 # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36         # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Grenzen des Datentyps float

- Gibt es eine Möglichkeit, große und kleine Zahlen genauer zu addieren?
- Natürlich können wir `1e16 + 1` niemals genau mit einem `float` repräsentieren.
- Trotzdem wollen wir das Endergebnis einer Summe doch so genau wie möglich darstellen.
- Wir wollen eine Möglichkeit, die Summe `(1e18 + 1) + -1e18` so zu berechnen, dass am Ende `1.0` herauskommt.
- Geht das?

```
1 Python 3.12.12 (main, Oct 11 2025, 01:16:26) [GCC 15.2.0] on linux
2 Type "help", "copyright", "credits" or "license" for more information.
3 # Python's float supports 15 to 16 digits of precision.
4 >>> 1e15 + 1                      # This works and will give us 1_000_000_000_000_001
5 10000000000000001.0
6 >>> 1e16 + 1                      # This gives us 1e16, since we would need 17 digits
7 1e+16
8
9 # While 1e16 + 1 cannot be represented exactly with a `float`, one
10 # may wonder what happens if the computation produces results that
11 # can be represented exactly ... but has intermediate steps that
12 # would exceed the range that can be covered by a `float`.
13 >>> 1e16 + 1 - 1e16 # Since 1e16 + 1 == 1e16, this does yield 0.
14 0.0
15
16 # The problems with the range of `float` can also occur if we have
17 # two very large numbers ... as long as one is much larger than the
18 # other.
19 >>> 1e18 + 1e36      # 1_000_000_000_000_001 * 1e18 needs 19 digits
20 1e+36
21
22 # We here can again observe the problem with intermediate results.
23 # The final result may fit well into a `float`, but if precision is
24 # lost in intermediate steps, the final result can be off quite a
25 # bit.
26 >>> 1e18 + 1e36 - 1e36 # Since 1e18 + 1e36 == 1e36, this gives us 0.
27 0.0
28 >>> 1e18 + 1 + 1e36 - 1e36 - 1e18 # The exact result would be 1, but ...
29 -1e+18
```

Kahan-Summe: Sinn und Zweck



- Zum Glück hatten Kahan⁴² und Babuška⁴ in den 1960ern eine Idee, wie man die Präzision für Addition erhöhen kann^{29,47}.

Kahan-Summe: Sinn und Zweck



- Zum Glück hatten Kahan⁴² und Babuška⁴ in den 1960ern eine Idee, wie man die Präzision für Addition erhöhen kann^{29,47}:
- Wir addieren Zahlen in einer Variable `sum` und verwenden zusätzlich eine Variable `cs`, in der wir uns merken, wie weit das Ergebnis daneben liegt.

Kahan-Summe: Sinn und Zweck



- Zum Glück hatten Kahan⁴² und Babuška⁴ in den 1960ern eine Idee, wie man die Präzision für Addition erhöhen kann^{29,47}:
- Wir addieren Zahlen in einer Variable `sum` und verwenden zusätzlich eine Variable `cs`, in der wir uns merken, wie weit das Ergebnis daneben liegt.
- Stellen wir uns nochmal vor, dass wir `1e18 + 1 - 1e18` berechnen wollen.

Kahan-Summe: Sinn und Zweck



- Zum Glück hatten Kahan⁴² und Babuška⁴ in den 1960ern eine Idee, wie man die Präzision für Addition erhöhen kann^{29,47}:
- Wir addieren Zahlen in einer Variable `sum` und verwenden zusätzlich eine Variable `cs`, in der wir uns merken, wie weit das Ergebnis daneben liegt.
- Stellen wir uns nochmal vor, dass wir `1e18 + 1 - 1e18` berechnen wollen.
- Wie würden wir bei dem Ergebnis `1.0` ankommen?

Kahan-Summe: Sinn und Zweck



- Zum Glück hatten Kahan⁴² und Babuška⁴ in den 1960ern eine Idee, wie man die Präzision für Addition erhöhen kann^{29,47}:
- Wir addieren Zahlen in einer Variable `sum` und verwenden zusätzlich eine Variable `cs`, in der wir uns merken, wie weit das Ergebnis daneben liegt.
- Stellen wir uns nochmal vor, dass wir `1e18 + 1 - 1e18` berechnen wollen.
- Wie würden wir bei dem Ergebnis `1.0` ankommen?
- In dem wir den folgenden, einfachen Algorithmus anwenden, der die Summe aller Zahlen in einer Variable `sum` bildet und zusätzlich einen Fehler-Term `cs` mitführt.

Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
1. Wir beginnen mit `sum = 0` und `cs = 0`.





Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
`t` ist damit die Summe, und zwar zu der Genauigkeit, die `float` darstellen kann.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t)+ value`. Weil
`t = sum + value`, könnte man denken
das `error` eigentlich immer `0.0` seien
sollte, schließlich sieht das aus wie
`(sum - (sum + value))+ value`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t)+ value`. Weil
`t = sum + value`, könnte man denken
das `error` eigentlich immer `0.0` seien
sollte, schließlich sieht das aus wie
`(sum - (sum + value))+ value`.
Aber es können ja ein paar Ziffern
„verloren gehen“, wenn wir `sum + value`
berechnen.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t)+ value`. Weil
`t = sum + value`, könnte man denken
das `error` eigentlich immer `0.0` seien
sollte, schließlich sieht das aus wie
`(sum - (sum + value))+ value`.
Aber es können ja ein paar Ziffern
„verloren gehen“, wenn wir `sum + value`
berechnen. Diese tauchen dann wieder in
`error` auf.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.

1. Wir beginnen mit `sum = 0` und `cs = 0`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
- 1. Wir beginnen mit `sum = 0` und `cs = 0`.
- 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
- 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 - 2.1 In ersten Schritt berechnen wir `t = sum + value`, also `t = 0 + 1e18`, also `t = 1e18`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 - 2.1 In ersten Schritt berechnen wir `t = sum + value`, also `t = 0 + 1e18`, also `t = 1e18`.
 - 2.2 Dann setzen wir `error = (sum - t) + value`, woraus dann `error = (0 - 1e18) + 1e18` wird, was uns `error = 0.0` gibt.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 - 2.1 In ersten Schritt berechnen wir `t = sum + value`, also `t = 0 + 1e18`, also `t = 1e18`.
 - 2.2 Dann setzen wir `error = (sum - t) + value`, woraus dann `error = (0 - 1e18) + 1e18` wird, was uns `error = 0.0` gibt.
 - 2.3 Wir setzen also `sum = 1e18`, also `sum = 1e18`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 - 2.1 In ersten Schritt berechnen wir `t = sum + value`, also `t = 0 + 1e18`, also `t = 1e18`.
 - 2.2 Dann setzen wir `error = (sum - t) + value`, woraus dann `error = (0 - 1e18) + 1e18` wird, was uns `error = 0.0` gibt.
 - 2.3 Wir setzen also `sum = 1e18`, also `sum = 1e18`.
 - 2.4 `cs`, was `0` war, wird `0 + 0.0`, und ist somit `cs = 0.0`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 - 3.1 Wir berechnen `t = sum + value`, was nun `t = 1e18 + 1` ist, was uns dann `t = 1e18` gibt.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 - 3.1 Wir berechnen `t = sum + value`, was nun `t = 1e18 + 1` ist, was uns dann `t = 1e18` gibt. **Die 1 geht verloren.**



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 - 3.1 Wir berechnen `t = sum + value`, was nun `t = 1e18 + 1` ist, was uns dann `t = 1e18` gibt. **Die 1 geht verloren.**
 - 3.2 Also fast zumindest.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 - 3.1 Wir berechnen `t = sum + value`, was nun `t = 1e18 + 1` ist, was uns dann `t = 1e18` gibt. **Die 1 geht verloren.**
 - 3.2 Also fast zumindest:
`error = (sum - t) + value` wird zu
`error = (1e18 - 1e18) + 1`, also ist `error = 1.0`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 - 3.1 Wir berechnen `t = sum + value`, was nun `t = 1e18 + 1` ist, was uns dann `t = 1e18` gibt. **Die 1 geht verloren.**
 - 3.2 Also fast zumindest:
`error = (sum - t) + value` wird zu
`error = (1e18 - 1e18) + 1`, also ist `error = 1.0`.
 - 3.3 Nach diesem Schritt ist `sum = 1e18`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 - 3.1 Wir berechnen `t = sum + value`, was nun `t = 1e18 + 1` ist, was uns dann `t = 1e18` gibt. **Die 1 geht verloren.**
 - 3.2 Also fast zumindest:
`error = (sum - t) + value` wird zu
`error = (1e18 - 1e18) + 1`, also ist `error = 1.0`.
 - 3.3 Nach diesem Schritt ist `sum = 1e18`.
 - 3.4 Und `cs = 0 + 1.0` wird `1.0`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.

1. Wir beginnen mit `sum = 0` und `cs = 0`.
2. Als erste Zahl addieren wir `1e18` zur Summe.
3. Jetzt addieren wir `1` zu der Summe.
4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.



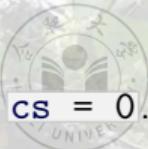
Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
 - Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
 - 4.1 Das bedeutet, dass wir zuerst `t = sum + value` berechnen, also `t = 1e18 + -1e18`, woraus dann `t = 0.0` wird.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
 - 4.1 Das bedeutet, dass wir zuerst `t = sum + value` berechnen, also `t = 1e18 + -1e18`, woraus dann `t = 0.0` wird.
 - 4.2 Für den Fehlerterm bekommen wir `error = (1e18 - 0.0) + -1e18`, was `error = 0.0` ergibt.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.

1. Wir beginnen mit `sum = 0` und `cs = 0`.
2. Als erste Zahl addieren wir `1e18` zur Summe.
3. Jetzt addieren wir `1` zu der Summe.
4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
 - 4.1 Das bedeutet, dass wir zuerst `t = sum + value` berechnen, also `t = 1e18 + -1e18`, woraus dann `t = 0.0` wird.
 - 4.2 Für den Fehlerterm bekommen wir `error = (1e18 - 0.0) + -1e18`, was `error = 0.0` ergibt.
 - 4.3 Wir bekommen nun `sum = 0.0`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.

1. Wir beginnen mit `sum = 0` und `cs = 0`.
2. Als erste Zahl addieren wir `1e18` zur Summe.
3. Jetzt addieren wir `1` zu der Summe.
4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
 - 4.1 Das bedeutet, dass wir zuerst `t = sum + value` berechnen, also `t = 1e18 + -1e18`, woraus dann `t = 0.0` wird.
 - 4.2 Für den Fehlerterm bekommen wir `error = (1e18 - 0.0) + -1e18`, was `error = 0.0` ergibt.
 - 4.3 Wir bekommen nun `sum = 0.0`.
 - 4.4 Und `cs = 1.0 + 0.0`, bleibt also bei `cs = 1.0`.



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel
`1e18 + 1 - 1e18` aus.

1. Wir beginnen mit `sum = 0` und `cs = 0`.
2. Als erste Zahl addieren wir `1e18` zur Summe.
3. Jetzt addieren wir `1` zu der Summe.
4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
5. Das Endergebnis ist dann `sum + cs` und das ergibt tatsächlich `0.0 + 1.0`, also `1.0!`



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
 5. Das Endergebnis ist dann `sum + cs` und das ergibt tatsächlich `0.0 + 1.0`, also `1.0`!



Kahan-Summe: Ausprobieren

- In dem wir den folgenden, einfachen Algorithmus anwenden:
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Für jede Zahl, die zur Summe dazuaddiert werden soll, machen wir die folgende Schritte:
 - 2.1 Wir berechnen zuerst `t = sum + value`.
 - 2.2 Dann berechnen wir
`error = (sum - t) + value`.
 - 2.3 Dann setzen wir `sum = t`.
 - 2.4 Wir addieren die Fehler in `cs` auf, setzen also `cs += error`.
 3. Das Endergebnis ist dann `sum + cs`.
- Probieren wir das am Beispiel `1e18 + 1 - 1e18` aus.
 1. Wir beginnen mit `sum = 0` und `cs = 0`.
 2. Als erste Zahl addieren wir `1e18` zur Summe.
 3. Jetzt addieren wir `1` zu der Summe.
 4. Wir ziehen nun `1e18` von der Summe ab, was das Gleiche wie `-1e18` dazuaddieren ist.
 5. Das Endergebnis ist dann `sum + cs` und das ergibt tatsächlich `0.0 + 1.0`, also `1.0!`
- Mit der Kahan-Summe konnten wir also `1e18 + 1 - 1e18` berechnen und erhalten korrekt `1.0`.
- Der Preis ist, dass wir zusätzliche Variablen brauchen und diese bei jedem Rechenschritt updaten müssen.

Kahan-Summe: Algorithmus

- Es gibt eine Vielzahl von verschiedenen Implementierungen dieser sogenannten Kahan-Summe der Kahan-Babuška-Summe.



Kahan-Summe: Algorithmus



- Es gibt eine Vielzahl von verschiedenen Implementierungen dieser sogenannten Kahan-Summe der Kahan-Babuška-Summe.
- Neumaier, z. B., hat 1974 eine Verbesserung für den Fall, dass die laufende Summe kleiner als die zu addierende Zahl ist, beigesteuert⁵⁶.

