合肥大學
HEFEI UNIVERSITY



Programming with Python
28. Zwischenspiel: Unit Tests

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

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

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-11-11

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. Unit Tests
3. Unit Testing with pytest
4. Beispiel
5. Zusammenfassung



Einleitung

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.
- Denn beim Programmieren passieren ja Fehler.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.
- Denn beim Programmieren passieren ja Fehler.
- Sie passieren oft.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.
- Denn beim Programmieren passieren ja Fehler.
- Sie passieren oft.
- Und große Programme haben wahrscheinlicher Fehler und wahrscheinlich auch mehr Fehler als kleine Programme.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.
- Denn beim Programmieren passieren ja Fehler.
- Sie passieren oft.
- Und große Programme haben wahrscheinlicher Fehler und wahrscheinlich auch mehr Fehler als kleine Programme.
- Wir können jetzt beliebig große Programme entwickeln, die gut strukturiert und wartbar sind.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.
- Denn beim Programmieren passieren ja Fehler.
- Sie passieren oft.
- Und große Programme haben wahrscheinlicher Fehler und wahrscheinlich auch mehr Fehler als kleine Programme.
- Wir können jetzt beliebig große Programme entwickeln, die gut strukturiert und wartbar sind.
- Auf der Architekturebene sind wir da nicht mehr eingeschränkt.

Einleitung



- Kode in Funktionen und Module zu strukturieren hat viele Vorteile.
- Wir können Kode wiederverwenden und große Applikationen in kleinere Teile zerlegen, was uns hilft zu verstehen, was der Kode macht.
- Oder besser: machen soll.
- Denn beim Programmieren passieren ja Fehler.
- Sie passieren oft.
- Und große Programme haben wahrscheinlicher Fehler und wahrscheinlich auch mehr Fehler als kleine Programme.
- Wir können jetzt beliebig große Programme entwickeln, die gut strukturiert und wartbar sind.
- Auf der Architekturebene sind wir da nicht mehr eingeschränkt.
- Wir wollen aber nicht nur große, wartbare Software entwickeln ... wir wollen auch, dass diese korrekt und verlässlich ist.

Unit Tests



Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.
- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.
- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.
- Und selbst wenn wir herausfinden würden, dass sich das Programm in ein paar Situationen falsch verhält...

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.
- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.
- Und selbst wenn wir herausfinden würden, dass sich das Programm in ein paar Situationen falsch verhält...
- ...wie finden wir die Stelle in dem gewaltigen Haufen Kode, die den Fehler auslöst?

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.
- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.
- Und selbst wenn wir herausfinden würden, dass sich das Programm in ein paar Situationen falsch verhält...
- ...wie finden wir die Stelle in dem gewaltigen Haufen Kode, die den Fehler auslöst?
- Das Aufteilen von Kode in Funktionen und das Aufteilen von Funktionen in Module hat einen weiteren Vorteil.

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.
- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.
- Und selbst wenn wir herausfinden würden, dass sich das Programm in ein paar Situationen falsch verhält...
- ...wie finden wir die Stelle in dem gewaltigen Haufen Kode, die den Fehler auslöst?
- Das Aufteilen von Kode in Funktionen und das Aufteilen von Funktionen in Module hat einen weiteren Vorteil:
- Wir haben nun kleinere Einheiten von Kode (die Funktionen) die wir einzeln testen können.

Idee: Testen

- Stellen Sie sich vor, wir hätten nur eine große Applikation aus unstrukturiertem Kode zum Lesen und Schreiben von Dateien, für mathematische Berechnungen, usw.
- Es wäre dann nicht einfach herauszufinden, ob sich dieses große Programm genau so verhält, wie es soll.
- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.
- Und selbst wenn wir herausfinden würden, dass sich das Programm in ein paar Situationen falsch verhält...
- ...wie finden wir die Stelle in dem gewaltigen Haufen Kode, die den Fehler auslöst?
- Das Aufteilen von Kode in Funktionen und das Aufteilen von Funktionen in Module hat einen weiteren Vorteil:
- Wir haben nun kleinere Einheiten von Kode (die Funktionen) die wir einzeln testen können.
- Es ist doch viel einfacher, zu prüfen ob unsere Funktionen `factorial` und `sqrt` stimmen, als ein ganzes Programm zu testen.

Idee: Testen

- Es gäbe wahrscheinlich viel zu viele Kombinationen von Eingabedaten und Umgebungen die wir ausprobieren müssten.
- Und selbst wenn wir herausfinden würden, dass sich das Programm in ein paar Situationen falsch verhält...
- ...wie finden wir die Stelle in dem gewaltigen Haufen Kode, die den Fehler auslöst?
- Das Aufteilen von Kode in Funktionen und das Aufteilen von Funktionen in Module hat einen weiteren Vorteil:
- Wir haben nun kleinere Einheiten von Kode (die Funktionen) die wir einzeln testen können.
- Es ist doch viel einfacher, zu prüfen ob unsere Funktionen `factorial` und `sqrt` stimmen, als ein ganzes Programm zu testen.

Definition: Unit Testing

Unit Testing ist eine Software-Test-Technik, bei der einzelne Komponente oder Funktionen einer Applikation in Isolation getestet werden[5,53,55,57,65,76].

Hierarchische Unit Tests

- Diese Idee funktioniert hierarchisch.

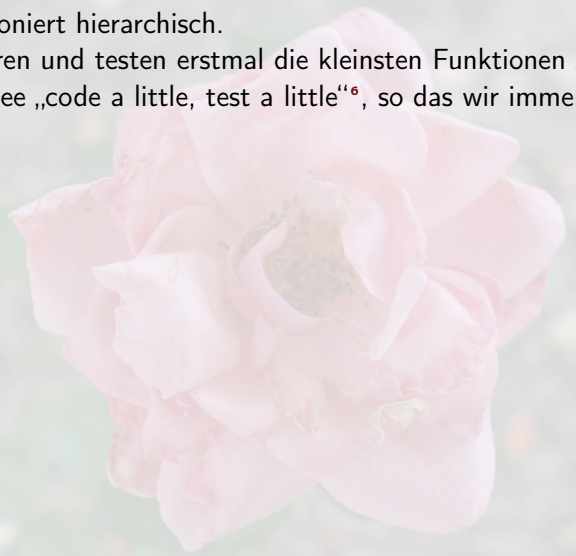


Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.
- Wir arbeiten uns hoch, vom Implementieren und Testen der kleinsten und primitivsten Stücken von Kode auf der niedrigsten Ebene unserers Systems bis hin zu den abstraktesten Funktionen auf der höchsten Ebene.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.
- Wir arbeiten uns hoch, vom Implementieren und Testen der kleinsten und primitivsten Stücken von Kode auf der niedrigsten Ebene unserers Systems bis hin zu den abstraktesten Funktionen auf der höchsten Ebene.
- Erst wenn die Tests auf einer niedrigen Ebene alle durchlaufen gehen wir zur nächsten Ebene.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.
- Wir arbeiten uns hoch, vom Implementieren und Testen der kleinsten und primitivsten Stücken von Kode auf der niedrigsten Ebene unserers Systems bis hin zu den abstraktesten Funktionen auf der höchsten Ebene.
- Erst wenn die Tests auf einer niedrigen Ebene alle durchlaufen gehen wir zur nächsten Ebene.
- Wenn dann ein Test dort schief läuft, dann wissen wir, wo wir zuerst nach dem Fehler suchen müssen.

Hierarchische Unit Tests



- Diese Idee funktioniert hierarchisch.
- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.
- Wir arbeiten uns hoch, vom Implementieren und Testen der kleinsten und primitivsten Stücken von Kode auf der niedrigsten Ebene unserers Systems bis hin zu den abstraktesten Funktionen auf der höchsten Ebene.
- Erst wenn die Tests auf einer niedrigen Ebene alle durchlaufen gehen wir zur nächsten Ebene.
- Wenn dann ein Test dort schief läuft, dann wissen wir, wo wir zuerst nach dem Fehler suchen müssen.
- Natürlich kann es sein, dass die Tests auf der niedrigeren Ebene unzureichend oder sogar falsch waren...

Hierarchische Unit Tests



- Wir implementieren und testen erstmal die kleinsten Funktionen $\mathcal{A}$.
- Wir folgen der Idee „code a little, test a little“[6], so das wir immer unseren Kode verstehen und verifizieren.
- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.
- Wir arbeiten uns hoch, vom Implementieren und Testen der kleinsten und primitivsten Stücken von Kode auf der niedrigsten Ebene unserers Systems bis hin zu den abstraktesten Funktionen auf der höchsten Ebene.
- Erst wenn die Tests auf einer niedrigen Ebene alle durchlaufen gehen wir zur nächsten Ebene.
- Wenn dann ein Test dort schief läuft, dann wissen wir, wo wir zuerst nach dem Fehler suchen müssen.
- Natürlich kann es sein, dass die Tests auf der niedrigeren Ebene unzureichend oder sogar falsch waren...
- Aber wir kennen zumindest den wahrscheinlisten Ort für den Fehler.

Hierarchische Unit Tests

- Dann implementieren und testen wir die Funktionen $\mathcal{B}$, die die Funktionen $\mathcal{A}$ benutzen.
- Dann implementieren und testen wir die Funktionen $\mathcal{C}$, die die Funktionen $\mathcal{B}$ und vielleicht auch die Funktionen $\mathcal{A}$ benutzen.
- Wir arbeiten uns hoch, vom Implementieren und Testen der kleinsten und primitivsten Stücken von Kode auf der niedrigsten Ebene unserers Systems bis hin zu den abstraktesten Funktionen auf der höchsten Ebene.
- Erst wenn die Tests auf einer niedrigen Ebene alle durchlaufen gehen wir zur nächsten Ebene.
- Wenn dann ein Test dort schief läuft, dann wissen wir, wo wir zuerst nach dem Fehler suchen müssen.
- Natürlich kann es sein, dass die Tests auf der niedrigeren Ebene unzureichend oder sogar falsch waren...
- Aber wir kennen zumindest den wahrscheinlisten Ort für den Fehler.
- Das kann die Fehlersuche unglaublich beschleunigen.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.
- Wir haben ihn getestet.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.
- Wir haben ihn getestet.
- Und wir haben den Kode getestet, der diesen Kode verwendet.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.
- Wir haben ihn getestet.
- Und wir haben den Kode getestet, der diesen Kode verwendet.
- Und alle Tests laufen fehlerlos durch.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.
- Wir haben ihn getestet.
- Und wir haben den Kode getestet, der diesen Kode verwendet.
- Und alle Tests laufen fehlerlos durch.
- Klar kann es noch unentdeckte Fehler geben.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.
- Wir haben ihn getestet.
- Und wir haben den Kode getestet, der diesen Kode verwendet.
- Und alle Tests laufen fehlerlos durch.
- Klar kann es noch unentdeckte Fehler geben.
- Aber wir haben zumindest die wahrscheinlichsten Use Cases abgedeckt und sie haben funktioniert.

Vertrauen durch Tests



- Testen kann auch unser Vertrauen in unsere Arbeit und unseren Kode erhöhen.
- Wir haben „diesen Kode“ nicht einfach nur hingeschrieben.
- Wir haben ihn getestet.
- Und wir haben den Kode getestet, der diesen Kode verwendet.
- Und alle Tests laufen fehlerlos durch.
- Klar kann es noch unentdeckte Fehler geben.
- Aber wir haben zumindest die wahrscheinlichsten Use Cases abgedeckt und sie haben funktioniert.
- Das ist hier kein pfuschig dahingeklatschtes Zeugs... ...das ist das Ergebnis von vernünftiger Ingenieursarbeit.

Technischer Ansatz

- Aber wie funktioniert das auf einer technischen Ebene?

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.

Technischer Ansatz



- Aber wie funktioniert das auf einer technischen Ebene?
- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.

Technischer Ansatz



- Ein Unit Test ist im Grunde ein separates Programm.
- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.

Technischer Ansatz



- Das Programm lädt die Funktion, die es testen soll.
- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.
- Sie haben keine echte Funktion außer dem Testen und vielleicht dem expliziten Dokumentieren von erwartetem Verhalten.

Technischer Ansatz



- Es ruft die Funktion auf und vergleicht ihr Verhalten mit dem erwarteten Verhalten in einem spezifizierten Szenario.
- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.
- Sie haben keine echte Funktion außer dem Testen und vielleicht dem expliziten Dokumentieren von erwartetem Verhalten.
- Genaugenommen haben wir sowas sogar fast schon gemacht.

Technischer Ansatz



- Wenn beide übereinstimmen, dann war der Test erfolgreich.
- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.
- Sie haben keine echte Funktion außer dem Testen und vielleicht dem expliziten Dokumentieren von erwartetem Verhalten.
- Genaugenommen haben wir sowas sogar fast schon gemacht.
- Erinnern Sie sich, dass wir die Ergebnisse unserer eigenen `sqrt`-Funktion mit denen der `sqrt`-Funktion aus dem Modul `math` verglichen haben?

Technischer Ansatz



- Ist das beobachtete Verhalten ungleich dem erwarteten Verhalten, dann schlägt der Test fehl.
- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.
- Sie haben keine echte Funktion außer dem Testen und vielleicht dem expliziten Dokumentieren von erwartetem Verhalten.
- Genaugenommen haben wir sowas sogar fast schon gemacht.
- Erinnern Sie sich, dass wir die Ergebnisse unserer eigenen `sqrt`-Funktion mit denen der `sqrt`-Funktion aus dem Modul `math` verglichen haben?
- Wir haben die Werte nebeneinander ausgegeben.

Technischer Ansatz



- Hier kann das „Szenario“ ein Satz Parameterwerte (Argumente) für eine Funktion sein und das „Verhalten“ kann dann den Rückgabewerten entsprechen.
- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.
- Sie haben keine echte Funktion außer dem Testen und vielleicht dem expliziten Dokumentieren von erwartetem Verhalten.
- Genaugenommen haben wir sowas sogar fast schon gemacht.
- Erinnern Sie sich, dass wir die Ergebnisse unserer eigenen `sqrt`-Funktion mit denen der `sqrt`-Funktion aus dem Modul `math` verglichen haben?
- Wir haben die Werte nebeneinander ausgegeben.
- Wir hätten sie auch programmatisch vergleichen können.

Technischer Ansatz



- Wir wissen, welche Ergebnisse wir für die Parameterwerte erwarten und können dann die zurückgelieferten Werte damit vergleichen.
- Wir können beliebig viele solche Szenarios, also Test Cases, für eine Funktion entwickeln.
- Die Tests sind dann normalerweise nicht Teil des Produkts, des Programms, was wir den Benutzern geben.
- Sie haben keine echte Funktion außer dem Testen und vielleicht dem expliziten Dokumentieren von erwartetem Verhalten.
- Genaugenommen haben wir sowas sogar fast schon gemacht.
- Erinnern Sie sich, dass wir die Ergebnisse unserer eigenen `sqrt`-Funktion mit denen der `sqrt`-Funktion aus dem Modul `math` verglichen haben?
- Wir haben die Werte nebeneinander ausgegeben.
- Wir hätten sie auch programmatisch vergleichen können.
- Wir hätten „Erfolg“ melden können, wenn `isclose` für sie wahr ist und „Test Fehler!“ andernfalls. . .

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.
- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.
- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.
- Der konsequente Einsatz von Unit Tests führt zu modularem, sauberen Programmieren und erzwingt geradezu, komplexe Programme in einzelne, testbare Komponenten zu zerlegen.

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.
- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.
- Der konsequente Einsatz von Unit Tests führt zu modularem, sauberen Programmieren und erzwingt geradezu, komplexe Programme in einzelne, testbare Komponenten zu zerlegen.
- Unit Tests sind besonders nützlich, wenn eine Applikation über eine lange Zeit entwickelt, gewartet, und weiterentwickelt wird.

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.
- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.
- Der konsequente Einsatz von Unit Tests führt zu modularem, sauberen Programmieren und erzwingt geradezu, komplexe Programme in einzelne, testbare Komponenten zu zerlegen.
- Unit Tests sind besonders nützlich, wenn eine Applikation über eine lange Zeit entwickelt, gewartet, und weiterentwickelt wird.
- Es ist wichtig, alle Unit Tests aufzuheben und nochmal durchlaufen zu lassen, wann immer wir unser Programm verändern.

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.
- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.
- Der konsequente Einsatz von Unit Tests führt zu modularem, sauberen Programmieren und erzwingt geradezu, komplexe Programme in einzelne, testbare Komponenten zu zerlegen.
- Unit Tests sind besonders nützlich, wenn eine Applikation über eine lange Zeit entwickelt, gewartet, und weiterentwickelt wird.
- Es ist wichtig, alle Unit Tests aufzuheben und nochmal durchlaufen zu lassen, wann immer wir unser Programm verändern.
- So können wir feststellen, ob eine Änderung an einem älteren Stück Kode einen Test fehlschlagen lässt.

Unit Tests nochmal

- Das Ziel von Unit Tests ist es, sicherzustellen, das die Software wie erwartet funktioniert.
- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.
- Der konsequente Einsatz von Unit Tests führt zu modularem, sauberen Programmieren und erzwingt geradezu, komplexe Programme in einzelne, testbare Komponenten zu zerlegen.
- Unit Tests sind besonders nützlich, wenn eine Applikation über eine lange Zeit entwickelt, gewartet, und weiterentwickelt wird.
- Es ist wichtig, alle Unit Tests aufzuheben und nochmal durchlaufen zu lassen, wann immer wir unser Programm verändern.
- So können wir feststellen, ob eine Änderung an einem älteren Stück Kode einen Test fehlschlagen lässt.
- Damit können wir wiederum feststellen, ob eine Änderung an einem Modul unerwartete Konsequenzen in anderem Kode hat und vielleicht das Verhalten unseres Programm auf unerwünschte Art ändert.

Unit Tests nochmal

- Weil Tests gleichzeitig oder sogar vor den Funktionen entwickelt werden, die sie testen, können wir früh potentielle Fehler im Entwicklungsprozess finden.
- Der konsequente Einsatz von Unit Tests führt zu modularem, sauberen Programmieren und erzwingt geradezu, komplexe Programme in einzelne, testbare Komponenten zu zerlegen.
- Unit Tests sind besonders nützlich, wenn eine Applikation über eine lange Zeit entwickelt, gewartet, und weiterentwickelt wird.
- Es ist wichtig, alle Unit Tests aufzuheben und nochmal durchlaufen zu lassen, wann immer wir unser Programm verändern.
- So können wir feststellen, ob eine Änderung an einem älteren Stück Kode einen Test fehlschlagen lässt.
- Damit können wir wiederum feststellen, ob eine Änderung an einem Modul unerwartete Konsequenzen in anderem Kode hat und vielleicht das Verhalten unseres Programm auf unerwünschte Art ändert.
- Besonders mit steigender Automatisierung durch Continuous Integration (CI) sind Unit Tests in der Softwareentwicklung immer wichtiger geworden[65,76,85] und ein integraler Bestandteil der Softwareentwicklung mit Python geworden[22,54,57].

Unit Testing with pytest

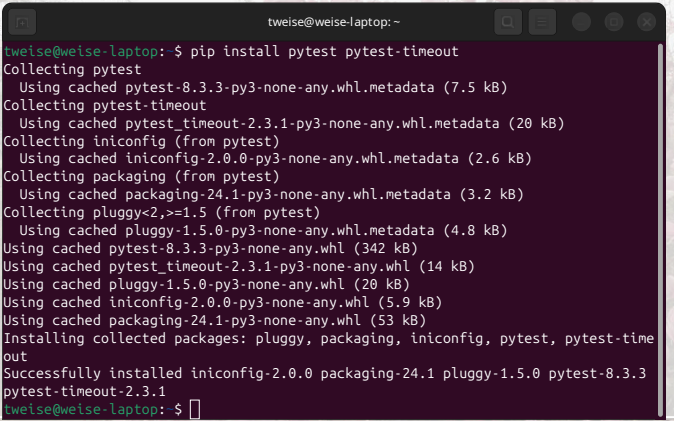


- Nun wollen wir unseren Kode testen.

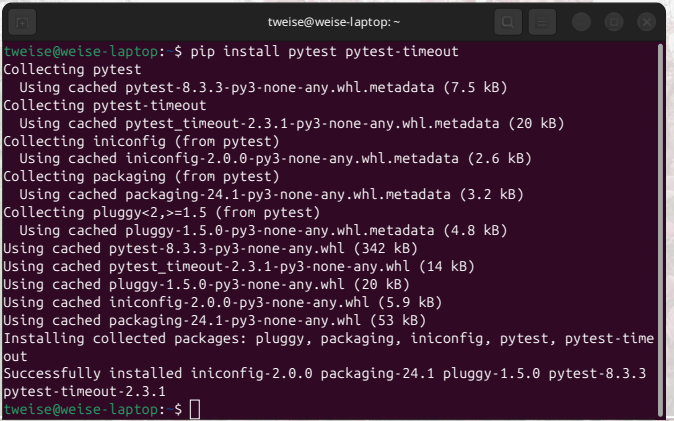
- Nun wollen wir unseren Kode testen.
- Wir werden dafür pytest verwenden, ein Python Framework für Unit Tests[38].



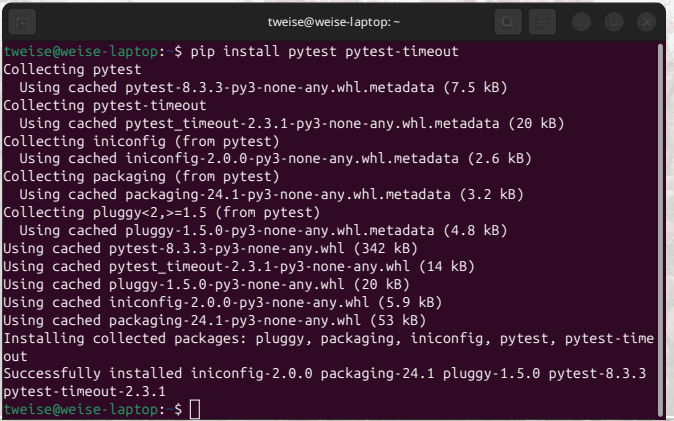
- Nun wollen wir unseren Kode testen.
- Wir werden dafür pytest verwenden, ein Python Framework für Unit Tests[38].
- Wir installieren es, in dem wir ein Terminal öffnen, in dem wir unter Ubuntu Linux Ctrl + Alt + T drücken, oder unter Microsoft Windows via Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.