Algorithm 1: Die Kahan-Babuška-Neumaier Summe Zweiter Ordnung^{4,42,43,56} über einen Array x mit n Zahlen, nach der Spezifikation von Klein⁴³.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do
     $t \leftarrow sum + x[i]$ ;                                ▷ Für jede der  $n$  zu addierenden Zahlen in  $x$ .
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i]$ ;      ▷ Summe und Fehlerterm 1. Ordnung (unten)4,42.
    else  $c \leftarrow (x[i] - t) + sum$ ;
     $sum \leftarrow t$ ;                                         ▷ Die Gesamtsumme wird upgedated.
     $t \leftarrow cs + c$ ;                                     ▷ Fehlersumme 2. Ordnung von Klein43 beginnt.
    if  $|cs| \geq |c|$  then  $cc \leftarrow (cs - t) + c$ ;        ▷ Neumaier's Verbesserung56
    else  $c \leftarrow (c - t) + cs$ ;
     $cs \leftarrow t$ ;    $ccs \leftarrow ccs + cc$ ;
return  $sum + cs + ccs$ 
```



Kahan-Summe: Algorithmus

- Neumaier, z. B., hat 1974 eine Verbesserung für den Fall, dass die laufende Summe kleiner als die zu addierende Zahl ist, beigesteuert⁵⁶.
- Klein⁴³ hat dann die Präzision weiter verbessert, in dem eine Fehlersumme für die Fehlersumme eingeführt hat, also eine Fehlersumme 2. Ordnung, und so weiter.

Algorithm 1: Die Kahan-Babuška-Neumaier Summe Zweiter Ordnung^{4,42,43,56} über einen Array x mit n Zahlen, nach der Spezifikation von Klein⁴³.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do                                ▷ Für jede der  $n$  zu addierenden Zahlen in  $x$ .
     $t \leftarrow sum + x[i]$ ;                         ▷ Summe und Fehlerterm 1. Ordnung (unten)4,42.
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i]$ ;      ▷ Neumaier's Verbesserung56
    else  $c \leftarrow (x[i] - t) + sum$ ;
     $sum \leftarrow t$ ;                                ▷ Die Gesamtsumme wird upgedated.
     $t \leftarrow cs + c$ ;                           ▷ Fehlersumme 2. Ordnung von Klein43 beginnt.
    if  $|cs| \geq |c|$  then  $cc \leftarrow (cs - t) + c$ ;      ▷ Neumaier's Verbesserung56
    else  $c \leftarrow (c - t) + cs$ ;
     $cs \leftarrow t$ ;    $ccs \leftarrow ccs + cc$ ;
return  $sum + cs + ccs$ 
```



Kahan-Summe: Algorithmus

- Klein⁴³ hat dann die Präzision weiter verbessert, in dem eine Fehlersumme für die Fehlersumme eingeführt hat, also eine Fehlersumme 2. Ordnung, und so weiter.
- Dieser verbesserte Algorithmus ist hier dargestellt.

Algorithm 1: Die Kahan-Babuška-Neumaier Summe Zweiter Ordnung^{4,42,43,56} über einen Array x mit n Zahlen, nach der Spezifikation von Klein⁴³.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do                                ▷ Für jede der  $n$  zu addierenden Zahlen in  $x$ .
     $t \leftarrow sum + x[i]$ ;                         ▷ Summe und Fehlerterm 1. Ordnung (unten)4,42.
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i]$ ;      ▷ Neumaier's Verbesserung56
    else  $c \leftarrow (x[i] - t) + sum$ ;
     $sum \leftarrow t$ ;                                ▷ Die Gesamtsumme wird upgedated.
     $t \leftarrow cs + c$ ;                           ▷ Fehlersumme 2. Ordnung von Klein43 beginnt.
    if  $|cs| \geq |c|$  then  $cc \leftarrow (cs - t) + c$ ;      ▷ Neumaier's Verbesserung56
    else  $c \leftarrow (c - t) + cs$ ;
     $cs \leftarrow t$ ;    $ccs \leftarrow ccs + cc$ ;
return  $sum + cs + ccs$ 
```

Haben Sie keine Angst.

- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.



Haben Sie keine Angst.

- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.



Haben Sie keine Angst.

- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.



Haben Sie keine Angst.

- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen.



Haben Sie keine Angst.

- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.



Haben Sie keine Angst.



- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.

Haben Sie keine Angst.



- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.

Haben Sie keine Angst.



- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.
- Während wir den Algorithmus implementieren, also seine Komponenten auf Elemente der Programmiersprache übertragen, können wir lernen, ihn besser zu verstehen.

Haben Sie keine Angst.

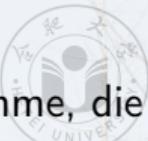


- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.
- Während wir den Algorithmus implementieren, also seine Komponenten auf Elemente der Programmiersprache übertragen, können wir lernen, ihn besser zu verstehen.
- Wir können den Kode auch Zwischenergebnisse für von uns gewählte Beispiele ausdrucken lassen.

Haben Sie keine Angst.



- Es ist nicht so wichtig, diesen Algorithmus hier im Detail zu diskutieren.
- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.
- Während wir den Algorithmus implementieren, also seine Komponenten auf Elemente der Programmiersprache übertragen, können wir lernen, ihn besser zu verstehen.
- Wir können den Kode auch Zwischenergebnisse für von uns gewählte Beispiele ausdrucken lassen.
- Dann können wir den Algorithmus nachverfolgen und sein Verhalten besser verstehen.



Haben Sie keine Angst.

- Im Grunde ist er ja nur eine weiterentwickelte Version der einfachen Kahan-Summe, die wir schon ausprobiert haben.
- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.
- Während wir den Algorithmus implementieren, also seine Komponenten auf Elemente der Programmiersprache übertragen, können wir lernen, ihn besser zu verstehen.
- Wir können den Kode auch Zwischenergebnisse für von uns gewählte Beispiele ausdrucken lassen.
- Dann können wir den Algorithmus nachverfolgen und sein Verhalten besser verstehen.
- Wir könnten ihn auch Schritt-für-Schritt mit einem Debugger ausführen (lernen wir später).

Haben Sie keine Angst.



- Ich habe mit Absicht diese weiterentwickelte, gruselig aussehende Variante des Algorithmus ausgewählt.
- Ich will nämlich einen wichtigen Punkt machen:
- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.
- Während wir den Algorithmus implementieren, also seine Komponenten auf Elemente der Programmiersprache übertragen, können wir lernen, ihn besser zu verstehen.
- Wir können den Kode auch Zwischenergebnisse für von uns gewählte Beispiele ausdrucken lassen.
- Dann können wir den Algorithmus nachverfolgen und sein Verhalten besser verstehen.
- Wir könnten ihn auch Schritt-für-Schritt mit einem Debugger ausführen (lernen wir später).
- Wenn wir eine klare Algorithmusdefinition haben, dann können wir diese nach Python übersetzen.

Haben Sie keine Angst.



- Ja. Manchmal sehen Algorithmen gefährlich und schwierig aus.
- Aber wenn wir der Definition ordentlich folgen, dann können wir sie trotzdem implementieren.
- Selbst wenn wir am Anfang nicht alles ganz verstehen.
- Während wir den Algorithmus implementieren, also seine Komponenten auf Elemente der Programmiersprache übertragen, können wir lernen, ihn besser zu verstehen.
- Wir können den Kode auch Zwischenergebnisse für von uns gewählte Beispiele ausdrucken lassen.
- Dann können wir den Algorithmus nachverfolgen und sein Verhalten besser verstehen.
- Wir könnten ihn auch Schritt-für-Schritt mit einem Debugger ausführen (lernen wir später).
- Wenn wir eine klare Algorithmusdefinition haben, dann können wir diese nach Python übersetzen.
- Und wir können davon lernen.

Design der Klasse



- Jetzt implementieren wir die Kahan-Babuška-Neumaier Summe Zweiter Ordnung in Datei `kahan_sum.py`.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

Design der Klasse



- Jetzt implementieren wir die Kahan-Babuška-Neumaier Summe Zweiter Ordnung in Datei **kahan_sum.py**.
- Die Frage ist nun *Wie machen wir das?*.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

Design der Klasse



- Jetzt implementieren wir die Kahan-Babuška-Neumaier Summe Zweiter Ordnung in Datei `kahan_sum.py`.
- Die Frage ist nun *Wie machen wir das?*.
- Wir könnten ihn als Funktion implementieren, die ein Iterable `x` mit den zu summierenden Zahlen als Parameter erhält.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n – 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum – t) + x[i];
    else c ← (x[i] – t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs – t) + c;
    else c ← (c – t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

Design der Klasse



- Die Frage ist nun *Wie machen wir das?*.
- Wir könnten ihn als Funktion implementieren, die ein Iterable x mit den zu summierenden Zahlen als Parameter erhält.
- So ist der Algorithmus ja im Grunde definiert.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do
     $t \leftarrow sum + x[i];$ 
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i];$ 
    else  $c \leftarrow (x[i] - t) + sum;$ 
     $sum \leftarrow t;$ 
     $t \leftarrow cs + c;$ 
    if  $|cs| \geq |c|$  then  $cc \leftarrow (cs - t) + c;$ 
    else  $c \leftarrow (c - t) + cs;$ 
     $cs \leftarrow t;  ccs \leftarrow ccs + cc;$ 
return  $sum + cs + ccs$ 
```

Design der Klasse



- Wir könnten ihn als Funktion implementieren, die ein `Iterable` `x` mit den zu summierenden Zahlen als Parameter erhält.
- So ist der Algorithmus ja im Grunde definiert.
- Wir entscheiden uns anders.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n – 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum – t) + x[i];
    else c ← (x[i] – t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs – t) + c;
    else c ← (c – t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```



Design der Klasse

- So ist der Algorithmus ja im Grunde definiert.
- Wir entscheiden uns anders.
- Wir wollen ihn als Klasse `KahanSum` implementieren, die eine laufende Summe darstellt.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do
    t ← sum +  $x[i]$ ;
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i]$ ;
    else  $c \leftarrow (x[i] - t) + sum$ ;
    sum ← t;
    t ← cs + c;
    if  $|cs| \geq |c|$  then cc ← (cs - t) + c;
    else  $c \leftarrow (c - t) + cs$ ;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum



Design der Klasse

- Wir entscheiden uns anders.
- Wir wollen ihn als Klasse `KahanSum` implementieren, die eine laufende Summe darstellt.
- Eine Klasse hat Attribute.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```





Design der Klasse

- Wir wollen ihn als Klasse `KahanSum` implementieren, die eine laufende Summe darstellt.
- Eine Klasse hat Attribute und Methoden.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```





Design der Klasse

- Wir wollen ihn als Klasse `KahanSum` implementieren, die eine laufende Summe darstellt.
- Eine Klasse hat Attribute und Methoden.
- Schauen wir uns den Algorithmus genauer an.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do
    t ← sum +  $x[i]$ ;
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i]$ ;
    else  $c \leftarrow (x[i] - t) + sum$ ;
    sum ← t;
    t ← cs + c;
    if  $|cs| \geq |c|$  then cc ← (cs - t) + c;
    else  $c \leftarrow (c - t) + cs$ ;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```





Design der Klasse

- Eine Klasse hat Attribute und Methoden.
- Schauen wir uns den Algorithmus genauer an.
- Er beginnt mit der Initialisierung der Zustandsvariablen sum , cs , und ccs .

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do
     $t \leftarrow sum + x[i]$ ;
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i]$ ;
    else  $c \leftarrow (x[i] - t) + sum$ ;
     $sum \leftarrow t$ ;
     $t \leftarrow cs + c$ ;
    if  $|cs| \geq |c|$  then  $cc \leftarrow (cs - t) + c$ ;
    else  $c \leftarrow (c - t) + cs$ ;
     $cs \leftarrow t$ ;   $ccs \leftarrow ccs + cc$ ;
return  $sum + cs + ccs$ 
```

KahanSum

attributes

methods



Design der Klasse

- Schauen wir uns den Algorithmus genauer an.
- Er beginnt mit der Initialisierung der Zustandsvariablen sum , cs , und ccs .
- Diese müsste man als Attribute der Instanzen der Klasse realisieren.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for  $i \in 0..n - 1$  do
     $t \leftarrow sum + x[i];$ 
    if  $|sum| \geq |x[i]|$  then  $c \leftarrow (sum - t) + x[i];$ 
    else  $c \leftarrow (x[i] - t) + sum;$ 
     $sum \leftarrow t;$ 
     $t \leftarrow cs + c;$ 
    if  $|cs| \geq |c|$  then  $cc \leftarrow (cs - t) + c;$ 
    else  $c \leftarrow (c - t) + cs;$ 
     $cs \leftarrow t;  ccs \leftarrow ccs + cc;$ 
return  $sum + cs + ccs$ 
```