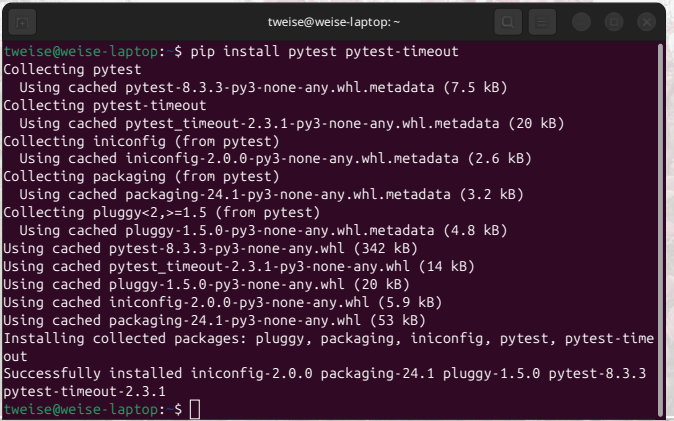
- Wir werden dafür pytest verwenden, ein Python Framework für Unit Tests[38].
- Wir installieren es, in dem wir ein Terminal öffnen, in dem wir unter Ubuntu Linux Ctrl + Alt + T drücken, oder unter Microsoft Windows via Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.
- Dann geben wir `pip install pytest pytest-timeout` ein und drücken ↵.



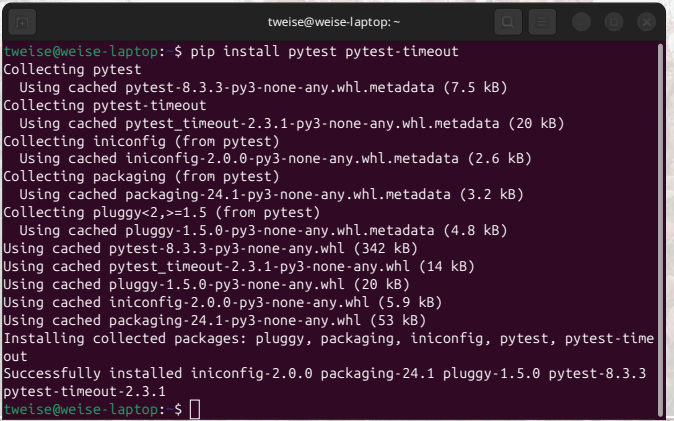
- Wir installieren es, in dem wir ein Terminal öffnen, in dem wir unter Ubuntu Linux Ctrl + Alt + T drücken, oder unter Microsoft Windows via Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.
- Dann geben wir `pip install pytest pytest-timeout` ein und drücken ↵.
- Normalerwise würden wir das in einem virtuellen Environment machen, aber dazu kommen wir irgendwann später.



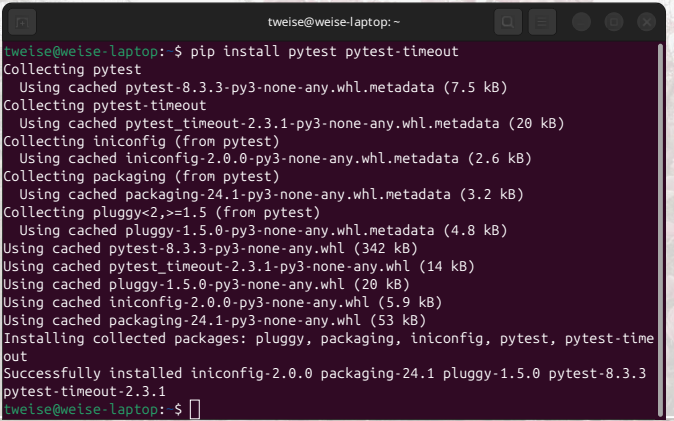
- Dann geben wir `pip install pytest pytest-timeout` ein und drücken ↵.
- Normalerwise würden wir das in einmem virtuellen Environment machen, aber dazu kommen wir irgendwann später.
- Wichtig ist, dass wir diesmal *zwei* Python Pakete installieren.



- Normalerwise würden wir das in einmem virtuellen Environment machen, aber dazu kommen wir irgendwann später.
- Wichtig ist, dass wir diesmal *zwei* Python Pakete installieren
  1. Das Paket `pytest` bietet uns die grundlegende Test-Funktionalität.



- Wichtig ist, dass wir diesmal *zwei* Python Pakete installieren
  1. Das Paket `pytest` bietet uns die grundlegende Test-Funktionalität.
  2. Das Paket `pytest-timeout` erlaubt es uns, die Laufzeit von Tests zu begrenzen, was später wichtig werden wird.



- Wir werden dafür pytest verwenden, ein Python Framework für Unit Tests[38].
- Dann geben wir `pip install pytest pytest-timeout` ein und drücken ↵.
- Wichtig ist, dass wir diesmal *zwei* Python Pakete installieren
  1. Das Paket `pytest` bietet uns die grundlegende Test-Funktionalität.
  2. Das Paket `pytest-timeout` erlaubt es uns, die Laufzeit von Tests zu begrenzen, was später wichtig werden wird.

Nützliches Werkzeug

pytest ist ein Python Framework zum Schreiben und Ausführen von Unit Tests[38].

- Wir werden dafür pytest verwenden, ein Python Framework für Unit Tests[38].
- Dann geben wir `pip install pytest pytest-timeout` ein und drücken ↵.
- Wichtig ist, dass wir diesmal *zwei* Python Pakete installieren
  1. Das Paket `pytest` bietet uns die grundlegende Test-Funktionalität.
  2. Das Paket `pytest-timeout` erlaubt es uns, die Laufzeit von Tests zu begrenzen, was später wichtig werden wird.

Nützliches Werkzeug

pytest ist ein Python Framework zum Schreiben und Ausführen von Unit Tests[38]. Es kann via `pip install pytest pytest-timeout` installiert werden.

- Wir werden dafür pytest verwenden, ein Python Framework für Unit Tests[38].
- Dann geben wir `pip install pytest pytest-timeout` ein und drücken ↵.
- Wichtig ist, dass wir diesmal *zwei* Python Pakete installieren
  1. Das Paket `pytest` bietet uns die grundlegende Test-Funktionalität.
  2. Das Paket `pytest-timeout` erlaubt es uns, die Laufzeit von Tests zu begrenzen, was später wichtig werden wird.

Nützliches Werkzeug

pytest ist ein Python Framework zum Schreiben und Ausführen von Unit Tests[38]. Es kann via `pip install pytest pytest-timeout` installiert werden. Sie können dann pytest mit dem Kommando `pytest --timeout=toInS file(s)` ausführen, wobei `toInS` mit einem vernünftigen Timeout in Sekunden und `file(s)` mit den Dateien mit den Testcases ersetzt werden.

Beispiel

Testen wir unser Eigenes Modul

- In der vorigen Einheit haben wir unser eigenes Modul `my_math.py` erstellt.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Testen wir unser Eigenes Modul

- In der vorigen Einheit haben wir unser eigenes Modul `my_math.py` erstellt.
- Das wollen wir jetzt testen.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Testen wir unser Eigenes Modul

- In der vorigen Einheit haben wir unser eigenes Modul `my_math.py` erstellt.
- Das wollen wir jetzt testen.
- Wir erstellen also eine neue Datei mit den Tests und nennen diese z. B. `test_my_math.py`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)² should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10² = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- In der vorigen Einheit haben wir unser eigenes Modul `my_math.py` erstellt.
- Das wollen wir jetzt testen.
- Wir erstellen also eine neue Datei mit den Tests und nennen diese z. B. `test_my_math.py`.
- Der Test-Kode ist wieder in Form von Funktionen definiert.

```
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- In der vorigen Einheit haben wir unser eigenes Modul `my_math.py` erstellt.
- Das wollen wir jetzt testen.
- Wir erstellen also eine neue Datei mit den Tests und nennen diese z. B. `test_my_math.py`.
- Der Test-Kode ist wieder in Form von Funktionen definiert.
- Der Name jeder Test-Funktion muss mit `test_` anfangen.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- Das wollen wir jetzt testen.
- Wir erstellen also eine neue Datei mit den Tests und nennen diese z. B. `test_my_math.py`.
- Der Test-Kode ist wieder in Form von Funktionen definiert.
- Der Name jeder Test-Funktion muss mit `test_` anfangen.
- In unserem Modul `my_math` hatten wir zwei Funktionen, `factorial` und `sqrt`.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Testen wir unser Eigenes Modul

- Wir erstellen also eine neue Datei mit den Tests und nennen diese z. B. `test_my_math.py`.
- Der Test-Kode ist wieder in Form von Funktionen definiert.
- Der Name jeder Test-Funktion muss mit `test_` anfangen.
- In unserem Modul `my_math` hatten wir zwei Funktionen, `factorial` und `sqrt`.
- Oben in unserem Test-Modul `test_my_math` importieren wir diese.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- Der Test-Kode ist wieder in Form von Funktionen definiert.
- Der Name jeder Test-Funktion muss mit `test_` anfangen.
- In unserem Modul `my_math` hatten wir zwei Funktionen, `factorial` und `sqrt`.
- Oben in unserem Test-Modul `test_my_math` importieren wir diese.
- Wir erstellen zwei Test-Funktionen und nennen diese `test_factorial` und `test_sqrt`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- Der Name jeder Test-Funktion muss mit `test_` anfangen.
- In unserem Modul `my_math` hatten wir zwei Funktionen, `factorial` und `sqrt`.
- Oben in unserem Test-Modul `test_my_math` importieren wir diese.
- Wir erstellen zwei Test-Funktionen und nennen diese `test_factorial` und `test_sqrt`.
- Diese Funktionen haben weder Parameter noch Rückgabewerte.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- In unserem Modul `my_math` hatten wir zwei Funktionen, `factorial` und `sqrt`.
- Oben in unserem Test-Modul `test_my_math` importieren wir diese.
- Wir erstellen zwei Test-Funktionen und nennen diese `test_factorial` und `test_sqrt`.
- Diese Funktionen haben weder Parameter noch Rückgabewerte.
- Tests werden oft als sogenannte *assertions* definitiert.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Testen wir unser Eigenes Modul

- Oben in unserem Test-Modul `test_my_math` importieren wir diese.
- Wir erstellen zwei Test-Funktionen und nennen diese `test_factorial` und `test_sqrt`.
- Diese Funktionen haben weder Parameter noch Rückgabewerte.
- Tests werden oft als sogenannte *assertions* definitiert.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir erstellen zwei Test-Funktionen und nennen diese `test_factorial` und `test_sqrt`.
- Diese Funktionen haben weder Parameter noch Rückgabewerte.
- Tests werden oft als sogenannte *assertions* definitiert.
- Eine Assertion beginnt dem Schlüsselwort `assert`, welchem ein beliebiger Boolescher Ausdruck folgt.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Diese Funktionen haben weder Parameter noch Rückgabewerte.
- Tests werden oft als sogenannte *assertions* definitiert.
- Eine Assertion beginnt dem Schlüsselwort `assert`, welchem ein beliebiger Boolescher Ausdruck folgt.
- Wenn der Ausdruck `True` ergibt, passiert gar nichts.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Tests werden oft als sogenannte *assertions* definitiert.
- Eine Assertion beginnt dem Schlüsselwort `assert`, welchem ein beliebiger Boolescher Ausdruck folgt.
- Wenn der Ausdruck `True` ergibt, passiert gar nichts.
- Ist er jedoch `False`, dann wird ein `AssertionError` ausgelöst.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Eine Assertion beginnt dem Schlüsselwort `assert`, welchem ein beliebiger Boolescher Ausdruck folgt.
- Wenn der Ausdruck `True` ergibt, passiert gar nichts.
- Ist er jedoch `False`, dann wird ein `AssertionError` ausgelöst.
- (Wir lernen später über mehr über solche sogenannten `Exceptions`...)

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Eine Assertion beginnt dem Schlüsselwort `assert`, welchem ein beliebiger Boolescher Ausdruck folgt.
- Wenn der Ausdruck `True` ergibt, passiert gar nichts.
- Ist er jedoch `False`, dann wird ein `AssertionError` ausgelöst.
- (Wir lernen später über mehr über solche sogenannten `Exceptions`...)
- So oder so, dieser Fehler lässt den Test sofort fehlschlagen und beendet ihn auch.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wenn der Ausdruck `True` ergibt, passiert gar nichts.
- Ist er jedoch `False`, dann wird ein `AssertionError` ausgelöst.
- (Wir lernen später über mehr über solche sogenannten `Exceptions`...)
- So oder so, dieser Fehler lässt den Test sofort fehlschlagen und beendet ihn auch.
- Wenn eine Test-Funktion durchläuft, ohne irgendeine `Exception` auszulösen, dann war der Test erfolgreich.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Ist er jedoch `False`, dann wird ein `AssertionError` ausgelöst.
- (Wir lernen später über mehr über solche sogenannten `Exceptions`...)
- So oder so, dieser Fehler lässt den Test sofort fehlschlagen und beendet ihn auch.
- Wenn eine Test-Funktion durchläuft, ohne irgendeine `Exception` auszulösen, dann war der Test erfolgreich.
- Wenn er irgendwo eine `Exception` auslöst, dann ist der Test fehlgeschlagen.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- (Wir lernen später über mehr über solche sogenannten `Exceptions`...)
- So oder so, dieser Fehler lässt den Test sofort fehlschlagen und beendet ihn auch.
- Wenn eine Test-Funktion durchläuft, ohne irgendeine `Exception` auszulösen, dann war der Test erfolgreich.
- Wenn er irgendwo eine `Exception` auslöst, dann ist der Test fehlgeschlagen.
- Wir fangen mit der Funktion `test_factorial` zum Testen von `factorial` an.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- So oder so, dieser Fehler lässt den Test sofort fehlschlagen und beendet ihn auch.
- Wenn eine Test-Funktion durchläuft, ohne irgendeine `Exception` auszulösen, dann war der Test erfolgreich.
- Wenn er irgendwo eine `Exception` auslöst, dann ist der Test fehlgeschlagen.
- Wir fangen mit der Funktion `test_factorial` zum Testen von `factorial` an.
- Wir wissen, dass $0! = 1$ gilt und daher muss `factorial(0)== 1` gelten.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wenn eine Test-Funktion durchläuft, ohne irgendeine `Exception` auszulösen, dann war der Test erfolgreich.
- Wenn er irgendwo eine `Exception` auslöst, dann ist der Test fehlgeschlagen.
- Wir fangen mit der Funktion `test_factorial` zum Testen von `factorial` an.
- Wir wissen, dass $0! = 1$ gilt und daher muss `factorial(0)== 1` gelten.
- Wir schreiben also erstmal `assert factorial(0)== 1` hin.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wenn er irgendwo eine `Exception` auslöst, dann ist der Test fehlgeschlagen.
- Wir fangen mit der Funktion `test_factorial` zum Testen von `factorial` an.
- Wir wissen, dass $0! = 1$ gilt und daher muss `factorial(0)== 1` gelten.
- Wir schreiben also erstmal `assert factorial(0)== 1` hin.
- Der Boolesche Ausdruck hier ist `factorial(0)== 1` und der ist nur wahr, wenn `factorial(0)` wirklich `1` ergibt.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir fangen mit der Funktion `test_factorial` zum Testen von `factorial` an.
- Wir wissen, dass $0! = 1$ gilt und daher muss `factorial(0)== 1` gelten.
- Wir schreiben also erstmal `assert factorial(0)== 1` hin.
- Der Boolesche Ausdruck hier ist `factorial(0)== 1` und der ist nur wahr, wenn `factorial(0)` wirklich `1` ergibt.
- Wir berechnen noch ein paar andere Fakultäten von Hand.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir wissen, dass $0! = 1$ gilt und daher muss `factorial(0)== 1` gelten.
- Wir schreiben also erstmal `assert factorial(0)== 1` hin.
- Der Boolesche Ausdruck hier ist `factorial(0)== 1` und der ist nur wahr, wenn `factorial(0)` wirklich `1` ergibt.
- Wir berechnen noch ein paar andere Fakultäten von Hand.
- Damit können wir dann ähnliche Test Cases für 1!, 2!, und 3! definieren.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir schreiben also erstmal `assert factorial(0)== 1` hin.
- Der Boolesche Ausdruck hier ist `factorial(0)== 1` und der ist nur wahr, wenn `factorial(0)` wirklich `1` ergibt.
- Wir berechnen noch ein paar andere Fakultäten von Hand.
- Damit können wir dann ähnliche Test Cases für 1!, 2!, und 3! definieren.
- Natürlich können wir keine vollständige Liste aller möglicher Eingabewerte abarbeiten.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Der Boolesche Ausdruck hier ist `factorial(0)== 1` und der ist nur wahr, wenn `factorial(0)` wirklich `1` ergibt.
- Wir berechnen noch ein paar andere Fakultäten von Hand.
- Damit können wir dann ähnliche Test Cases für 1!, 2!, und 3! definieren.
- Natürlich können wir keine vollständige Liste aller möglicher Eingabewerte abarbeiten.
- Das sind aber die vier kleinsten Werte, für die die Fakultät definiert ist.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir berechnen noch ein paar andere Fakultäten von Hand.
- Damit können wir dann ähnliche Test Cases für 1!, 2!, und 3! definieren.
- Natürlich können wir keine vollständige Liste aller möglicher Eingabewerte abarbeiten.
- Das sind aber die vier kleinsten Werte, für die die Fakultät definiert ist.
- Also testen wir sie.
- Dann testen wir auch noch einen Wert irgendwo in der Mitte, sagen wir 12!.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Damit können wir dann ähnliche Test Cases für 1!, 2!, und 3! definieren.
- Natürlich können wir keine vollständige Liste aller möglicher Eingabewerte abarbeiten.
- Das sind aber die vier kleinsten Werte, für die die Fakultät definiert ist.
- Also testen wir sie.
- Dann testen wir auch noch einen Wert irgendwo in der Mitte, sagen wir 12!.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Natürlich können wir keine vollständige Liste aller möglicher Eingabewerte abarbeiten.
- Das sind aber die vier kleinsten Werte, für die die Fakultät definiert ist.
- Also testen wir sie.
- Dann testen wir auch noch einen Wert irgendwo in der Mitte, sagen wir 12!.
- Und wir müssen einen großen Wert testen, sagen wir 30!.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Das sind aber die vier kleinsten Werte, für die die Fakultät definiert ist.
- Also testen wir sie.
- Dann testen wir auch noch einen Wert irgendwo in der Mitte, sagen wir 12!.
- Und wir müssen einen großen Wert testen, sagen wir 30!.
- Es ist *sehr sehr* wichtig, dass wir nicht unsere eigene `factorial` Funktion benutzen, um die erwarteten Ausgabewerte zu berechnen ... sonst wären die Tests ja sinnlos.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Also testen wir sie.
- Dann testen wir auch noch einen Wert irgendwo in der Mitte, sagen wir 12!.
- Und wir müssen einen großen Wert testen, sagen wir 30!.
- Es ist *sehr sehr* wichtig, dass wir nicht unsere eigene `factorial` Funktion benutzen, um die erwarteten Ausgabewerte zu berechnen … sonst wären die Tests ja sinnlos.
- Hier haben wir ein anderes Werzeug benutzt, nämlich einen Online-Rechner mit beliebiger Genauigkeit.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Es ist *sehr sehr* wichtig, dass wir nicht unsere eigene `factorial` Funktion benutzen, um die erwarteten Ausgabewerte zu berechnen … sonst wären die Tests ja sinnlos.
- Hier haben wir ein anderes Werzeug benutzt, nämlich einen Online-Rechner mit beliebiger Genauigkeit.
- Das Ergebnis von 30! ist mehr als $265 * 10^{30}$, was über den Wertebereich eines 64 bit Integers hinausgeht (und uns darum wieder beweist, dass Ganzzahlen in Python 3 eine unbegrenze Größe haben können).

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Hier haben wir ein anderes Werzeug benutzt, nämlich einen Online-Rechner mit beliebiger Genauigkeit.
- Das Ergebnis von 30! ist mehr als $265 * 10^{30}$, was über den Wertebereich eines 64 bit Integers hinausgeht (und uns darum wieder beweist, dass Ganzzahlen in Python 3 eine unbegrenze Größe haben können).
- Mit diesem Test Cast können wir also prüfen, ob unsere `factorial`-Funktion auch für große Zahlen funktioniert.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Das Ergebnis von 30! ist mehr als $265 * 10^{30}$, was über den Wertebereich eines 64 bit Integers hinausgeht (und uns darum wieder beweist, dass Ganzzahlen in Python 3 eine unbegrenze Größe haben können).
- Mit diesem Test Cast können wir also prüfen, ob unsere `factorial`-Funktion auch für große Zahlen funktioniert.
- Wenn unsere `factorial`-Funktion alle diese Tests besteht, dann können wir relativ sicher sein, dass sie korrekt implementiert wurde.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Mit diesem Test Cast können wir also prüfen, ob unsere `factorial`-Funktion auch für große Zahlen funktioniert.
- Wenn unsere `factorial`-Funktion alle diese Tests besteht, dann können wir relativ sicher sein, dass sie korrekt implementiert wurde.
- Sie könnte natürlich immer noch falsch sein.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Mit diesem Test Cast können wir also prüfen, ob unsere `factorial`-Funktion auch für große Zahlen funktioniert.
- Wenn unsere `factorial`-Funktion alle diese Tests besteht, dann können wir relativ sicher sein, dass sie korrekt implementiert wurde.
- Sie könnte natürlich immer noch falsch sein.