KahanSum

attributes

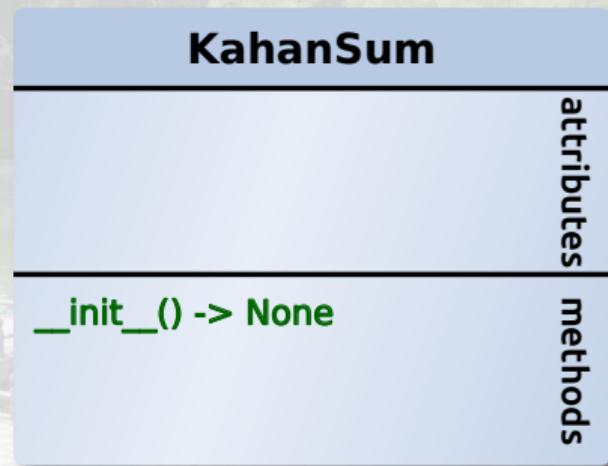
methods



Design der Klasse

- Er beginnt mit der Initialisierung der Zustandsvariablen sum , cs , und ccs .
- Diese müsste man als Attribute der Instanzen der Klasse realisieren.
- In einer Klasse kommt der Kode, der Attribute initialisiert, in den Initialisierer `__init__`.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs;
```

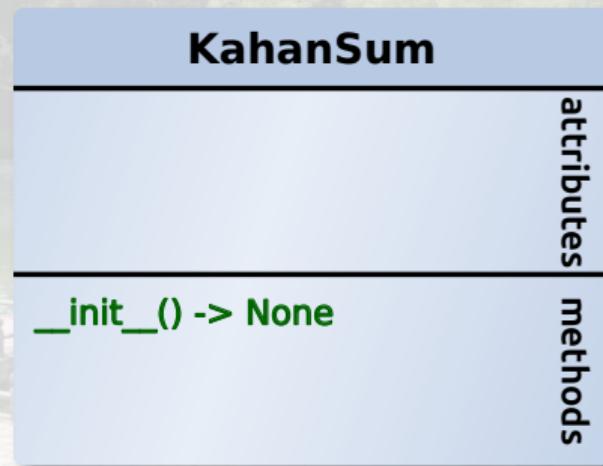


Design der Klasse



- Diese müsste man als Attribute der Instanzen der Klasse realisieren.
- In einer Klasse kommt der Kode, der Attribute initialisiert, in den Initialisierer `__init__`.
- Also tun wir diesen Teil des Algorithmus auch dort hin.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs;
```

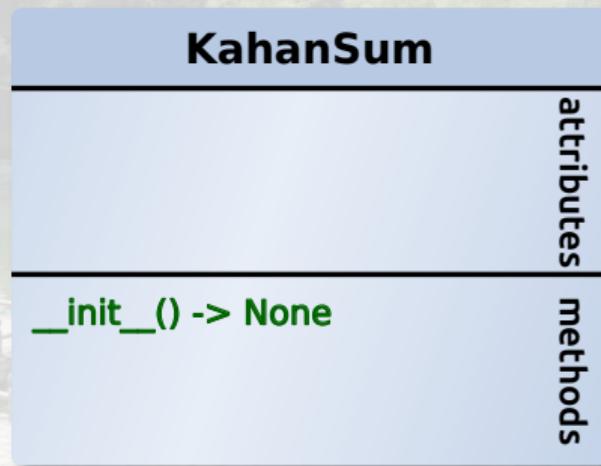


Design der Klasse



- In einer Klasse kommt der Code, der Attribute initialisiert, in den Initialisierer `__init__`.
- Also tun wir diesen Teil des Algorithmus auch dort hin.
- In einer Schleife addiert der Algorithmus eine Zahl nach der anderen zu der Summe hinzu.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs;
```





Design der Klasse

- Also tun wir diesen Teil des Algorithmus auch dort hin.
- In einer Schleife addiert der Algorithmus eine Zahl nach der anderen zu der Summe hinzu.
- Wir wollen eine Methode `add` implementieren, die jeweils genau eine Zahl zur Summe hinzufügt.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__()` -> None

`add(value: int | float)` -> None



Design der Klasse

- In einer Schleife addiert der Algorithmus eine Zahl nach der anderen zu der Summe hinzu.
- Wir wollen eine Methode `add` implementieren, die jeweils genau eine Zahl zur Summe hinzufügt.
- Diese könnte ja dann genauso gut in einer Schleife aufgerufen werden.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__()` -> None

`add(value: int | float)` -> None

Design der Klasse



- Wir wollen eine Methode `add` implementieren, die jeweils genau eine Zahl zur Summe hinzufügt.
- Diese könnte ja dann genauso gut in einer Schleife aufgerufen werden.
- Der Parameter `value` tritt dann an Stelle der Schleifenvariable $x[i]$.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__()` -> None

`add(value: int | float)` -> None



Design der Klasse

- Diese könnte ja dann genauso gut in einer Schleife aufgerufen werden.
- Der Parameter `value` tritt dann an Stelle der Schleifenvariable $x[i]$.
- Nach dem Ende der Schleife, am Ende des Algorithmus, wird die entgültige Summe berechnet.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

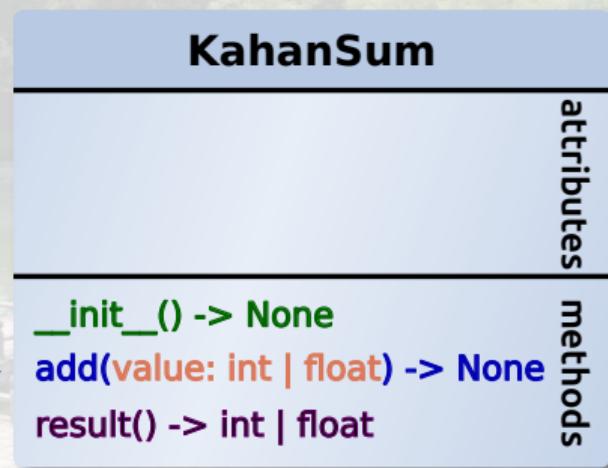
`add(value: int | float) -> None`



Design der Klasse

- Der Parameter `value` tritt dann an Stelle der Schleifenvariable $x[i]$.
- Nach dem Ende der Schleife, am Ende des Algorithmus, wird die entgültige Summe berechnet.
- Dazu werden die Summenvariable sum und die Fehlerterme 1. und 2. Ordnung (cs und ccs) addiert.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```





Design der Klasse

- Nach dem Ende der Schleife, am Ende des Algorithmus, wird die entgültige Summe berechnet.
- Dazu werden die Summenvariable *sum* und die Fehlerterme 1. und 2. Ordnung (*cs* und *ccs*) addiert.
- Wir packen das in eine Methode `result`, die das Ergebnis liefern soll.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

`result() -> int | float`



Design der Klasse

- Dazu werden die Summenvariable *sum* und die Fehlerterme 1. und 2. Ordnung (*cs* und *ccs*) addiert.
- Wir packen das in eine Methode `result`, die das Ergebnis liefern soll.
- Genaugenommen ist dieses „entgültige“ Ergebnis gar nicht so entgültig.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
__init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Design der Klasse

- Wir packen das in eine Methode `result`, die das Ergebnis liefern soll.
- Genaugenommen ist dieses „entgültige“ Ergebnis gar nicht so entgültig.
- Bei seiner Berechnung werden ja keine Attribute verändert.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__()` -> None

`add(value: int | float)` -> None

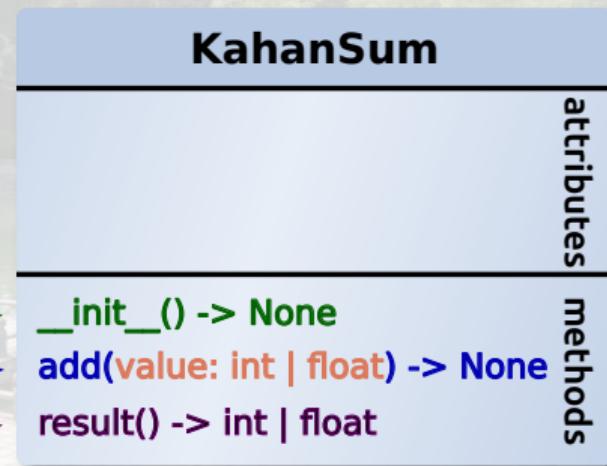
`result()` -> int | float



Design der Klasse

- Genaugenommen ist dieses „entgültige“ Ergebnis gar nicht so entgültig.
- Bei seiner Berechnung werden ja keine Attribute verändert.
- Wir könnten also einfach ein paar Zahlen mit `add` addieren, das Ergebnis mit `result` abfragen, dann noch ein paar Zahlen dazuaddieren, dann das Ergebnis nochmal abfragen.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```





Design der Klasse

- Bei seiner Berechnung werden ja keine Attribute verändert.
- Wir könnten also einfach ein paar Zahlen mit `add` addieren, das Ergebnis mit `result` abfragen, dann noch ein paar Zahlen dazuaddieren, dann das Ergebnis nochmal abfragen.
- Wir haben eine wirkliche laufende Summe.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__()` -> None

`add(value: int | float)` -> None

`result()` -> int | float



Design der Klasse

- Wir könnten also einfach ein paar Zahlen mit `add` addieren, das Ergebnis mit `result` abfragen, dann noch ein paar Zahlen dazuaddieren, dann das Ergebnis nochmal abfragen.
- Wir haben eine wirkliche laufende Summe.
- Was wir noch nicht diskutiert haben, sind die Attribute unserer Klasse.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
_init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Design der Klasse

- Wir haben eine wirkliche laufende Summe.
- Was wir noch nicht diskutiert haben, sind die Attribute unserer Klasse.
- Wir brauchen Attribute für die Summe sum , den Fehlerterm erster Ordnung cs und den Fehlerterm zweiter Ordnung ccs .

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
__init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Design der Klasse

- Was wir noch nicht diskutiert haben, sind die Attribute unserer Klasse.
- Wir brauchen Attribute für die Summe sum , den Fehlerterm erster Ordnung cs und den Fehlerterm zweiter Ordnung ccs .
- Die Werte dieser Attribute ändern sich während der Summation, also sind sie nicht **Final**.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
__init__() -> None
add(value: int | float) -> None
result() -> int | float
```

Design der Klasse



- Wir brauchen Attribute für die Summe *sum*, den Fehlerterm erster Ordnung *cs* und den Fehlerterm zweiter Ordnung *ccs*.
- Die Werte dieser Attribute ändern sich während der Summation, also sind sie nicht `Final`.
- Sie repräsentieren den *internen* Zustand unserer Summe.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
_init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Design der Klasse

- Die Werte dieser Attribute ändern sich während der Summation, also sind sie nicht **Final**.
- Sie repräsentieren den *internen* Zustand unserer Summe.
- Sie sind bedeutungslos für jeden anderen, äußeren Kode.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

`result() -> int | float`



Design der Klasse

- Sie repräsentieren den *internen* Zustand unserer Summe.
- Sie sind bedeutungslos für jeden anderen, äußeren Kode.
- Wahrscheinlich kann niemand außer uns ihre Bedeutung verstehen.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

`result() -> int | float`



Design der Klasse

- Sie sind bedeutungslos für jeden anderen, äußereren Kode.
- Wahrscheinlich kann niemand außer uns ihre Bedeutung verstehen.
- Deshalb sollte auch niemand sie sehen oder gar verändern können.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

`result() -> int | float`



Design der Klasse

- Wahrscheinlich kann niemand außer uns ihre Bedeutung verstehen.
- Deshalb sollte auch niemand sie sehen oder gar verändern können.
- Denn dafür gibt es ja keinen Grund.

```
sum ← 0;  cs ← 0;  ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t;  ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
__init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Design der Klasse

- Deshalb sollte auch niemand sie sehen oder gar verändern können.
- Denn dafür gibt es ja keinen Grund.
- Die Attribute sollten „versteckt“ sein.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

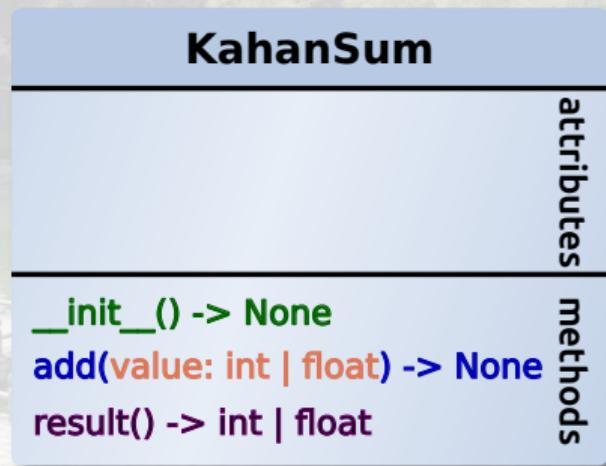
`result() -> int | float`



Design der Klasse

- Denn dafür gibt es ja keinen Grund.
- Die Attribute sollten „versteckt“ sein.
- Anders als bei den Attributen `x` und `y` unserer Klasse `Point`, hat der Benutzer (ein anderer Programmierer) keinen Grund, auf den (internen) Zustand unserer Summe zuzugreifen.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

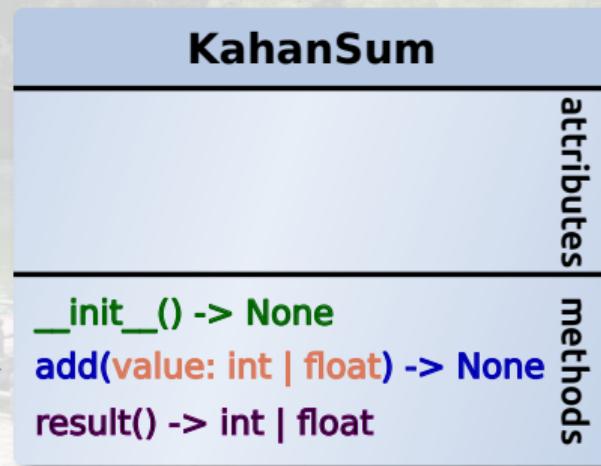


Design der Klasse



- Die Attribute sollten „versteckt“ sein.
- Anders als bei den Attributen `x` und `y` unserer Klasse `Point`, hat der Benutzer (ein anderer Programmierer) keinen Grund, auf den (internen) Zustand unserer Summe zuzugreifen.
- Stattdessen sollte er mit unseren Objekten nur über die Methoden `add` und `result` interagieren.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```





Definition: (Vollständige) Kapselung

Kapselung (EN: *encapsulation*) bedeutet, dass auf die Attribute eines Objekts nur durch seine Methoden zugegriffen wird.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
__init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Definition: (Vollständige) Kapselung

Kapselung (EN: *encapsulation*) bedeutet, dass auf die Attribute eines Objekts nur durch seine Methoden zugegriffen wird. Unter vollständiger Kapselung (EN: *complete encapsulation*) können die Attribute eines Objekts nur durch seine Methoden gelesen oder verändert werden.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