- Vielleicht liefert sie ja ein falsches Ergebnis für 4!.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Mit diesem Test Cast können wir also prüfen, ob unsere `factorial`-Funktion auch für große Zahlen funktioniert.
- Wenn unsere `factorial`-Funktion alle diese Tests besteht, dann können wir relativ sicher sein, dass sie korrekt implementiert wurde.
- Sie könnte natürlich immer noch falsch sein.
- Vielleicht liefert sie ja ein falsches Ergebnis für 4!.
- Es erscheint zumindest unwahrscheinlich, dass 3! und 12! „funktionieren," aber nicht 4!.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)² should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10² = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wenn unsere `factorial`-Funktion alle diese Tests besteht, dann können wir relativ sicher sein, dass sie korrekt implementiert wurde.
- Sie könnte natürlich immer noch falsch sein.
- Vielleicht liefert sie ja ein falsches Ergebnis für 4!.
- Es erscheint zumindest unwahrscheinlich, dass 3! und 12! „funktionieren," aber nicht 4!.
- Für die `sqrt`-Funktion erstellen wir eine ähnliche Testfunktion und nennen sie `test_sqrt`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)² should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10² = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Sie könnte natürlich immer noch falsch sein.
- Vielleicht liefert sie ja ein falsches Ergebnis für 4!.
- Es erscheint zumindest unwahrscheinlich, dass 3! und 12! „funktionieren," aber nicht 4!.
- Für die `sqrt`-Funktion erstellen wir eine ähnliche Testfunktion und nennen sie `test_sqrt`.
- Vernünftige Test Cases sind `sqrt(0.0)== 0.0`, `sqrt(1.0)== 1.0`, und `sqrt(4.0)== 2.0`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Vielleicht liefert sie ja ein falsches Ergebnis für 4!.
- Es erscheint zumindest unwahrscheinlich, dass 3! und 12! „funktionieren," aber nicht 4!.
- Für die `sqrt`-Funktion erstellen wir eine ähnliche Testfunktion und nennen sie `test_sqrt`.
- Vernünftige Test Cases sind `sqrt(0.0)== 0.0`, `sqrt(1.0)== 1.0`, und `sqrt(4.0)== 2.0`.
- Wir würden auch erwarten dass `sqrt(x)* sqrt(x)== x` für verscheidene `x`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Es erscheint zumindest unwahrscheinlich, dass 3! und 12! „funktionieren,“ aber nicht 4!.
- Für die `sqrt`-Funktion erstellen wir eine ähnliche Testfunktion und nennen sie `test_sqrt`.
- Vernünftige Test Cases sind `sqrt(0.0)== 0.0`, `sqrt(1.0)== 1.0`, und `sqrt(4.0)== 2.0`.
- Wir würden auch erwarten dass `sqrt(x)* sqrt(x)== x` für verscheidene `x`.
- Wir müssen aber die begrenzte Präzision von Fließkommazahlen beachten.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Vernünftige Test Cases sind `sqrt(0.0)== 0.0`, `sqrt(1.0)== 1.0`, und `sqrt(4.0)== 2.0`.
- Wir würden auch erwarten dass `sqrt(x)* sqrt(x)== x` für verscheidene `x`.
- Wir müssen aber die begrenzte Präzision von Fließkommazahlen beachten.
- Selbst wenn wir so nahe wie nur möglich an $\sqrt{x}$ für eine `x` herankommen, so haben wir trotzdem nur eine Auflösung von 15 bis 16 Ziffern.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir würden auch erwarten dass `sqrt(x)* sqrt(x)== x` für verscheidene `x`.
- Wir müssen aber die begrenzte Präzision von Fließkommazahlen beachten.
- Selbst wenn wir so nahe wie nur möglich an $\sqrt{x}$ für eine `x` herankommen, so haben wir trotzdem nur eine Auflösung von 15 bis 16 Ziffern.
- Wir müssen also etwas Lust lassen, wenn wir berechnen `s3 = sqrt(3.0)` und hoffen, dass `s3 * s3 == 3.0`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Selbst wenn wir so nahe wie nur möglich an $\sqrt{x}$ für eine `x` herankommen, so haben wir trotzdem nur eine Auflösung von 15 bis 16 Ziffern.
- Wir müssen also etwas Lust lassen, wenn wir berechnen `s3 = sqrt(3.0)` und hoffen, dass `s3 * s3 == 3.0`.
- Wir machen dass, in dem wir schreiben `abs(s3 * s3 - 3.0)<= 5e-16`, also annehmen, dass der Unterschied zwischen `3.0` und `s3 * s3` nicht größer ist als $5 * 10^{-16}$.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir müssen also etwas Lust lassen, wenn wir berechnen `s3 = sqrt(3.0)` und hoffen, dass `s3 * s3 == 3.0`.
- Wir machen dass, in dem wir schreiben `abs(s3 * s3 - 3.0)<= 5e-16`, also annehmen, dass der Unterschied zwischen `3.0` und `s3 * s3` nicht größer ist als $5 * 10^{-16}$.
- Auf der anderen Seite sollte die Quadratwurzel von `1e10 * 1e10` genau darstellbar sein, nämlich als `1e10`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir machen dass, in dem wir schreiben `abs(s3 * s3 - 3.0)<= 5e-16`, also annehmen, dass der Unterschied zwischen `3.0` und `s3 * s3` nicht größer ist als $5 * 10^{-16}$.
- Auf der anderen Seite sollte die Quadratwurzel von `1e10 * 1e10` genau darstellbar sein, nämlich als `1e10`.
- Der Datentyp `float` bietet uns auch die besonderen Werte `inf` und `nan` an, wie wir aus dem Modul `math` importieren.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Auf der anderen Seite sollte die Quadratwurzel von `1e10 * 1e10` genau darstellbar sein, nämlich als `1e10`.
- Der Datentyp `float` bietet uns auch die besonderen Werte `inf` und `nan` an, wie wir aus dem Modul `math` importieren.
- `inf` steht für „zu große für den Datentyp `float`“ und wird oft als $+\infty$ interpretiert.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Auf der anderen Seite sollte die Quadratwurzel von `1e10 * 1e10` genau darstellbar sein, nämlich als `1e10`.
- Der Datentyp `float` bietet uns auch die besonderen Werte `inf` und `nan` an, wie wir aus dem Modul `math` importieren.
- `inf` steht für „zu große für den Datentyp `float`“ und wird oft als $+\infty$ interpretiert.
- `nan` steht im Grunde für „undefiniert“ und ist das Ergebnis von Operationen wie z. B. `inf - inf`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)² should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10² = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Der Datentyp `float` bietet uns auch die besonderen Werte `inf` und `nan` an, wie wir aus dem Modul `math` importieren.
- `inf` steht für „zu große für den Datentyp `float`“ und wird oft als $+\infty$ interpretiert.
- `nan` steht im Grunde für „undefiniert“ und ist das Ergebnis von Operationen wie z. B. `inf - inf`.
- Unsere `sqrt`-Funktion sollte beide Werte verstehen und vernünftige Ergebnisse zurückliefern.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Der Datentyp `float` bietet uns auch die besonderen Werte `inf` und `nan` an, wie wir aus dem Modul `math` importieren.
- `inf` steht für „zu große für den Datentyp `float`“ und wird oft als $+\infty$ interpretiert.
- `nan` steht im Grunde für „undefiniert“ und ist das Ergebnis von Operationen wie z. B. `inf - inf`.
- Unsere `sqrt`-Funktion sollte beide Werte verstehen und vernünftige Ergebnisse zurückliefern.
- `sqrt(inf)` sollte wieder `inf` zurückliefern.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- `inf` steht für „zu große für den Datentyp `float`“ und wird oft als $+\infty$ interpretiert.
- `nan` steht im Grunde für „undefiniert“ und ist das Ergebnis von Operationen wie z. B. `inf - inf`.
- Unsere `sqrt`-Funktion sollte beide Werte verstehen und vernünftige Ergebnisse zurückliefern.
- `sqrt(inf)` sollte wieder `inf` zurückliefern.
- `sqrt(nan)` sollte wieder `nan` liefern.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- `nan` steht im Grunde für „undefiniert“ und ist das Ergebnis von Operationen wie z. B. `inf - inf`.
- Unsere `sqrt`-Funktion sollte beide Werte verstehen und vernünftige Ergebnisse zurückliefern.
- `sqrt(inf)` sollte wieder `inf` zurückliefern.
- `sqrt(nan)` sollte wieder `nan` liefern.
- Wie können letzteres natürlich nicht mit `sqrt(nan) == nan`, weil `==` immer `False` liefert, sobald ein `nan` involviert ist.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)² should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10² = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Unsere `sqrt`-Funktion sollte beide Werte verstehen und vernünftige Ergebnisse zurückliefern.
- `sqrt(inf)` sollte wieder `inf` zurückliefern.
- `sqrt(nan)` sollte wieder `nan` liefern.
- Wie können letzteres natürlich nicht mit `sqrt(nan)== nan`, weil `==` immer `False` liefert, sobald ein `nan` involviert ist.
- Wir müssen also die Funktion `isnan` aus dem `math`-Module verwenden, um auf `nan` zu testen.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- `sqrt(inf)` sollte wieder `inf` zurückliefern.
- `sqrt(nan)` sollte wieder `nan` liefern.
- Wie können letzteres natürlich nicht mit `sqrt(nan) == nan`, weil `==` immer `False` liefert, sobald ein `nan` involviert ist.
- Wir müssen also die Funktion `isnan` aus dem `math`-Module verwenden, um auf `nan` zu testen.
- Damit haben wir die vernünftigsten Eingaben sowohl für `factorial` und `sqrt` in unseren Tests berücksichtigt.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- `sqrt(nan)` sollte wieder `nan` liefern.
- Wie können letzteres natürlich nicht mit `sqrt(nan)== nan`, weil `==` immer `False` liefert, sobald ein `nan` involviert ist.
- Wir müssen also die Funktion `isnan` aus dem `math`-Module verwenden, um auf `nan` zu testen.
- Damit haben wir die vernünftigsten Eingaben sowohl für `factorial` und `sqrt` in unseren Tests berücksichtigt.
- Wir haben angegeben, was für Ausgaben wir für diese Werte erwarten.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)² should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10² = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wie können letzteres natürlich nicht mit `sqrt(nan)== nan`, weil `==` immer `False` liefert, sobald ein `nan` involviert ist.
- Wir müssen also die Funktion `isnan` aus dem `math`-Module verwenden, um auf `nan` zu testen.
- Damit haben wir die vernünftigsten Eingaben sowohl für `factorial` und `sqrt` in unseren Tests berücksichtigt.
- Wir haben angegeben, was für Ausgaben wir für diese Werte erwarten.
- Diese Erwartungen sind nun als Test Cases implementiert.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir müssen also die Funktion `isnan` aus dem `math`-Module verwenden, um auf `nan` zu testen.
- Damit haben wir die vernünftigsten Eingaben sowohl für `factorial` und `sqrt` in unseren Tests berücksichtigt.
- Wir haben angegeben, was für Ausgaben wir für diese Werte erwarten.
- Diese Erwartungen sind nun als Test Cases implementiert.
- Gleichgültig wie `factorial` oder `sqrt` realisiert wurden, sie sollten diese Test Cases bestehen.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Damit haben wir die vernünftigsten Eingaben sowohl für `factorial` und `sqrt` in unseren Tests berücksichtigt.
- Wir haben angegeben, was für Ausgaben wir für diese Werte erwarten.
- Diese Erwartungen sind nun als Test Cases implementiert.
- Gleichgültig wie `factorial` oder `sqrt` realisiert wurden, sie sollten diese Test Cases bestehen.
- Oder sie sind falsch.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wir haben angegeben, was für Ausgaben wir für diese Werte erwarten.
- Diese Erwartungen sind nun als Test Cases implementiert.
- Gleichgültig wie `factorial` oder `sqrt` realisiert wurden, sie sollten diese Test Cases bestehen.
- Oder sie sind falsch.
- Wir führen nun unsere Tests aus, mit dem Kommando `pytest --timeout=10 --no-header --tb=short test_my_math.py`.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Diese Erwartungen sind nun als Test Cases implementiert.
- Gleichgültig wie `factorial` oder `sqrt` realisiert wurden, sie sollten diese Test Cases bestehen.
- Oder sie sind falsch.
- Wir führen nun unsere Tests aus, mit dem Kommando `pytest --timeout=10 --no-header --tb=short test_my_math.py`.
- Der erste Teil, `pytest`, ruft pytest auf.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Gleichgültig wie `factorial` oder `sqrt` realisiert wurden, sie sollten diese Test Cases bestehen.
- Oder sie sind falsch.
- Wir führen nun unsere Tests aus, mit dem Kommando `pytest --timeout=10 --no-header --tb=short test_my_math.py`.
- Der erste Teil, `pytest`, ruft pytest auf.
- Das `--timeout=10` definiert ein Timeout von zehn Sekunden.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Oder sie sind falsch.
- Wir führen nun unsere Tests aus, mit dem Kommando `pytest --timeout=10 --no-header --tb=short test_my_math.py`.
- Der erste Teil, `pytest`, ruft pytest auf.
- Das `--timeout=10` definiert ein Timeout von zehn Sekunden.
- Wenn ein Test länger als zehn Sekunden braucht, dann wird er abgebrochen und gilt als fehlgeschlagen.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Der erste Teil, `pytest`, ruft pytest auf.
- Das `--timeout=10` definiert ein Timeout von zehn Sekunden.
- Wenn ein Test länger als zehn Sekunden braucht, dann wird er abgebrochen und gilt als fehlgeschlagen.
- `--no-header` und `--tb=short` sind nur da, um pytest zu sagen, dass es kürzeren und kompakten Output produzieren soll.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Der erste Teil, `pytest`, ruft pytest auf.
- Das `--timeout=10` definiert ein Timeout von zehn Sekunden.
- Wenn ein Test länger als zehn Sekunden braucht, dann wird er abgebrochen und gilt als fehlgeschlagen.
- `--no-header` und `--tb=short` sind nur da, um pytest zu sagen, dass es kürzeren und kompakten Output produzieren soll.
- Sonst passt der Output nicht auf eine Slide...

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Das `--timeout=10` definiert ein Timeout von zehn Sekunden.
- Wenn ein Test länger als zehn Sekunden braucht, dann wird er abgebrochen und gilt als fehlgeschlagen.
- `--no-header` und `--tb=short` sind nur da, um pytest zu sagen, dass es kürzeren und kompakten Output produzieren soll.
- Sonst passt der Output nicht auf eine Slide...
- Sie brauchen diese Parameter normalerweise nicht.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- Wenn ein Test länger als zehn Sekunden braucht, dann wird er abgebrochen und gilt als fehlgeschlagen.
- `--no-header` und `--tb=short` sind nur da, um pytest zu sagen, dass es kürzeren und kompakten Output produzieren soll.
- Sonst passt der Output nicht auf eine Slide...
- Sie brauchen diese Parameter normalerweise nicht.
- Zu guter Letzt geben wir mit `test_my_math.py` die Datei mit den Tests an, die ausgeführt werden soll.

```python
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 4790'016'00
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

```python
"""The syntax of an assert statement in Python."""

assert booleanExpr  # raises AssertionError if not booleanExpr
```

Testen wir unser Eigenes Modul

- `--no-header` und `--tb=short` sind nur da, um pytest zu sagen, dass es kürzeren und kompakten Output produzieren soll.
- Sonst passt der Output nicht auf eine Slide...
- Sie brauchen diese Parameter normalerweise nicht.
- Zu guter Letzt geben wir mit `test_my_math.py` die Datei mit den Tests an, die ausgeführt werden soll.
- Vom Output unserer Testcases sehen wir, dass `test_factorial` ohne Probleme durchgelaufen ist.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================= test session starts
   ↪ ==========================
collected 2 items
test_my_math.py .F                                                       [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ -------------------------------
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info
   ↪ ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Testen wir unser Eigenes Modul

- Sonst passt der Output nicht auf eine Slide...
- Sie brauchen diese Parameter normalerweise nicht.
- Zu guter Letzt geben wir mit `test_my_math.py` die Datei mit den Tests an, die ausgeführt werden soll.
- Vom Output unserer Testcases sehen wir, dass `test_factorial` ohne Probleme durchgelaufen ist.
- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================= test session starts
    ↪ ==========================
collected 2 items
test_my_math.py .F                                                    [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ ------------------------------
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info
    ↪ ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s
    ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Testen wir unser Eigenes Modul

- Sie brauchen diese Parameter normalerweise nicht.
- Zu guter Letzt geben wir mit `test_my_math.py` die Datei mit den Tests an, die ausgeführt werden soll.
- Vom Output unserer Testcases sehen wir, dass `test_factorial` ohne Probleme durchgelaufen ist.
- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.
- Das passierte, als wir versucht haben, `sqrt(0.0)` zu berechnen.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================= test session starts
    ↪ ==========================
collected 2 items
test_my_math.py .F                                                       [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ ------------------------------
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info
    ↪ ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s
    ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================= test session starts
    ↪ ==========================
collected 2 items
test_my_math.py .F                                                    [100%]
============================== FAILURES
    ↪ ==============================
------------------------------ test_sqrt
    ↪ ------------------------------
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info
    ↪ ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s
    ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.
- Das passierte, als wir versucht haben, `sqrt(0.0)` zu berechnen.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================= test session starts
   ↪ ==========================
collected 2 items
test_my_math.py .F                                                    [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ ------------------------------
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info
   ↪ ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.
- Das passierte, als wir versucht haben, `sqrt(0.0)` zu berechnen.
- Was ist da schiefgelaufen?

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================= test session starts
   ↪ ==========================
collected 2 items
test_my_math.py .F                                                       [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ ------------------------------
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info
   ↪ ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.
- Das passierte, als wir versucht haben, `sqrt(0.0)` zu berechnen.
- Was ist da schiefgelaufen?
- Gucken wir uns unseren Kode an und öffnen wir unser Modul `my_math`.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem `ZeroDivisionError`, der in unserer `sqrt`-Funktion passiert ist.
- Das passierte, als wir versucht haben, `sqrt(0.0)` zu berechnen.
- Was ist da schiefgelaufen?
- Gucken wir uns unseren Kode an und öffnen wir unser Modul `my_math`.
- Wir sehen, dass unsere erste Annäherund für die Quadratwurzel gleich `1` ist.

```
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Das passierte, als wir versucht haben, `sqrt(0.0)` zu berechnen.
- Was ist da schiefgelaufen?
- Gucken wir uns unseren Kode an und öffnen wir unser Modul `my_math`.
- Wir sehen, dass unsere erste Annäherund für die Quadratwurzel gleich `1` ist.
- Wir setzen ihn als `guess: float = 1.0`.

```
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Was ist da schiefgelaufen?
- Gucken wir uns unseren Kode an und öffnen wir unser Modul `my_math`.
- Wir sehen, dass unsere erste Annäherund für die Quadratwurzel gleich `1` ist.
- Wir setzen ihn als `guess: float = 1.0`.
- In jedem Schritt rechnen wir dann `guess = 0.5 * (guess + number / guess)`.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Gucken wir uns unseren Kode an und öffnen wir unser Modul `my_math`.
- Wir sehen, dass unsere erste Annäherund für die Quadratwurzel gleich `1` ist.
- Wir setzen ihn als `guess: float = 1.0`.
- In jedem Schritt rechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Wenn `number` nun `0.0` ist, dann bedeutet dass effektiv `guess = 0.5 * guess`.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Wir sehen, dass unsere erste Annäherund für die Quadratwurzel gleich `1` ist.
- Wir setzen ihn als `guess: float = 1.0`.
- In jedem Schritt rechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Wenn `number` nun `0.0` ist, dann bedeutet dass effektiv `guess = 0.5 * guess`.
- `number` bleibt ja unverändert bei `0.0`, also wiederholen wir das immer wieder.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Wir setzen ihn als `guess: float = 1.0`.
- In jedem Schritt rechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Wenn `number` nun `0.0` ist, dann bedeutet dass effektiv `guess = 0.5 * guess`.
- `number` bleibt ja unverändert bei `0.0`, also wiederholen wir das immer wieder.
- `guess` wird also immer wieder halbiert.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- In jedem Schritt rechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Wenn `number` nun `0.0` ist, dann bedeutet dass effektiv `guess = 0.5 * guess`.
- `number` bleibt ja unverändert bei `0.0`, also wiederholen wir das immer wieder.
- `guess` wird also immer wieder halbiert.
- Wir wissen, dass `floats` eine begrenzte Auflösung haben.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Wenn `number` nun `0.0` ist, dann bedeutet dass effektiv `guess = 0.5 * guess`.
- `number` bleibt ja unverändert bei `0.0`, also wiederholen wir das immer wieder.
- `guess` wird also immer wieder halbiert.
- Wir wissen, dass `floats` eine begrenzte Auflösung haben.
- Wir wissen auch, dass wenn wir immer kleinere `float`-Werte angeben oder berechnen, wir irgendwann bei `0.0` herauskommen.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- `number` bleibt ja unverändert bei `0.0`, also wiederholen wir das immer wieder.
- `guess` wird also immer wieder halbiert.
- Wir wissen, dass `floats` eine begrenzte Auflösung haben.
- Wir wissen auch, dass wenn wir immer kleinere `float`-Werte angeben oder berechnen, wir irgendwann bei `0.0` herauskommen.
- Das passiert hier auch.

```
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- `guess` wird also immer wieder halbiert.
- Wir wissen, dass `floats` eine begrenzte Auflösung haben.
- Wir wissen auch, dass wenn wir immer kleinere `float`-Werte angeben oder berechnen, wir irgendwann bei `0.0` herauskommen.
- Das passiert hier auch.
- `guess` wird irgendwann `0.0`.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Wir wissen, dass `floats` eine begrenzte Auflösung haben.
- Wir wissen auch, dass wenn wir immer kleinere `float`-Werte angeben oder berechnen, wir irgendwann bei `0.0` herauskommen.
- Das passiert hier auch.
- `guess` wird irgendwann `0.0`.
- In dem Rechenschritt danach machen wir dann `0.0 / 0.0`.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Wir wissen auch, dass wenn wir immer kleinere `float`-Werte angeben oder berechnen, wir irgendwann bei `0.0` herauskommen.
- Das passiert hier auch.
- `guess` wird irgendwann `0.0`.
- In dem Rechenschritt danach machen wir dann `0.0 / 0.0`.
- Ka-Boom! `ZeroDivisionError`!

```
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Das passiert hier auch.
- `guess` wird irgendwann `0.0`.
- In dem Rechenschritt danach machen wir dann `0.0 / 0.0`.
- Ka-Boom! `ZeroDivisionError`!
- Um dieses Problem zu lösen, führen wir eine Prüfung auf `if number <= 0.0` ein.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- `guess` wird irgendwann `0.0`.
- In dem Rechenschritt danach machen wir dann `0.0 / 0.0`.
- Ka-Boom! `ZeroDivisionError`!
- Um dieses Problem zu lösen, führen wir eine Prüfung auf `if number <= 0.0` ein.
- Wenn diese zutrifft, liefern wir einfach `0.0` zurück.

```
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- In dem Rechenschritt danach machen wir dann `0.0 / 0.0`.
- Ka-Boom! `ZeroDivisionError`!
- Um dieses Problem zu lösen, führen wir eine Prüfung auf `if number <= 0.0` ein.
- Wenn diese zutrifft, liefern wir einfach `0.0` zurück.
- (Wir ignorieren hier erstmal negative Zahlen ... darum kümmern wir uns, wenn wir lernen, wie man `Exceptions` selber auslöst.)

```
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Ka-Boom! `ZeroDivisionError`!
- Um dieses Problem zu lösen, führen wir eine Prüfung auf `if number <= 0.0` ein.
- Wenn diese zutrifft, liefern wir einfach `0.0` zurück.
- (Wir ignorieren hier erstmal negative Zahlen ... darum kümmern wir uns, wenn wir lernen, wie man `Exceptions` selber auslöst.)
- Wir haben also nun eine neue Version `my_math_2.py` unseres Moduls.

```
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir die Division durch 0

- Um dieses Problem zu lösen, führen wir eine Prüfung auf `if number <= 0.0` ein.
- Wenn diese zutrifft, liefern wir einfach `0.0` zurück.
- (Wir ignorieren hier erstmal negative Zahlen ... darum kümmern wir uns, wenn wir lernen, wie man `Exceptions` selber auslöst.)
- Wir haben also nun eine neue Version `my_math_2.py` unseres Moduls.
- Diese müssen wir wieder mit den selben Test Cases testen.

```
"""Testing our second version of the `my_math` module."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math_2 import sqrt  # Get our 2nd square root implementation.