```
__init__() -> None
add(value: int | float) -> None
result() -> int | float
```



Definition: (Vollständige) Kapselung

Kapselung (EN: *encapsulation*) bedeutet, dass auf die Attribute eines Objekts nur durch seine Methoden zugegriffen wird. Unter vollständiger Kapselung (EN: *complete encapsulation*) können die Attribute eines Objekts nur durch seine Methoden gelesen oder verändert werden. Kapselung erlaubt es daher Programmierern, sicherzustellen, dass der Zustand von Objekten nur einer konsistenten und korrekten Art und Weise verändert werden kann.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

`result() -> int | float`

Design der Klasse



- Anders als bei den Attributen `x` und `y` unserer Klasse `Point`, hat der Benutzer (ein anderer Programmierer) keinen Grund, auf den (internen) Zustand unserer Summe zuzugreifen.
- Stattdessen sollte er mit unseren Objekten nur über die Methoden `add` und `result` interagieren.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes

methods

`__init__() -> None`

`add(value: int | float) -> None`

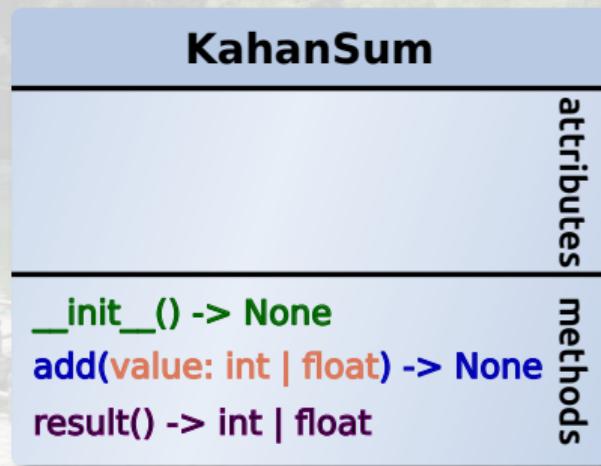
`result() -> int | float`

Design der Klasse



- Anders als bei den Attributen `x` und `y` unserer Klasse `Point`, hat der Benutzer (ein anderer Programmierer) keinen Grund, auf den (internen) Zustand unserer Summe zuzugreifen.
- Stattdessen sollte er mit unseren Objekten nur über die Methoden `add` und `result` interagieren.
- Wir wollen unsere Objekte vollständig kapseln.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```



Design der Klasse



- Stattdessen sollte er mit unseren Objekten nur über die Methoden `add` und `result` interagieren.
- Wir wollen unsere Objekte vollständig kapseln.
- Wir erstellen drei Attribute `__sum`, `__cs` und `__ccs`.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum	
attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>



Design der Klasse

- Wir wollen unsere Objekte vollständig kapseln.
- Wir erstellen drei Attribute `__sum`, `__cs` und `__ccs`.
- Die Namen stammen aus dem Algorithmus, aber sehen Sie die beiden Unterstriche jeweils am Anfang?

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>



Gute Praxis

Namen von Attributen oder Methoden, die mit einem doppelten Unterstrich beginnen (`__`) müssen als *privat* betrachtet werden^{68,69}.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

`__sum: int | float`

`__cs: int | float`

`__ccs: int | float`

`__init__() -> None`

`add(value: int | float) -> None`

`result() -> int | float`

attributes

methods



Gute Praxis

Namen von Attributen oder Methoden, die mit einem doppelten Unterstrich beginnen (`__`) müssen als *privat* betrachtet werden^{68,69}. Auf sie sollte nicht von außerhalb der Klasse zugegriffen werden.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum	
attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>



Gute Praxis

Namen von Attributen oder Methoden, die mit einem doppelten Unterstrich beginnen (`__`) müssen als *privat* betrachtet werden^{68,69}. Auf sie sollte nicht von außerhalb der Klasse zugegriffen werden. Alle internen Attribute und Methoden einer Klasse sollte nicht von außen zugänglich sein und sollten deshalb nach dieser Konvention benannt werden, mit einem doppelten Unterstrich als Präfix.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum	
attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>

Design der Klasse



- Wir erstellen drei Attribute `__sum`, `__cs` und `__ccs`.
- Die Namen stammen aus dem Algorithmus, aber sehen Sie die beiden Unterstriche jeweils am Anfang?
- Mit anderen Worten, niemand sollte von Außen auf `__sum`, `__cs` oder `__ccs`.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>

Design der Klasse



- Die Namen stammen aus dem Algorithmus, aber sehen Sie die beiden Unterstriche jeweils am Anfang?
- Mit anderen Worten, niemand sollte von Außen auf `__sum`, `__cs` oder `__ccs`.
- Wie alle solche Dinge in Python wird das natürlich **nicht** vom Interpreter erzwungen...

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum	
attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>

Design der Klasse



- Mit anderen Worten, niemand sollte von Außen auf `__sum`, `__cs` oder `__ccs`.
- Wie alle solche Dinge in Python wird das natürlich **nicht** vom Interpreter erzwungen...
- ... es ist also kein absoluter Schutz unserer Attribute.

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes	methods
<code>__sum: int float</code>	
<code>__cs: int float</code>	
<code>__ccs: int float</code>	
	<code>__init__() -> None</code>
	<code>add(value: int float) -> None</code>
	<code>result() -> int float</code>

Design der Klasse



- Wie alle solche Dinge in Python wird das natürlich **nicht** vom Interpreter erzwungen.
- ... es ist also kein absoluter Schutz unserer Attribute.
- Es ist nur ein sehr klarer Hinweis an andere Programmierer, dass sie da die Finger von lassen sollen.

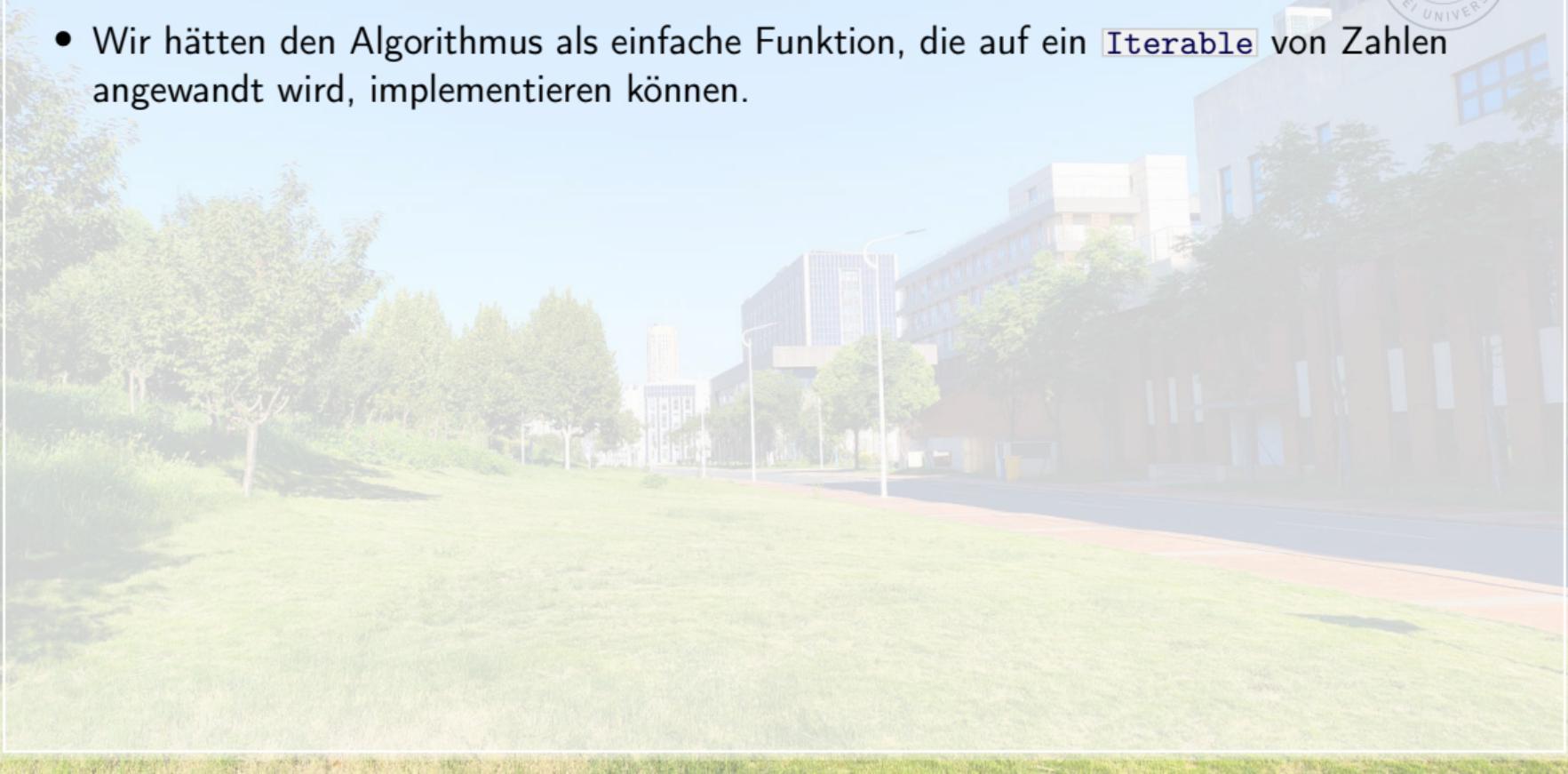
```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n - 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum - t) + x[i];
    else c ← (x[i] - t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs - t) + c;
    else c ← (c - t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

KahanSum

attributes	methods
sum: int float	
cs: int float	
ccs: int float	
	__init__() -> None
	add(value: int float) -> None
	result() -> int float

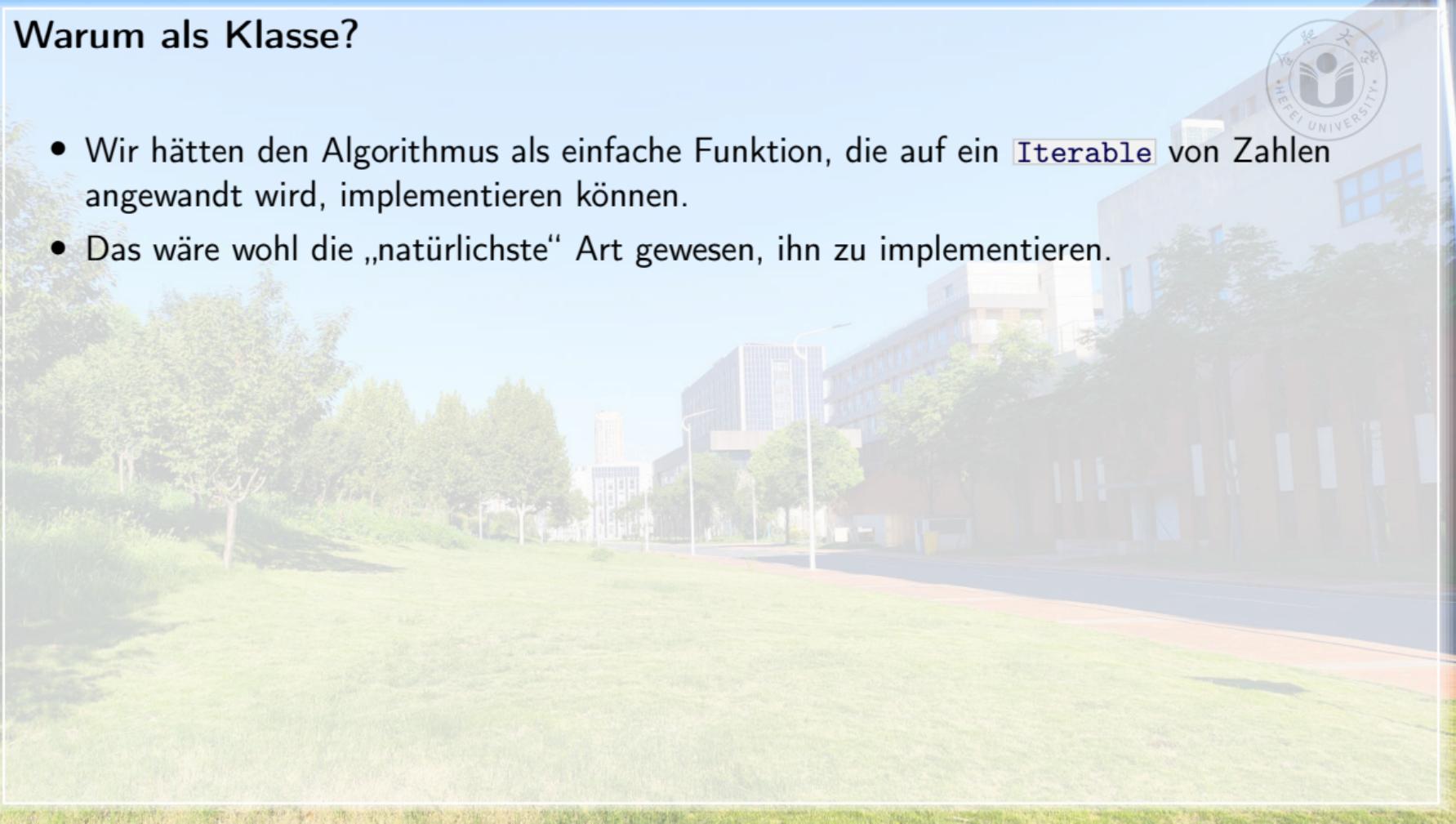
Warum als Klasse?

- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.



Warum als Klasse?

- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.



Warum als Klasse?



- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.
- Wir haben aber das Klassen-basierte Design gewählt, weil es uns größere Flexibilität gibt.

Warum als Klasse?



- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.
- Wir haben aber das Klassen-basierte Design gewählt, weil es uns größere Flexibilität gibt.
- Wir können damit natürlich trotzdem über Sequenzen von Zahlen summieren.

Warum als Klasse?



- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.
- Wir haben aber das Klassen-basierte Design gewählt, weil es uns größere Flexibilität gibt.
- Wir können damit natürlich trotzdem über Sequenzen von Zahlen summieren.
- Wir können aber eben auch anders summieren.

Warum als Klasse?



- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.
- Wir haben aber das Klassen-basierte Design gewählt, weil es uns größere Flexibilität gibt.
- Wir können damit natürlich trotzdem über Sequenzen von Zahlen summieren.
- Wir können aber eben auch anders summieren.
- Wir können Zwischenergebnisse abfragen.

Warum als Klasse?



- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.
- Wir haben aber das Klassen-basierte Design gewählt, weil es uns größere Flexibilität gibt.
- Wir können damit natürlich trotzdem über Sequenzen von Zahlen summieren.
- Wir können aber eben auch anders summieren.
- Wir können Zwischenergebnisse abfragen.
- Wir können auch mehrere `KahanSum`-Objekte auf einmal verwenden.

Warum als Klasse?



- Wir hätten den Algorithmus als einfache Funktion, die auf ein `Iterable` von Zahlen angewandt wird, implementieren können.
- Das wäre wohl die „natürliche“ Art gewesen, ihn zu implementieren.
- Wir haben aber das Klassen-basierte Design gewählt, weil es uns größere Flexibilität gibt.
- Wir können damit natürlich trotzdem über Sequenzen von Zahlen summieren.
- Wir können aber eben auch anders summieren.
- Wir können Zwischenergebnisse abfragen.
- Wir können auch mehrere `KahanSum`-Objekte auf einmal verwenden.
- Wenn wir z. B. einen Datenstrom haben, dann könnten wir zwei Instanzen von `KahanSum` verwenden, um die Zahlen (in denen) zu addieren und um deren Quadrate aufzusummieren (in der anderen Instanz), was durchaus nützlich seien kann, wenn wir z. B. die Sample-Varianz bestimmen wollen.



Kahan-Summe: Implementation

- Implementieren wir diesen Algorithmus nun also.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0;  cs ← 0;  ccs ← 0;  
for i ∈ 0..n − 1 do  
    t ← sum + x[i];  
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];  
    else c ← (x[i] − t) + sum;  
    sum ← t;  
    t ← cs + c;  
    if |cs| ≥ |c| then cc ← (cs − t) + c;  
    else c ← (c − t) + cs;  
    cs ← t;  ccs ← ccs + cc;  
return sum + cs + ccs
```

Kahan-Summe: Implementation

- Eine neue Instanz von `KahanSum` beginnt mit allen ihren drei Summenattributen auf 0 gesetzt, was im Initialisierer `__init__` passiert.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir implementieren zuerst die Methode `add`.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                           else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                            else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir implementieren zuerst die Methode `add`.
- Diese entspricht dem Körper der Schleife im Algorithmus.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Diese entspricht dem Körper der Schleife im Algorithmus.
- Im Algorithmus wird in jeder Iteration ein Wert $x[i]$ zur Summe addiert.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Unsere Methode `add` hat dafür den Parameter `value`, mit dem Wert den wir zur Summe addieren wollen.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir müssen also nur den Schleifenkörper vom Algorithmus in die Methode `add` übertragen.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                           else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                            else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Dabei ersetzen wir sum , cs , ccs und $x[i]$ mit `self.__sum`, `self.__cs`, `self.__ccs` und `value`.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir schreiben die Verbesserung von Neumaier⁵⁶ als ein inline `if...else`-Statement.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                           else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Um den Absolutwert $|a|$ einer Zahl a zu berechnen, können wir Python's Funktion `abs` verwenden.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Um den Absolutwert $|a|$ einer Zahl a zu berechnen, können wir Python's Funktion `abs` verwenden.
- Und das war's auch schon.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Es gibt nichts im Schleifenkörper vom Algorithmus, das wir nicht mehr oder weniger direkt in Python-Kode übertragen können.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                           else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir können nun die letzte Zeile des Algorithmus, die das Endergebnis der Summe liefert, in die neue Methode `result` packen.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Natürlich wird diese auf die Attribute `self.__sum`, `self.__cs` und `self.__ccs` anstelle von `sum`, `cs` und `ccs` zugreifen.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Und damit ist das fertig.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Und damit ist das fertig.
- Wir haben einen relativ komplexen mathematischen Algorithmus in ein Stück Python-Kode übersetzt.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir haben einen monolithischen Algorithmus in eine API verpackt, die iterativ aufgerufen werden kann.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir haben einen monolithischen Algorithmus in eine API verpackt, die iterativ aufgerufen werden kann.
- Aber funktioniert er auch?

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Aber funktioniert er auch?
- Wir platzieren erstmal einen Doctest in den Docstring des Moduls.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Dieser Doctest berechnet die Summe
 $1e18 + 1 + 1e36 - 1e36 - 1e18$,
die unser Beispiel am Anfang der Einheit war.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Dort haben wir gelernt, dass wenn wir die Summe direkt berechnen, einfach 0.0 rauskommt.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;  
for i ∈ 0..n − 1 do  
    t ← sum + x[i];  
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];  
    else c ← (x[i] − t) + sum;  
    sum ← t;  
    t ← cs + c;  
    if |cs| ≥ |c| then cc ← (cs − t) + c;  
    else c ← (c − t) + cs;  
    cs ← t; ccs ← ccs + cc;  
return sum + cs + ccs
```

```
"""  
The second-order Kahan-Babuška-Neumaier-Summation by Klein.  
A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.  
Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x  
"""  
kahan_sum = KahanSum()  
for xi in [1e18, 1, 1e36, -1e36, -1e18]:  
...    kahan_sum.add(xi)  
kahan_sum.result()  
1.0  
"""  
  
class KahanSum:  
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""  
  
    def __init__(self) -> None:  
        """Create the summation object."""  
        #: the running sum, an internal variable invisible from outside  
        self.__sum: float | int = 0  
        #: the first correction term, another internal variable  
        self.__cs: float | int = 0  
        #: the second correction term, another internal variable  
        self.__ccs: float | int = 0  
  
    def add(self, value: int | float) -> None:  
        """  
        Add a value to the sum.  
  
        :param value: the value to add  
        """  
        s: int | float = self.__sum # Get the current running sum.  
        t: int | float = s + value # Compute the new sum value.  
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)  
                         else ((value - t) + s)) # The Neumaier tweak.  
        self.__sum = t # Store the new sum value.  
        cs: int | float = self.__cs # the current 1st-order correction  
        t = cs + c # Compute the new first-order correction term.  
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)  
                           else ((c - t) + cs)) # 2nd Neumaier tweak.  
        self.__cs = t # Store the updated first-order correction term.  
        self.__ccs += cc # Update the second-order correction.  
  
    def result(self) -> int | float:  
        """  
        Get the current result of the summation.  
  
        :return: the current result of the summation  
        """  
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wir wissen aber, dass das richtige Ergebnis 1.0 ist.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                           else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                            else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wenn wir mit unserer Klasse `KahanSum` die Zahlen `[1e18, 1, 1e36, -1e36, -1e18]` aufaddieren, dann *sollte* dieses richtige Ergebnis herauskommen.

Algorithm 1: Kahan-Babuška-Neumaier Summe

```
sum ← 0; cs ← 0; ccs ← 0;
for i ∈ 0..n − 1 do
    t ← sum + x[i];
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;
    if |cs| ≥ |c| then cc ← (cs − t) + c;
    else c ← (c − t) + cs;
    cs ← t; ccs ← ccs + cc;
return sum + cs + ccs
```

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""

class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum # Get the current running sum.
        t: int | float = s + value # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                           else ((value - t) + s)) # The Neumaier tweak.
        self.__sum = t # Store the new sum value.
        cs: int | float = self.__cs # the current 1st-order correction
        t = cs + c # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                            else ((c - t) + cs)) # 2nd Neumaier tweak.
        self.__cs = t # Store the updated first-order correction term.
        self.__ccs += cc # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Kahan-Summe: Implementation

- Wenn wir mit unserer Klasse `KahanSum` die Zahlen `[1e18, 1, 1e36, -1e36, -1e18]` aufaddieren, dann *sollte* dieses richtige Ergebnis herauskommen.
- Wir wir sehen, tut es das auch.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules kahan_sum
2   ↪ .py
3 ===== test session starts
4   ↪ =====
5 collected 1 item
6
7 kahan_sum.py . [100%]
8
9 ===== 1 passed in 0.02s
10  ↪ =====
11 # pytest 9.0.2 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