# test_factorial() is omitted for brevity

def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math_2`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Lösen wir die Division durch 0

- Wenn diese zutrifft, liefern wir einfach `0.0` zurück.
- (Wir ignorieren hier erstmal negative Zahlen ... darum kümmern wir uns, wenn wir lernen, wie man `Exceptions` selber auslöst.)
- Wir haben also nun eine neue Version `my_math_2.py` unseres Moduls.
- Diese müssen wir wieder mit den selben Test Cases testen.
- Auf der positiven Seite: Der Fehler von vorhin ist weg.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_2.py F                                                    [100%]
============================== FAILURES
    ↪ ===============================
_____________________________ test_sqrt
    ↪ ------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
    ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
    ↪ pytest-timeout.
========================= 1 failed in 10.01s
    ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- (Wir ignorieren hier erstmal negative Zahlen ... darum kümmern wir uns, wenn wir lernen, wie man `Exceptions` selber auslöst.)
- Wir haben also nun eine neue Version `my_math_2.py` unseres Moduls.
- Diese müssen wir wieder mit den selben Test Cases testen.
- Auf der positiven Seite: Der Fehler von vorhin ist weg.
- Jetzt schlägt der Test mit einem Timeout fehl.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_2.py F                                                    [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
    ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
    ↪ pytest-timeout.
========================= 1 failed in 10.01s
    ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- Wir haben also nun eine neue Version `my_math_2.py` unseres Moduls.
- Diese müssen wir wieder mit den selben Test Cases testen.
- Auf der positiven Seite: Der Fehler von vorhin ist weg.
- Jetzt schlägt der Test mit einem Timeout fehl.
- Wir rufen pytest nämlich mit der Option `--timeout=10` (für die wir das Package `pytest-timeout` brauchen).

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
   ↪ ==========================
collected 1 item
test_my_math_2.py F                                                      [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
   ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
   ↪ pytest-timeout.
========================= 1 failed in 10.01s
   ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- Diese müssen wir wieder mit den selben Test Cases testen.
- Auf der positiven Seite: Der Fehler von vorhin ist weg.
- Jetzt schlägt der Test mit einem Timeout fehl.
- Wir rufen pytest nämlich mit der Option `--timeout=10` (für die wir das Package `pytest-timeout` brauchen).
- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_2.py F                                                    [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
    ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
    ↪ pytest-timeout.
========================= 1 failed in 10.01s
    ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- Auf der positiven Seite: Der Fehler von vorhin ist weg.
- Jetzt schlägt der Test mit einem Timeout fehl.
- Wir rufen pytest nämlich mit der Option `--timeout=10` (für die wir das Package `pytest-timeout` brauchen).
- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.
- Das ist ein vernünftiges Limit für unsere Beispiele hier.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
   ↪ ==========================
collected 1 item
test_my_math_2.py F                                                      [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
   ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
   ↪ pytest-timeout.
========================= 1 failed in 10.01s
   ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0

- Jetzt schlägt der Test mit einem Timeout fehl.
- Wir rufen pytest nämlich mit der Option `--timeout=10` (für die wir das Package `pytest-timeout` brauchen).
- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.
- Das ist ein vernünftiges Limit für unsere Beispiele hier.
- In praktischen Situationen wollen Sie vielleicht eine größere Zahl nehmen.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
   ↪ ==========================
collected 1 item
test_my_math_2.py F                                                  [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
   ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
   ↪ pytest-timeout.
========================= 1 failed in 10.01s
   ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir die Division durch 0



- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.
- Das ist ein vernünftiges Limit für unsere Beispiele hier.
- In praktischen Situationen wollen Sie vielleicht eine größere Zahl nehmen.

Gute Praxis

Setzen Sie immer ein Timeout für Ihre Unit Tests.

Lösen wir die Division durch 0

- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.
- Das ist ein vernünftiges Limit für unsere Beispiele hier.
- In praktischen Situationen wollen Sie vielleicht eine größere Zahl nehmen.

Gute Praxis

Setzen Sie immer ein Timeout für Ihre Unit Tests. Dieses Timeout kann groß sein, vielleicht eine Stunde, aber es dient als Sicherheitsgarantie gegen endlose Schleifen, Deadlocks, und andere Situationen, die praktisch äquivalent zu einem Test-Fehlschlag sind.

Lösen wir die Division durch 0

- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.
- Das ist ein vernünftiges Limit für unsere Beispiele hier.
- In praktischen Situationen wollen Sie vielleicht eine größere Zahl nehmen.

Gute Praxis

Setzen Sie immer ein Timeout für Ihre Unit Tests. Dieses Timeout kann groß sein, vielleicht eine Stunde, aber es dient als Sicherheitsgarantie gegen endlose Schleifen, Deadlocks, und andere Situationen, die praktisch äquivalent zu einem Test-Fehlschlag sind. Timouts schützen automatische Builds und Continuous Integration-Systeme vor dem „Verstopfen“.

Lösen wir die Division durch 0

- Das begrenzt die maximale Laufzeit unserer Tests auf zehn Sekunden.
- Das ist ein vernünftiges Limit für unsere Beispiele hier.
- In praktischen Situationen wollen Sie vielleicht eine größere Zahl nehmen.
- Die Frage ist nun: Warum bekommen wir hier ein Timeout?

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
   ↪ ==========================
collected 1 item
test_my_math_2.py F                                                    [100%]
============================== FAILURES
   ↪ ===============================
------------------------------ test_sqrt
   ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
   ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
   ↪ pytest-timeout.
========================= 1 failed in 10.01s
   ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir das Timeout

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem Timeout.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_2.py F                                                    [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
    ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
    ↪ pytest-timeout.
========================= 1 failed in 10.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir das Timeout

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem Timeout.
- Wir sehen vom Output von pytest, dass das Timeout passiert, wenn wir Argument `inf` als Wert für Parameter `number` unserer `sqrt`-Funktion übergeben.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_2.py F                                                      [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
    ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
    ↪ pytest-timeout.
========================= 1 failed in 10.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir das Timeout

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem Timeout.
- Wir sehen vom Output von pytest, dass das Timeout passiert, wenn wir Argument `inf` als Wert für Parameter `number` unserer `sqrt`-Funktion übergeben.
- Was könnte hier passiert sein?

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_2.py F                                                    [100%]
============================== FAILURES
    ↪ ===============================
------------------------------ test_sqrt
    ↪ -------------------------------
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info
    ↪ ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from
    ↪ pytest-timeout.
========================= 1 failed in 10.01s
    ↪ ============================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Lösen wir das Timeout

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem Timeout.
- Wir sehen vom Output von pytest, dass das Timeout passiert, wenn wir Argument `inf` als Wert für Parameter `number` unserer `sqrt`-Funktion übergeben.
- Was könnte hier passiert sein?
- OK, wir setzen am Anfang wieder `guess = 1.0`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- `test_sqrt` ist fehlgeschlagen, und zwar mit einem Timeout.
- Wir sehen vom Output von pytest, dass das Timeout passiert, wenn wir Argument `inf` als Wert für Parameter `number` unserer `sqrt`-Funktion übergeben.
- Was könnte hier passiert sein?
- OK, wir setzen am Anfang wieder `guess = 1.0`.
- In der ersten Iteration berechnen wir dann `guess = 0.5 * (guess + number / guess)`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Wir sehen vom Output von pytest, dass das Timeout passiert, wenn wir Argument `inf` als Wert für Parameter `number` unserer `sqrt`-Funktion übergeben.
- Was könnte hier passiert sein?
- OK, wir setzen am Anfang wieder `guess = 1.0`.
- In der ersten Iteration berechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Das ist das selbe wie `0.5 * (1 + inf / 1)`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Was könnte hier passiert sein?
- OK, wir setzen am Anfang wieder `guess = 1.0`.
- In der ersten Iteration berechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Das ist das selbe wie `0.5 * (1 + inf / 1)`.
- Und das liefert dann wiederum `guess = 0.5 * inf`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- OK, wir setzen am Anfang wieder `guess = 1.0`.
- In der ersten Iteration berechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Das ist das selbe wie `0.5 * (1 + inf / 1)`.
- Und das liefert dann wiederum `guess = 0.5 * inf`.
- Das macht `guess` dann auch zu `inf`.

```
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- In der ersten Iteration berechnen wir dann `guess = 0.5 * (guess + number / guess)`.
- Das ist das selbe wie `0.5 * (1 + inf / 1)`.
- Und das liefert dann wiederum `guess = 0.5 * inf`.
- Das macht `guess` dann auch zu `inf`.
- In der zweiten Iteration haben wir wieder `guess = 0.5 * (guess + number / guess)`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Das ist das selbe wie `0.5 * (1 + inf / 1)`.
- Und das liefert dann wiederum `guess = 0.5 * inf`.
- Das macht `guess` dann auch zu `inf`.
- In der zweiten Iteration haben wir wieder `guess = 0.5 * (guess + number / guess)`.
- Wir berechnen nun allerdings `0.5 * (inf + inf / inf)`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Und das liefert dann wiederum `guess = 0.5 * inf`.
- Das macht `guess` dann auch zu `inf`.
- In der zweiten Iteration haben wir wieder `guess = 0.5 * (guess + number / guess)`.
- Wir berechnen nun allerdings `0.5 * (inf + inf / inf)`.
- Und `inf / inf` ergibt `nan`[28].

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Das macht `guess` dann auch zu `inf`.
- In der zweiten Iteration haben wir wieder `guess =` `0.5 * (guess + number / guess)`.
- Wir berechnen nun allerdings `0.5 * (inf + inf / inf)`.
- Und `inf / inf` ergibt `nan`[28].
- Das `nan` „infiziert“ den Rest der Berechnung, so dass am Ende `guess = nan`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- In der zweiten Iteration haben wir wieder `guess = 0.5 * (guess + number / guess)`.
- Wir berechnen nun allerdings `0.5 * (inf + inf / inf)`.
- Und `inf / inf` ergibt `nan`[28].
- Das `nan` „infiziert“ den Rest der Berechnung, so dass am Ende `guess = nan`.
- Von jetzt an ergeben alle Berechnungen `nan`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Wir berechnen nun allerdings `0.5 * (inf + inf / inf)`.
- Und `inf / inf` ergibt `nan`[28].
- Das `nan` „infiziert“ den Rest der Berechnung, so dass am Ende `guess = nan`.
- Von jetzt an ergeben alle Berechnungen `nan`.
- Nun gilt aber `nan != nan` und `isclose(nan, nan)` wir niemals `True`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Und `inf / inf` ergibt `nan`[28].
- Das `nan` „infiziert“ den Rest der Berechnung, so dass am Ende `guess = nan`.
- Von jetzt an ergeben alle Berechnungen `nan`.
- Nun gilt aber `nan != nan` und `isclose(nan, nan)` wir niemals `True`.
- Unsere Schleife endet also niemals.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Das `nan` „infiziert“ den Rest der Berechnung, so dass am Ende `guess = nan`.
- Von jetzt an ergeben alle Berechnungen `nan`.
- Nun gilt aber `nan != nan` und `isclose(nan, nan)` wir niemals `True`.
- Unsere Schleife endet also niemals.
- Das Zeitlimit hat uns hier gerettet!

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Von jetzt an ergeben alle Berechnungen `nan`.
- Nun gilt aber `nan != nan` und `isclose(nan, nan)` wir niemals `True`.
- Unsere Schleife endet also niemals.
- Das Zeitlimit hat uns hier gerettet!
- Unsere `sqrt`-Funktion geht in eine Endlosschleife für `number = inf`.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Nun gilt aber `nan != nan` und `isclose(nan, nan)` wir niemals `True`.
- Unsere Schleife endet also niemals.
- Das Zeitlimit hat uns hier gerettet!
- Unsere `sqrt`-Funktion geht in eine Endlosschleife für `number = inf`.
- Und unser Unit Test wäre für immer weitergelaufen.