```
1 """
2 The second-order Kahan-Babuška-Neumaier-Summation by Klein.
3
4 [1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
5 Computing 76:279–293. 2006. doi:10.1007/s00607-005-0139-x
6
7 >>> kahan_sum = KahanSum()
8 >>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
9 ...     kahan_sum.add(xi)
10 >>> kahan_sum.result()
11 1.0
12 """
13
14
15 class KahanSum:
16     """The second-order Kahan-Babuška-Neumaier sum by Klein."""
17
18     def __init__(self) -> None:
19         """Create the summation object."""
20         #: the running sum, an internal variable invisible from outside
21         self.__sum: float | int = 0
22         #: the first correction term, another internal variable
23         self.__cs: float | int = 0
24         #: the second correction term, another internal variable
25         self.__ccs: float | int = 0
26
27     def add(self, value: int | float) -> None:
28         """
29             Add a value to the sum.
30
31             :param value: the value to add
32             """
33             s: int | float = self.__sum # Get the current running sum.
34             t: int | float = s + value # Compute the new sum value.
35             c: int | float = (((s - t) + value) if abs(s) >= abs(value)
36                               else ((value - t) + s)) # The Neumaier tweak.
37             self.__sum = t # Store the new sum value.
38             cs: int | float = self.__cs # the current 1st-order correction
39             t = cs + c # Compute the new first-order correction term.
40             cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
41                               else ((c - t) + cs)) # 2nd Neumaier tweak.
42             self.__cs = t # Store the updated first-order correction term.
43             self.__ccs += cc # Update the second-order correction.
44
45     def result(self) -> int | float:
46         """
47             Get the current result of the summation.
48
49             :return: the current result of the summation
50             """
51             return self.__sum + self.__cs + self.__ccs
```

Kahan Summe: Vergleich

- Im Programm `kahan_user.py` benutzen wir nun unsere neue KahanSum Klasse.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [ 1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12     print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13 k: KahanSum = KahanSum() # Create our Kahan summation object.
14 for n in numbers:        # Iterate over the numbers...
15     k.add(n)              # ...and let our object add them up.
16     print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17     print(f"Kahan sum      = {k.result()}") # our better result
18     print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ `python3 kahan_user.py` ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum    = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum    = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum    = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↪ =====
14 sum(numbers) = -1e+18
15 Kahan sum    = 0.0
16 fsum(numbers) = 1.0
17 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
18 sum(numbers) = 1.0
19 Kahan sum    = 1.0
20 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- Im Programm `kahan_user.py` benutzen wir nun unsere neue `KahanSum` Klasse.
- Wir vergleichen ihre Ergebnisse mit der built-in Funktion `sum` und der genauen Summenfunktion `fsum` aus dem `math` Modul.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [ 1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:        # Iterate over the numbers...
15        k.add(n)             # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↪ =====
14 sum(numbers) = -1e+18
15 Kahan sum   = 0.0
16 fsum(numbers) = 1.0
17 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
18 sum(numbers) = 1.0
19 Kahan sum   = 1.0
20 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- Im Programm `kahan_user.py` benutzen wir nun unsere neue `KahanSum` Klasse.
- Wir vergleichen ihre Ergebnisse mit der built-in Funktion `sum` und der genauen Summenfunktion `fsum` aus dem `math` Modul.
- Alle drei können `[1e-15, 1e-14, 1e-13, 1e-16, 1e-12]` exakt zu `1.1111e-12` aufaddieren.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:       # Iterate over the numbers...
15        k.add(n)           # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ `python3 kahan_user.py` ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↪ =====
14 sum(numbers) = -1e+18
15 Kahan sum   = 0.0
16 fsum(numbers) = 1.0
17 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
18 sum(numbers) = 1.0
19 Kahan sum   = 1.0
20 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- Im Programm `kahan_user.py` benutzen wir nun unsere neue `KahanSum` Klasse.
- Wir vergleichen ihre Ergebnisse mit der built-in Funktion `sum` und der genauen Summenfunktion `fsum` aus dem `math` Modul.
- Alle drei können $[1e-15, 1e-14, 1e-13, 1e-16, 1e-12]$ exakt zu $1.1111e-12$ aufaddieren.
- `sum` liefert 0.0 für die Summe von $[1e+18, 1, -1e+18]$ und auch für $[1e+36, 1e+18, 1, -1e+36, -1e+18]$.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:       # Iterate over the numbers...
15        k.add(n)           # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↪ =====
14 sum(numbers) = -1e+18
15 Kahan sum   = 0.0
16 fsum(numbers) = 1.0
17 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
18 sum(numbers) = 1.0
19 Kahan sum   = 1.0
20 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- Wir vergleichen ihre Ergebnisse mit der built-in Funktion `sum` und der genauen Summenfunktion `fsum` aus dem `math` Modul.
- Alle drei können $[1e-15, 1e-14, 1e-13, 1e-16, 1e-12]$ exakt zu $1.1111e-12$ aufaddieren.
- `sum` liefert 0.0 für die Summe von $[1e+18, 1, -1e+18]$ und auch für $[1e+36, 1e+18, 1, -1e+36, -1e+18]$.
- `KahanSum` und `fsum` können für beide Fälle korrekt 1.0 ausrechnen.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:       # Iterate over the numbers...
15        k.add(n)           # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14 sum(numbers) = -1e+18
15 Kahan sum   = 0.0
16 fsum(numbers) = 1.0
17 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
18 sum(numbers) = 1.0
19 Kahan sum   = 1.0
20 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- Alle drei können

[$1e-15, 1e-14, 1e-13,$

$1e-16, 1e-12]$ exakt zu

$1.1111e-12$ aufaddieren.

- `sum` liefert 0.0 für die Summe von
[$1e+18, 1, -1e+18$] und auch für
[$1e+36, 1e+18, 1,$
 $-1e+36, -1e+18$].

- `KahanSum` und `fsum` können für beide Fälle korrekt 1.0 ausrechnen.

- Wenn wir aber noch größere Zahlen mit einbeziehen und z.B. die Summe über [$1e+36, 1e+72, 1e+18,$
 $-1e+36, -1e+72, 1, -1e+18$] berechnen, dann liefert unsere `KahanSum` 0.0 anstatt des korrekten Ergebnisses 1.0 .

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:       # Iterate over the numbers...
15        k.add(n)           # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↗ =====
15 sum(numbers) = -1e+18
16 Kahan sum   = 0.0
17 fsum(numbers) = 1.0
18 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
19 sum(numbers) = 1.0
20 Kahan sum   = 1.0
21 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- `sum` liefert `0.0` für die Summe von `[1e+18, 1, -1e+18]` und auch für `[1e+36, 1e+18, 1, -1e+36, -1e+18]`.
- `KahanSum` und `fsum` können für beide Fälle korrekt `1.0` ausrechnen.
- Wenn wir aber noch größere Zahlen mit einbeziehen und z.B. die Summe über `[1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]` berechnen, dann liefert unsere `KahanSum` `0.0` anstatt des korrekten Ergebnisses `1.0`.
- `fsum` fand das richtige Ergebnis, wohingegen `sum` mit `-1e18` weit daneben liegt.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13 k: KahanSum = KahanSum() # Create our Kahan summation object.
14 for n in numbers:        # Iterate over the numbers...
15    k.add(n)               # ...and let our object add them up.
16 print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17 print(f"Kahan sum = {k.result()}") # our better result
18 print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↗ =====
15 sum(numbers) = -1e+18
16 Kahan sum = 0.0
17 fsum(numbers) = 1.0
18 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
19 sum(numbers) = 1.0
20 Kahan sum = 1.0
21 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- `KahanSum` und `fsum` können für beide Fälle korrekt `1.0` ausrechnen.
- Wenn wir aber noch größere Zahlen mit einbeziehen und z.B. die Summe über `[1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]` berechnen, dann liefert unsere `KahanSum` `0.0` anstatt des korrekten Ergebnisses `1.0`.
- `fsum` findet das richtige Ergebnis, wohingegen `sum` mit `-1e18` weit daneben liegt.
- Zuletzt addieren wir noch `[1, -1e-16, 1e-16, 1e-16]`.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [-1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:       # Iterate over the numbers...
15        k.add(n)           # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↗ =====
15 sum(numbers) = -1e+18
16 Kahan sum   = 0.0
17 fsum(numbers) = 1.0
18 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
19 sum(numbers) = 1.0
20 Kahan sum   = 1.0
21 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- Wenn wir aber noch größere Zahlen mit einbeziehen und z.B. die Summe über [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18] berechnen, dann liefert unsere KahanSum 0.0 anstatt des korrekten Ergebnisses 1.0.
- fsum fand das richtige Ergebnis, wohingegen sum mit -1e18 weit daneben liegt.
- Zuletzt addieren wir noch [1, -1e-16, 1e-16, 1e-16].
- Alle drei Methoden liefern 1.0, wohingegen das genaue Ergebnis $1 + 10^{-16}$ wäre.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:        # Iterate over the numbers...
15        k.add(n)            # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↪ =====
15 sum(numbers) = -1e+18
16 Kahan sum = 0.0
17 fsum(numbers) = 1.0
18 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
19 sum(numbers) = 1.0
20 Kahan sum = 1.0
21 fsum(numbers) = 1.0
```

Kahan Summe: Vergleich

- `fsum` fand das richtige Ergebnis, wohingegen `sum` mit `-1e18` weit daneben liegt.
- Zuletzt addieren wir noch `[1, -1e-16, 1e-16, 1e-16]`.
- Alle drei Methoden liefern `1.0`, wohingegen das genaue Ergebnis $1 + 10^{-16}$ wäre.
- Dies kann allerdings nicht mit dem Datentyp `float` dargestellt werden, also sind die Ergebnisse schon richtig.

```
1 """Examples for using our class :class:`KahanSum`."""
2
3 from math import fsum # An (even more) precise summation algorithm.
4
5 from kahan_sum import KahanSum # Import our new own class.
6
7 # Iterate over four example arrays.
8 for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
9     [-1e36, 1e18, 1, -1e36, -1e18],
10    [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
11    [1, -1e-16, 1e-16, 1e-16]]:
12    print(f"===== numbers = [{', '.join(map(str, numbers))}] =====")
13    k: KahanSum = KahanSum() # Create our Kahan summation object.
14    for n in numbers:       # Iterate over the numbers...
15        k.add(n)           # ...and let our object add them up.
16    print(f"sum(numbers) = {sum(numbers)}") # the normal sum
17    print(f"Kahan sum   = {k.result()}") # our better result
18    print(f"fsum(numbers) = {fsum(numbers)}") # the exact result
```

↓ python3 kahan_user.py ↓

```
1 ===== numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =====
2 sum(numbers) = 1.1111e-12
3 Kahan sum   = 1.1111e-12
4 fsum(numbers) = 1.1111e-12
5 ===== numbers = [1e+18, 1, -1e+18] =====
6 sum(numbers) = 0.0
7 Kahan sum   = 1.0
8 fsum(numbers) = 1.0
9 ===== numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =====
10 sum(numbers) = 0.0
11 Kahan sum   = 1.0
12 fsum(numbers) = 1.0
13 ===== numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]
14     ↪ =====
14 sum(numbers) = -1e+18
15 Kahan sum   = 0.0
16 fsum(numbers) = 1.0
17 ===== numbers = [1, -1e-16, 1e-16, 1e-16] =====
18 sum(numbers) = 1.0
19 Kahan sum   = 1.0
20 fsum(numbers) = 1.0
```

Ergebnis des Vergleichs

- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.



Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.



Ergebnis des Vergleichs

- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen.

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach.

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.

Ergebnis des Vergleichs



- Mit unserem Experiment haben wir herausgefunden, dass `fsum` uns das präziseste Ergebnis gibt.
- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.

Ergebnis des Vergleichs



- Unsere `KahanSum` ist aber definitiv besser als die built-in Funktion `sum`.
- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.



Ergebnis des Vergleichs

- Das bringt uns zu zwei Fragen:
 1. Warum kann unsere `KahanSum` uns nicht immer das genaue Ergebnis liefern?
 2. Wie und warum ist `fsum` besser?
- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlerterminen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.

Ergebnis des Vergleichs



- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlerterminen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*

Ergebnis des Vergleichs



- Die Antwort auf die erste Frage ist sehr einfach:
- Wenn wir genau eine Summationsvariable verwenden, so wie `sum` das tut, dann können präzise bis zu 15 oder 16 Ziffern sein und verliegen alle Ziffern darüber hinaus.
- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlerterminen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.

Ergebnis des Vergleichs



- Wenn wir eine Summationsvariable und einen Fehlerterm `cs` erster Ordnung verwenden, dann bekommen wir im Grunde 15 bis 16 Ziffern dazu, um Zwischenergebnisse genauer darzustellen.
- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir 1 zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen 1.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die 1 geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlertermen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen manages um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).

Ergebnis des Vergleichs



- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlerterminen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen manages um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.

Ergebnis des Vergleichs



- Mit dem Fehlerterm zweiter Ordnung `ccs` können wir dann Zwischenergebnisse mit 45 bis 48 Ziffern genau darstellen.
- Damit können wir `1` zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen `1`.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlertermen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen `managed` um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.

Ergebnis des Vergleichs



- Damit können wir 1 zu `1e36` addieren und dann wieder `1e36` abziehen und bekommen 1.
- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die 1 geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlertermen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen `managed` um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.

Ergebnis des Vergleichs



- Aber wenn wir noch `1e72` dazuaddieren, dann überschreiten wir das Fenster von 48 Ziffern und die `1` geht verloren.
- Mit unserer `KahanSum` mit zwei Fehlertermen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen `managed` um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.
- Es allokiert nicht dynamisch mehr Speicher.

Ergebnis des Vergleichs



- Mit unserer `KahanSum` mit zwei Fehlertermen können wir also Zwischenergebnisse von bis zu 48 Ziffern darstellen.
- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen `managed` um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.
- Es allokiert nicht dynamisch mehr Speicher.
- Deshalb ist die Anzahl der Rechenschritte, die es für jede hinzugefügte Zahl macht, auch konstant.

Ergebnis des Vergleichs



- Die zweite Frage war *Was macht `fsum` anders?*
- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen managed um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.
- Es allokiert nicht dynamisch mehr Speicher.
- Deshalb ist die Anzahl der Rechenschritte, die es für jede hinzugefügte Zahl macht, auch konstant.
- `fsum` macht eine variable Zahl von Rechenschritten, deren Anzahl auf der Länge der internen Datenstruktur basiert.

Ergebnis des Vergleichs



- Eigentlich nicht viel.
- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen manages um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.
- Es allokiert nicht dynamisch mehr Speicher.
- Deshalb ist die Anzahl der Rechenschritte, die es für jede hinzugefügte Zahl macht, auch konstant.
- `fsum` macht eine variable Zahl von Rechenschritten, deren Anzahl auf der Länge der internen Datenstruktur basiert.
- Damit ist `KahanSum` tatsächlich ein sehr schöne Kompromisslösung, die eine höhere Präzision als die normale Addition bietet und aber immer noch die selbe konstante Speicher- und Zeitkomplexität hat.



Ergebnis des Vergleichs

- Es basiert auf dem Algorithmus von Shewchuk^{36,76}, welcher dynamisch eine Liste von Hilfsvariablen manages um eine beweisbar genaue Summe zu berechnen (mit Ergebnis `float`).
- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.
- Es allokiert nicht dynamisch mehr Speicher.
- Deshalb ist die Anzahl der Rechenschritte, die es für jede hinzugefügte Zahl macht, auch konstant.
- `fsum` macht eine variable Zahl von Rechenschritten, deren Anzahl auf der Länge der internen Datenstruktur basiert.
- Damit ist `KahanSum` tatsächlich ein sehr schöne Kompromisslösung, die eine höhere Präzision als die normale Addition bietet und aber immer noch die selbe konstante Speicher- und Zeitkomplexität hat.
- Es ist auch vielseitiger als `fsum`.

Ergebnis des Vergleichs



- Im Grunde ist es eine dynamische Version der Kahan-Summe, welche bei Bedarf mehr Fehlerterme benutzt und diese auch wieder freigibt, wenn sie nicht mehr benötigt werden.
- Das Prinzip ist das gleiche, nur das eine Liste die Variablen `cs` und `ccs` ersetzt.
- Unser `KahanSum` benutzt genau drei Variablen für die ganze Summation.
- Es allokiert nicht dynamisch mehr Speicher.
- Deshalb ist die Anzahl der Rechenschritte, die es für jede hinzugefügte Zahl macht, auch konstant.
- `fsum` macht eine variable Zahl von Rechenschritten, deren Anzahl auf der Länge der internen Datenstruktur basiert.
- Damit ist `KahanSum` tatsächlich ein sehr schöne Kompromisslösung, die eine höhere Präzision als die normale Addition bietet und aber immer noch die selbe konstante Speicher- und Zeitkomplexität hat.
- Es ist auch vielseitiger als `fsum`.
- Wir müssen die aufzuhaltenden Werte nicht in einem `Iterable` haben.



Zusammenfassung



Zusammenfassung

- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen.



Zusammenfassung



- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.



Zusammenfassung

- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen.



Zusammenfassung

- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.



Zusammenfassung

- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.
- Bisher haben wir uns auf den ersten Fall konzentriert.

Zusammenfassung



- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.
- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.

Zusammenfassung



- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.
- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können.



Zusammenfassung

- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.
- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern.

Zusammenfassung



- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.
- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern. Unsere Klasse `Point` ist ein Beispiel dafür.

Zusammenfassung



- Klassen erlauben es uns, zwei Probleme beim Programmieren zu lösen:
 1. Wir können Daten und die Operationen auf den Daten semantisch gruppieren.
 2. Wir können Schnittstellen, also APIs, definieren, die aus mehreren Operationen bestehen. Wir können diese API dann auf verschiedene Arten realisieren und die Operationen jeweils anders implementieren.
- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern. Unsere Klasse `Point` ist ein Beispiel dafür.
 2. Wir können auch gekapselte Klassen erstellen, wobei der Zugriff auf Attribute nur durch die Methoden der Klasse möglich ist.



Zusammenfassung

- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern. Unsere Klasse `Point` ist ein Beispiel dafür.
 2. Wir können auch gekapselte Klassen erstellen, wobei der Zugriff auf Attribute nur durch die Methoden der Klasse möglich ist. Dadurch wird sichergestellt, dass jede Zustandsänderung konsistent und korrekt ist.



Zusammenfassung

- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern. Unsere Klasse `Point` ist ein Beispiel dafür.
 2. Wir können auch gekapselte Klassen erstellen, wobei der Zugriff auf Attribute nur durch die Methoden der Klasse möglich ist. Dadurch wird sichergestellt, dass jede Zustandsänderung konsistent und korrekt ist. Unsere Klasse `KahanSum` ist ein Beispiel dafür.



Zusammenfassung

- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern. Unsere Klasse `Point` ist ein Beispiel dafür.
 2. Wir können auch gekapselte Klassen erstellen, wobei der Zugriff auf Attribute nur durch die Methoden der Klasse möglich ist. Dadurch wird sichergestellt, dass jede Zustandsänderung konsistent und korrekt ist. Unsere Klasse `KahanSum` ist ein Beispiel dafür.
- Die meisten Klassen, die entweder Daten speichern oder irgendein Verhalten realisieren, können auf eine dieser beiden Arten implementiert werden.



Zusammenfassung

- Bisher haben wir uns auf den ersten Fall konzentriert.
- Wir haben zwei Beispiele dafür angeschaut.
- Wir haben auch zwei Prinzipien kennengelernt, wie wir Klassen entwerfen können
 1. Wir können unveränderliche Klassen als Kontainer konstanter Information erstellen, deren Attribute sich nicht ändern. Unsere Klasse `Point` ist ein Beispiel dafür.
 2. Wir können auch gekapselte Klassen erstellen, wobei der Zugriff auf Attribute nur durch die Methoden der Klasse möglich ist. Dadurch wird sichergestellt, dass jede Zustandsänderung konsistent und korrekt ist. Unsere Klasse `KahanSum` ist ein Beispiel dafür.
- Die meisten Klassen, die entweder Daten speichern oder irgendein Verhalten realisieren, können auf eine dieser beiden Arten implementiert werden.
- Es kann natürlich auch Designs geben, die dazwischen liegen, wo einige Attribute `Final` und von außen zugänglich sind und andere Attribute wieder nur durch Methoden geändert werden können.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.
- Die Operation `distance` ist untrennbar mit dieser Datenstruktur verbunden.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.
- Die Operation `distance` ist untrennbar mit dieser Datenstruktur verbunden.
- Wir haben gelernt, dass es oft eine gute Idee ist, Objekte unveränderlich zu machen.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.
- Die Operation `distance` ist untrennbar mit dieser Datenstruktur verbunden.
- Wir haben gelernt, dass es oft eine gute Idee ist, Objekte unveränderlich zu machen.
- Dann werden die Werte der Attribute eines Objektes nur während seiner Initialisierung (durch die `__init__` Methode) gesetzt und ändern sich danach nicht.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.
- Die Operation `distance` ist untrennbar mit dieser Datenstruktur verbunden.
- Wir haben gelernt, dass es oft eine gute Idee ist, Objekte unveränderlich zu machen.
- Dann werden die Werte der Attribute eines Objektes nur während seiner Initialisierung (durch die `__init__` Methode) gesetzt und ändern sich danach nicht.
- Dann kann es niemals Verwirrung über die Attributwerte geben.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.
- Die Operation `distance` ist untrennbar mit dieser Datenstruktur verbunden.
- Wir haben gelernt, dass es oft eine gute Idee ist, Objekte unveränderlich zu machen.
- Dann werden die Werte der Attribute eines Objektes nur während seiner Initialisierung (durch die `__init__` Methode) gesetzt und ändern sich danach nicht.
- Dann kann es niemals Verwirrung über die Attributwerte geben.
- Es kann nie passieren, dass ein Teil unseres Prozesses eine Referenz auf eine `Point`-Variable hat und „denkt“ das diese die Koordinaten $(0, 1)$ hat, aber das anderer Kode die Werte auf etwas anderes geändert hat.

Zusammenfassung: Unveränderliche Objekte



- Instanzen unserer Klasse `Point` speichern ein Paar Koordinaten in der zwei-dimensionalen Euklidischen Ebene.
- Die Operation `distance` ist untrennbar mit dieser Datenstruktur verbunden.
- Wir haben gelernt, dass es oft eine gute Idee ist, Objekte unveränderlich zu machen.
- Dann werden die Werte der Attribute eines Objektes nur während seiner Initialisierung (durch die `__init__` Methode) gesetzt und ändern sich danach nicht.
- Dann kann es niemals Verwirrung über die Attributwerte geben.
- Es kann nie passieren, dass ein Teil unseres Prozesses eine Referenz auf eine `Point`-Variable hat und „denkt“ das diese die Koordinaten $(0, 1)$ hat, aber das anderer Kode die Werte auf etwas anderes geändert hat.
- Das kann nicht passieren, weil sich die Koordinatenwerte einer Instanz von `Point` niemals ändern.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln
encapsulate.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln *encapsulate*.
- Kapselung bedeutet, dass die Attribute eines Objekts nur durch die Methoden des Objekts gelesen oder geändert werden können.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln *encapsulate*.
- Kapselung bedeutet, dass die Attribute eines Objekts nur durch die Methoden des Objekts gelesen oder geändert werden können.
- Unsere Klasse `KahanSum` ist ein Beispiel dafür.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln *encapsulate*.
- Kapselung bedeutet, dass die Attribute eines Objekts nur durch die Methoden des Objekts gelesen oder geändert werden können.
- Unsere Klasse `KahanSum` ist ein Beispiel dafür.
- Die Klasse erlaubt es uns, Zahlen more präzise zu addieren, in dem sie intern Fehlerterme zusätzlich zur normalen Summe unterhält.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln *encapsulate*.
- Kapselung bedeutet, dass die Attribute eines Objekts nur durch die Methoden des Objekts gelesen oder geändert werden können.
- Unsere Klasse `KahanSum` ist ein Beispiel dafür.
- Die Klasse erlaubt es uns, Zahlen more präzise zu addieren, in dem sie intern Fehlerterme zusätzlich zur normalen Summe unterhält.
- Der Benutzer bekommt diese internen Attribute niemals zu sehen und kann sie auch nicht direkt verändern.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln *encapsulate*.
- Kapselung bedeutet, dass die Attribute eines Objekts nur durch die Methoden des Objekts gelesen oder geändert werden können.
- Unsere Klasse `KahanSum` ist ein Beispiel dafür.
- Die Klasse erlaubt es uns, Zahlen more präzise zu addieren, in dem sie intern Fehlerterme zusätzlich zur normalen Summe unterhält.
- Der Benutzer bekommt diese internen Attribute niemals zu sehen und kann sie auch nicht direkt verändern.
- Stattdessen können sie verändert werden, in dem wir neue Zahlen an die Methode `add` liefern.



Zusammenfassung: Gekapselte Klasse

- Wenn sich Attribute ändern müssen, dann ist es oft eine gute Idee, sie zu kapseln *encapsulate*.
- Kapselung bedeutet, dass die Attribute eines Objekts nur durch die Methoden des Objekts gelesen oder geändert werden können.
- Unsere Klasse `KahanSum` ist ein Beispiel dafür.
- Die Klasse erlaubt es uns, Zahlen more präzise zu addieren, in dem sie intern Fehlerterme zusätzlich zur normalen Summe unterhält.
- Der Benutzer bekommt diese internen Attribute niemals zu sehen und kann sie auch nicht direkt verändern.
- Stattdessen können sie verändert werden, in dem wir neue Zahlen an die Methode `add` liefern.
- Über die Methode `result` kann der Nutzer eine konsistente Sicht auf den Zustand der Summe bekommen, ohne durch den internen Zustand verwirrt zu werden.

Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.





Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.

Zusammenfassung: Python is Nachsichtig



- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.



Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.



Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.



Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.



Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.

Zusammenfassung: Python is Nachsichtig



- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.



Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.



Zusammenfassung: Python is Nachsichtig

- Python ist eine sehr nachsichtige Sprache.
- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.



Zusammenfassung: Python is Nachsichtig

- Es ist sehr nachsichtig darin, was erlaubt ist und was nicht.
- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.



Zusammenfassung: Python is Nachsichtig

- Sie erinnern sich daran, dass Type Hints nur Hinweise für Werkzeuge und Programmierer sind.
- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.
- Sie werden dadurch nicht privat.



Zusammenfassung: Python is Nachsichtig

- Der Python-Interpreter setzt sie nicht durch.
- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.
- Sie werden dadurch nicht privat.
- Der Python-Interpreter verändert zwar ihre Namen intern⁹⁰, aber ein cleverer Programmierer kann sie trotzdem auslesen und ändern.



Zusammenfassung: Python is Nachsichtig

- Es ist problemlos möglich, so etwas wie `a: str = 5` auszuführen.
- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.
- Sie werden dadurch nicht privat.
- Der Python-Interpreter verändert zwar ihre Namen intern⁹⁰, aber ein cleverer Programmierer kann sie trotzdem auslesen und ändern.
- Werkzeuge wie Mypy können solches Fehlverhalten finden.

Zusammenfassung: Python is Nachsichtig



- Diese Nachsichtigkeit betrifft auch die beiden Designprinzipien von oben.
- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.
- Sie werden dadurch nicht privat.
- Der Python-Interpreter verändert zwar ihre Namen intern⁹⁰, aber ein cleverer Programmierer kann sie trotzdem auslesen und ändern.
- Werkzeuge wie Mypy können solches Fehlverhalten finden.
- Python gibt die Verantwortung in unsere Hände.

Zusammenfassung: Python is Nachsichtig



- Attribute und Variablen können „unveränderlich“ gemacht werden, in dem wir sie mit dem Type Hint `Final` annotieren.
- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.
- Sie werden dadurch nicht privat.
- Der Python-Interpreter verändert zwar ihre Namen intern⁹⁰, aber ein cleverer Programmierer kann sie trotzdem auslesen und ändern.
- Werkzeuge wie Mypy können solches Fehlverhalten finden.
- Python gibt die Verantwortung in unsere Hände.
- Wir müssen uns an Standards und Programmierregeln halten.

Zusammenfassung: Python is Nachsichtig



- Das ist natürlich wieder nur ein Type Hint, nur eine Information für Programmierer und Werkzeuge.
- Der Python-Interpreter ignoriert das.
- Wie können erst `a: Final[int] = 5` und dann `a = 6` machen, ohne bestraft zu werden.
- Ähnliches gilt wenn wir Attribute durch Namen die mit einem doppelten Unterstrich (`__`) anfangen als privat markieren.
- Sie werden dadurch nicht privat.
- Der Python-Interpreter verändert zwar ihre Namen intern⁹⁰, aber ein cleverer Programmierer kann sie trotzdem auslesen und ändern.
- Werkzeuge wie Mypy können solches Fehlverhalten finden.
- Python gibt die Verantwortung in unsere Hände.
- Wir müssen uns an Standards und Programmierregeln halten.
- Aber Python erzwingt das nicht.



谢谢您们！
Thank you!
Vielen Dank!



References I



- [1] David J. Agans. *Debugging*. New York, NY, USA: AMACOM, Sep. 2002. ISBN: 978-0-8144-2678-4 (siehe S. 294).
- [2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also³ (siehe S. 283, 295).
- [3] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also² (siehe S. 283, 295).
- [4] Ivo Babuška. "Numerical Stability in Mathematical Analysis". In: *World Congress on Information Processing (IFIP'1968)*. Bd. 1: Mathematics, Software. 5. Aug. 1966–10. Aug. 1968, Edinburgh, Scotland, UK. Hrsg. von A.J.H. Morrell. Laxenburg, Austria: International Federation for Information Processing (IFIP). Amsterdam, The Netherlands: North-Holland Publishing Co., 1969, S. 11–23. ISBN: 978-0-7204-2032-6 (siehe S. 65–69, 104–107, 121–123, 178–200, 287).
- [5] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 295, 296).
- [6] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 295).
- [7] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 297).
- [8] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: <https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt> (besucht am 2025-02-05) (siehe S. 297).
- [9] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 294).
- [10] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 295).
- [11] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 295).

References II



- [12] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 294).
- [13] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 295, 296).
- [14] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: <https://cacm.acm.org/research/50-years-of-queries> (besucht am 2025-01-09) (siehe S. 296).
- [15] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 296, 297).
- [16] Edgar Frank „Ted“ Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> (besucht am 2025-01-05) (siehe S. 296).
- [17] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 296).
- [18] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: <https://dataschool.com/how-to-teach-people-sql> (besucht am 2025-02-27) (siehe S. 296).
- [19] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 296).
- [20] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 297).
- [21] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (siehe S. 296).
- [22] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/doctest.html> (besucht am 2024-11-07) (siehe S. 295).

References III



- [23] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 295).
- [24] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: <https://www.researchgate.net/publication/301720080> (besucht am 2024-09-24). Translation of²⁵. (Siehe S. 285).
- [25] 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). See²⁴ for a translation. (Siehe S. 285, 298).
- [26] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 296).
- [27] 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. 298).
- [28] "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. 30–41).
- [29] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys (CSUR)* 23(1):5–48, März 1991. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/103162.103163. URL: <https://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/goldberg-floating-point.pdf> (besucht am 2025-09-03) (siehe S. 65–69, 288).
- [30] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: <https://peps.python.org/pep-0257> (besucht am 2024-07-27) (siehe S. 294).

References IV



- [31] 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. 294).
- [32] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 296).
- [33] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 296).
- [34] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 295).
- [35] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 295, 297).
- [36] Raymond Hettinger. *Binary Floating Point Summation Accurate to Full Precision (Python Recipe)*. Vancouver, BC, Canada: ActiveState Software Inc., 28. März 2005. URL: <http://code.activestate.com/recipes/393090> (besucht am 2024-11-19) (siehe S. 212–234).
- [37] 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. 30–41, 296).
- [38] 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. 296).
- [39] *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. 30–41, 296).

References V



- [40] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1.* International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (besucht am 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Siehe S. 296).
- [41] 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. 298).
- [42] William Kahan. "Pracniques: Further Remarks on Reducing Truncation Errors". *Communications of the ACM (CACM)* 8(1):40, Jan. 1965. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/363707.363723. URL: <https://www.convexoptimization.com/TOOLS/Kahan.pdf> (besucht am 2024-11-18) (siehe S. 65–69, 83–118, 121–123, 178–200, 212–235, 287).
- [43] Andreas Klein. "A Generalized Kahan-Babuška-Summation-Algorithm". *Computing* 76(3-4):279–293, Jan. 2006. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0010-485X. doi:10.1007/s00607-005-0139-x. Based on ^{4,42,56} (siehe S. 104–107).
- [44] Holger Krekel und pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Kap. 2.8, S. 65–69. URL: <https://docs.pytest.org/en/stable/how-to/doctest.html> (besucht am 2024-11-07) (siehe S. 295).
- [45] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: <https://readthedocs.org/projects/pytest/downloads/pdf/latest> (besucht am 2024-11-07) (siehe S. 296).
- [46] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 296).

References VI



- [47] Vincent Lafage. *Revisiting ‘What Every Computer Scientist Should Know About Floating-Point Arithmetic’*. arXiv.org: Computing Research Repository (CoRR) abs/2012.02492. Ithaca, NY, USA: Cornell Universiy Library, 4. Dez. 2020. doi:[10.48550/arXiv.2012.02492](https://doi.org/10.48550/arXiv.2012.02492). URL: <https://arxiv.org/abs/2012.02492> (besucht am 2025-09-03). arXiv:2012.02492v1 [math.NA] 4 Dec 2020, see also²⁹ (siehe S. 65–69).
- [48] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: <https://peps.python.org/pep-0482> (besucht am 2024-10-09) (siehe S. 297).
- [49] 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](https://doi.org/10.1007/978-3-319-13071-2). doi:[10.1007/978-3-319-13072-9](https://doi.org/10.1007/978-3-319-13072-9) (siehe S. 296).
- [50] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: <https://github.com/python/mypy> (besucht am 2024-08-17) (siehe S. 295).
- [51] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. “Client-Server Architecture”. In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: <https://www.britannica.com/technology/client-server-architecture> (besucht am 2025-01-20) (siehe S. 294).
- [52] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: [978-1-0981-7130-8](https://doi.org/10.10981/7130-8) (siehe S. 296).
- [53] MariaDB Server Documentation. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/kb/en/documentation> (besucht am 2025-04-24) (siehe S. 295).
- [54] “Mathematical Functions and Operators”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.3. URL: <https://www.postgresql.org/docs/17/functions-math.html> (besucht am 2025-02-27) (siehe S. 298).
- [55] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: [978-1-55860-456-8](https://doi.org/10.10981/55860-456-8) (siehe S. 296).

References VII



- [56] Arnold Neumaier. "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen". *ZAMM – Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 54(1):39–51, 1974. Weinheim, Baden-Württemberg, Germany: Wiley-VCH GmbH. ISSN: 0044-2267. doi:10.1002/zamm.19740540106. URL: <https://arnold-neumaier.at/scan/01.pdf> (besucht am 2024-11-18) (siehe S. 104–107, 121–123, 178–200, 287).
- [57] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 294).
- [58] 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. 298).
- [59] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 296).
- [60] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (siehe S. 297).
- [61] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (siehe S. 296).
- [62] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: <https://www.researchgate.net/publication/255673967> (besucht am 2025-09-05) (siehe S. 297).
- [63] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 294).
- [64] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: 978-1-4842-7854-3 (siehe S. 296, 297).

References VIII



- [65] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 295).
- [66] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: <https://www.postgresql.org/docs/17/index.html> (besucht am 2025-02-25).
- [67] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 296).
- [68] "Private Name Mangling". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 6.2.1.1. URL: <https://docs.python.org/3/reference/expressions.html#private-name-mangling> (besucht am 2025-09-23) (siehe S. 163–165).
- [69] "Private Variables". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 9.6. URL: <https://docs.python.org/3/tutorial/classes.html#private-variables> (besucht am 2025-09-23) (siehe S. 163–165).
- [70] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 294).
- [71] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 295).
- [72] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021)*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 295).
- [73] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 294).

References IX



- [74] Kristian Rother. *Pro Python Best Practices: Debugging, Testing and Maintenance*. New York, NY, USA: Apress Media, LLC, März 2017. ISBN: 978-1-4842-2241-6 (siehe S. 294).
- [75] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (siehe S. 297).
- [76] Jonathan Richard Shewchuk. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". *Discrete & Computational Geometry* 18(3):305–363, Okt. 1997. London, England, UK: Springer Nature Limited. ISSN: 0179-5376. doi:10.1007/PL00009321. URL: <https://people.eecs.berkeley.edu/~jrs/papers/robustr.pdf> (besucht am 2024-11-19) (siehe S. 212–234).
- [77] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 295).
- [78] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 295).
- [79] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 296).
- [80] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-commands.html> (besucht am 2025-02-25) (siehe S. 296).
- [81] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 291, 296).
- [82] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of⁸¹ (siehe S. 296).
- [83] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 296).

References X



- [84] Alkin Tezuyosal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 295, 296).
- [85] *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/tutorial> (besucht am 2025-04-26).
- [86] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: <https://www.researchgate.net/publication/220094077> (besucht am 2024-10-01) (siehe S. 297).
- [87] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 295).
- [88] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 295, 297).
- [89] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: <https://peps.python.org/pep-0484> (besucht am 2024-08-22) (siehe S. 297).
- [90] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: <https://peps.python.org/pep-0008> (besucht am 2024-07-27) (siehe S. 265–281, 294).
- [91] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 295).
- [92] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2025. URL: <https://thomasweise.github.io/databases> (besucht am 2025-01-05) (siehe S. 294, 296).

References XI



- [93] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2025. URL: <https://thomasweise.github.io/programmingWithPython> (besucht am 2025-01-05) (siehe S. 294–296).
- [94] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: <https://www.ibm.com/think/topics/relational-databases> (besucht am 2025-01-05) (siehe S. 296).
- [95] Ulf Michael „Monty“ Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 295).
- [96] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 294, 296).
- [97] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: <https://www.techtarget.com/searchdatamanagement/definition/database-management-system> (besucht am 2025-01-11) (siehe S. 294).
- [98] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 294).



Glossary (in English) I

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another³¹.

Bash is the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs^{12,57,98}. Learn more at <https://www.gnu.org/software/bash>.

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

client-server architecture is a system design where a central server receives requests from one or multiple clients^{9,51,63,70,73}. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*⁹².

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB⁹⁷.

debugger A debugger is a tool that lets you execute a program step-by-step while observing the current values of variables. This allows you to find errors in the code more easily^{1,74,96}. Learn more about debugging in⁹³.

docstring Docstrings are special string constants in Python that contain documentation for modules or functions³⁰. They must be delimited by `"""..."""`^{30,90}.

Glossary (in English) II



doctest *doctests* are unit tests in the form of small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by These snippets can be executed by modules like `doctest`²² or tools such as `pytest`⁴⁴. Their output is compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes^{78,88}. Learn more at <https://git-scm.com>.

GitHub is a website where software projects can be hosted and managed via the Git VCS^{65,88}. Learn more at <https://github.com>.

IT information technology

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP^{13,35}.

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows^{5,34,77,87,91}. We recommend using it for this course, for software development, and for research. Learn more at <https://www.linux.org>. Its variant Ubuntu is particularly easy to use and install.

MariaDB An open source relational database management system that has forked off from MySQL^{2,3,6,23,53,71}. See <https://mariadb.org> for more information.

Microsoft Windows is a commercial proprietary operating system¹¹. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at <https://www.microsoft.com/windows>.

Mypy is a static type checking tool for Python⁵⁰ that makes use of type hints. Learn more at <https://github.com/python/mypy> and in⁹³.

MySQL An open source relational database management system^{10,23,72,84,95}. MySQL is famous for its use in the LAMP Stack. See <https://www.mysql.com> for more information.

Glossary (in English) III



PostgreSQL An open source object-relational DBMS^{26,59,67,84}. See <https://postgresql.org> for more information.

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

pytest is a framework for writing and executing unit tests in Python^{21,45,61,64,96}. Learn more at <https://pytest.org>.

Python The Python programming language^{38,49,52,93}, i.e., what you will learn about in our book⁹³. Learn more at <https://python.org>.

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other^{16,32,33,79,83,92,94}.

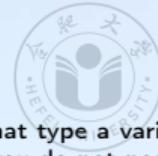
server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carrying the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers¹³ in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“⁴⁶.

significand The significand is the part of a floating point number that stores the digits of the number (in binary representation). In the 64 bit double precision IEEE Standard 754 floating point number layout^{37,39}, the exponent is 52 bits.

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases^{14,17–19,40,55,80–83}. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference⁸⁰.

terminal A terminal is a text-based window where you can enter commands and execute them^{5,15}. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf + , dann Schreiben von cmd, dann Druck auf . Under Ubuntu Linux, + + opens a terminal, which then runs a Bash shell inside.

Glossary (in English) IV



- type hint** are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be^{48,89}. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are basically a piece of documentation.
- Ubuntu** is a variant of the open source operating system Linux^{15,35}. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at <https://ubuntu.com>. If you are in China, you can download it from <https://mirrors.ustc.edu.cn/ubuntu-releases>.
- unit test** Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification^{7,60,62,64,75,86}. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.
- VCS** A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code⁸⁸. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.
- WWW** World Wide Web^{8,20}



Glossary (in English) V

π is the ratio of the circumference U of a circle and its diameter d , i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number^{27,41,58}, which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. In Python, it is provided by the `math` module as constant `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793`⁵⁴.

$i..j$ with $i, j \in \mathbb{Z}$ and $i \leq j$ is the set that contains all integer numbers in the inclusive range from i to j . For example, $5..9$ is equivalent to $\{5, 6, 7, 8, 9\}$

e is Euler's number²⁵, the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number^{27,41}, which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as constant `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045`⁵⁴.

$\text{mean}(A)$ The *arithmetic mean* $\text{mean}(A)$ is an estimate of the expected value of a distribution from which a data sample was, well, sampled. Its is computed on data sample $A = (a_0, a_1, \dots, a_{n-1})$ as the sum of all n elements a_i in the sample data A divided by the total number n of values, i.e., $\text{mean}(A) = \frac{1}{n} \sum_{i=0}^{n-1} a_i$.

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

$\text{var}(A)$ The *variance* of a distribution is the expectation of the squared deviation of the underlying random variable from its mean. The variance $\text{var}(A)$ of a data sample $A = (a_0, a_1, \dots, a_{n-1})$ with n observations can be estimated as $\text{var}(A) = \frac{1}{n-1} \sum_{i=0}^{n-1} (a_i - \text{mean}(A))^2$.

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