```python
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0   # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Unsere Schleife endet also niemals.
- Das Zeitlimit hat uns hier gerettet!
- Unsere `sqrt`-Funktion geht in eine Endlosschleife für `number = inf`.
- Und unser Unit Test wäre für immer weitergelaufen.
- Lösen wir das Problem in einer dritten Version unseres Moduls, die wir in Datei `my_math_3.py` speichern.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Das Zeitlimit hat uns hier gerettet!
- Unsere `sqrt`-Funktion geht in eine Endlosschleife für `number = inf`.
- Und unser Unit Test wäre für immer weitergelaufen.
- Lösen wir das Problem in einer dritten Version unseres Moduls, die wir in Datei `my_math_3.py` speichern.
- Wir fügen einfach eine neue Alternative ein.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Unsere `sqrt`-Funktion geht in eine Endlosschleife für `number = inf`.
- Und unser Unit Test wäre für immer weitergelaufen.
- Lösen wir das Problem in einer dritten Version unseres Moduls, die wir in Datei `my_math_3.py` speichern.
- Wir fügen einfach eine neue Alternative ein.
- Wir prüfen `if not isfinite(number)` und wenn das zutrifft, liefern wir einfach `number` direkt als Ergebnis zurück.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Und unser Unit Test wäre für immer weitergelaufen.
- Lösen wir das Problem in einer dritten Version unseres Moduls, die wir in Datei `my_math_3.py` speichern.
- Wir fügen einfach eine neue Alternative ein.
- Wir prüfen `if not isfinite(number)` und wenn das zutrifft, liefern wir einfach `number` direkt als Ergebnis zurück.
- `isfinite` ist eine weitere Funktion aus dem `math`-Modul.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Lösen wir das Problem in einer dritten Version unseres Moduls, die wir in Datei `my_math_3.py` speichern.
- Wir fügen einfach eine neue Alternative ein.
- Wir prüfen `if not isfinite(number)` und wenn das zutrifft, liefern wir einfach `number` direkt als Ergebnis zurück.
- `isfinite` ist eine weitere Funktion aus dem `math`-Modul.
- Sie nimmt einen Parameter an und liefert `True`, wenn der eine finite, also wohldefinierte und endliche Zahl ist.

```python
"""A third version of our module with mathematics routines."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Wir fügen einfach eine neue Alternative ein.
- Wir prüfen `if not isfinite(number)` und wenn das zutrifft, liefern wir einfach `number` direkt als Ergebnis zurück.
- `isfinite` ist eine weitere Funktion aus dem `math`-Modul.
- Sie nimmt einen Parameter an und liefert `True`, wenn der eine finite, also wohldefinierte und endliche Zahl ist.
- `isfinite` liefert `False`, wenn sein Argument `inf`, `-inf`, oder `nan` ist.

```python
"""A third version of our module with mathematics routines."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Wir prüfen `if not isfinite(number)` und wenn das zutrifft, liefern wir einfach `number` direkt als Ergebnis zurück.
- `isfinite` ist eine weitere Funktion aus dem `math`-Modul.
- Sie nimmt einen Parameter an und liefert `True`, wenn der eine finite, also wohldefinierte und endliche Zahl ist.
- `isfinite` liefert `False`, wenn sein Argument `inf`, `-inf`, oder `nan` ist.
- Diese Bedingung kann in unserer Funktion nur für `inf` oder `nan` `True` werden, weil wir ja schon `0.0` liefern wenn `number <= 0.0`.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- `isfinite` ist eine weitere Funktion aus dem `math`-Modul.
- Sie nimmt einen Parameter an und liefert `True`, wenn der eine finite, also wohldefinierte und endliche Zahl ist.
- `isfinite` liefert `False`, wenn sein Argument `inf`, `-inf`, oder `nan` ist.
- Diese Bedingung kann in unserer Funktion nur für `inf` oder `nan` `True` werden, weil wir ja schon `0.0` liefern wenn `number <= 0.0`.
- Wir würden also `inf` zurückgeben wenn unsere Funktion mit `inf` aufgerufen wird.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Sie nimmt einen Parameter an und liefert `True`, wenn der eine finite, also wohldefinierte und endliche Zahl ist.
- `isfinite` liefert `False`, wenn sein Argument `inf`, `-inf`, oder `nan` ist.
- Diese Bedingung kann in unserer Funktion nur für `inf` oder `nan` `True` werden, weil wir ja schon `0.0` liefern wenn `number <= 0.0`.
- Wir würden also `inf` zurückgeben wenn unsere Funktion mit `inf` aufgerufen wird.
- Und das ist richtig.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- `isfinite` liefert `False`, wenn sein Argument `inf`, `-inf`, oder `nan` ist.
- Diese Bedingung kann in unserer Funktion nur für `inf` oder `nan` `True` werden, weil wir ja schon `0.0` liefern wenn `number <= 0.0`.
- Wir würden also `inf` zurückgeben wenn unsere Funktion mit `inf` aufgerufen wird.
- Und das ist richtig.
- Wir würden `nan` zurückliefern, wenn unsere Funktion mit `nan` aufgerufen wird.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Diese Bedingung kann in unserer Funktion nur für `inf` oder `nan` `True` werden, weil wir ja schon `0.0` liefern wenn `number <= 0.0`.
- Wir würden also `inf` zurückgeben wenn unsere Funktion mit `inf` aufgerufen wird.
- Und das ist richtig.
- Wir würden `nan` zurückliefern, wenn unsere Funktion mit `nan` aufgerufen wird.
- Und das ist auch richtig.

```python
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Lösen wir das Timeout

- Wir würden also `inf` zurückgeben wenn unsere Funktion mit `inf` aufgerufen wird.
- Und das ist richtig.
- Wir würden `nan` zurückliefern, wenn unsere Funktion mit `nan` aufgerufen wird.
- Und das ist auch richtig.
- Testen wir also unser neues Modul nochmal mit den selben Test Cases.

```python
"""Testing our third version of the `my_math` module."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math_3 import sqrt  # Get our 3rd square root implementation.

# test_factorial() is omitted for brevity

def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math_3`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Lösen wir das Timeout

- Und das ist richtig.
- Wir würden `nan` zurückliefern, wenn unsere Funktion mit `nan` aufgerufen wird.
- Und das ist auch richtig.
- Testen wir also unser neues Modul nochmal mit den selben Test Cases.
- Und diesmal funktioniert es.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_3.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_3.py .                                                  [100%]
========================== 1 passed in 0.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Lösen wir das Timeout

- Wir würden `nan` zurückliefern, wenn unsere Funktion mit `nan` aufgerufen wird.
- Und das ist auch richtig.
- Testen wir also unser neues Modul nochmal mit den selben Test Cases.
- Und diesmal funktioniert es.
- Nice.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_3.py
========================= test session starts
    ↪ ==========================
collected 1 item
test_my_math_3.py .                                                      [100%]
========================== 1 passed in 0.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Zusammenfassung



Gezogene Lehren



Gute Praxis

Eine Funktion, die nicht mit Unit Tests getestet wurde, ist falsch.

Gezogene Lehren



Gute Praxis

Eine Funktion, die nicht mit Unit Tests getestet wurde, ist falsch.

Gute Praxis

Gute Unit Tests für eine Funktion sollten sowohl normale als auch extreme Eingabewerte abdecken.

Gezogene Lehren



Gute Praxis

Eine Funktion, die nicht mit Unit Tests getestet wurde, ist falsch.

Gute Praxis

Gute Unit Tests für eine Funktion sollten sowohl normale als auch extreme Eingabewerte abdecken. Wir sollten für alle Parameter sowohl die kleinsten als auch die größten möglichen Werte testen, sowie viele Werte aus dem normalerweise erwarteten Bereich.

Gezogene Lehren

Gute Praxis

Eine Funktion, die nicht mit Unit Tests getestet wurde, ist falsch.

Gute Praxis

Gute Unit Tests für eine Funktion sollten sowohl normale als auch extreme Eingabewerte abdecken. Wir sollten für alle Parameter sowohl die kleinsten als auch die größten möglichen Werte testen, sowie viele Werte aus dem normalerweise erwarteten Bereich.

Gute Praxis

Gute Unit Tests für eine Funktion sollten alle Zweige des Kontrollflusses in der Funktion abdecken.

Gezogene Lehren

Gute Praxis

Eine Funktion, die nicht mit Unit Tests getestet wurde, ist falsch.

Gute Praxis

Gute Unit Tests für eine Funktion sollten sowohl normale als auch extreme Eingabewerte abdecken. Wir sollten für alle Parameter sowohl die kleinsten als auch die größten möglichen Werte testen, sowie viele Werte aus dem normalerweise erwarteten Bereich.

Gute Praxis

Gute Unit Tests für eine Funktion sollten alle Zweige des Kontrollflusses in der Funktion abdecken. Wenn die Funktion eine Sache in einer Situation und eine andere Sache in einer anderen Situation tut, dann sollten beide Situationen mit Unit Tests abgedeckt sein.

Wichtigkeit

- Viele junge Programmer sind sich nicht der Wichtigkeit von Unit Tests bewusst.

Wichtigkeit

- Viele junge Programmer sind sich nicht der Wichtigkeit von Unit Tests bewusst.
- In der Lage zu seien, Unit Tests zu entwerfen und zu benutzen ist eine wichtige Fähigkeit in der Software Entwicklung.

Wichtigkeit

- Viele junge Programmer sind sich nicht der Wichtigkeit von Unit Tests bewusst.
- In der Lage zu seien, Unit Tests zu entwerfen und zu benutzen ist eine wichtige Fähigkeit in der Software Entwicklung.

> *No single factor is likely responsible for SQLite's popularity. Instead, in addition to its fundamentally embeddable design, several characteristics combine to make SQLite useful in a broad range of scenarios. In particular, SQLite strives to be:*
>
> *[...]*
>
> ***Reliable**. There are over 600 lines of test code for every line of code in SQLite[34]. Tests cover 100% of branches in the library. The test suite is extremely diverse, including fuzz tests, boundary value tests, regression tests, and tests that simulate operating system crashes, power losses, I/O errors, and out-of-memory errors. Due to its reliability, SQLite is often used in mission-critical applications such as flight software[33]*
>
> — *Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy und Jignesh M. Patel [27], 2022*

Wichtigkeit

> *No single factor is likely responsible for SQLite's popularity. Instead, in addition to its fundamentally embeddable design, several characteristics combine to make SQLite useful in a broad range of scenarios. In particular, SQLite strives to be:*
>
> *[. . .]*
>
> ***Reliable**. There are over 600 lines of test code for every line of code in SQLite[34]. Tests cover 100% of branches in the library. The test suite is extremely diverse, including fuzz tests, boundary value tests, regression tests, and tests that simulate operating system crashes, power losses, I/O errors, and out-of-memory errors. Due to its reliability, SQLite is often used in mission-critical applications such as flight software[33]*
>
> — *Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy und Jignesh M. Patel [27], 2022*

- SQLite ist die weitverbreitetste SQL Datenbank der Welt.

Wichtigkeit

> *No single factor is likely responsible for SQLite's popularity. Instead, in addition to its fundamentally embeddable design, several characteristics combine to make SQLite useful in a broad range of scenarios. In particular, SQLite strives to be:*
>
> *[. . .]*
>
> ***Reliable**. There are over 600 lines of test code for every line of code in SQLite[34]. Tests cover 100% of branches in the library. The test suite is extremely diverse, including fuzz tests, boundary value tests, regression tests, and tests that simulate operating system crashes, power losses, I/O errors, and out-of-memory errors. Due to its reliability, SQLite is often used in mission-critical applications such as flight software[33]*
>
> — *Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy und Jignesh M. Patel [27], 2022*

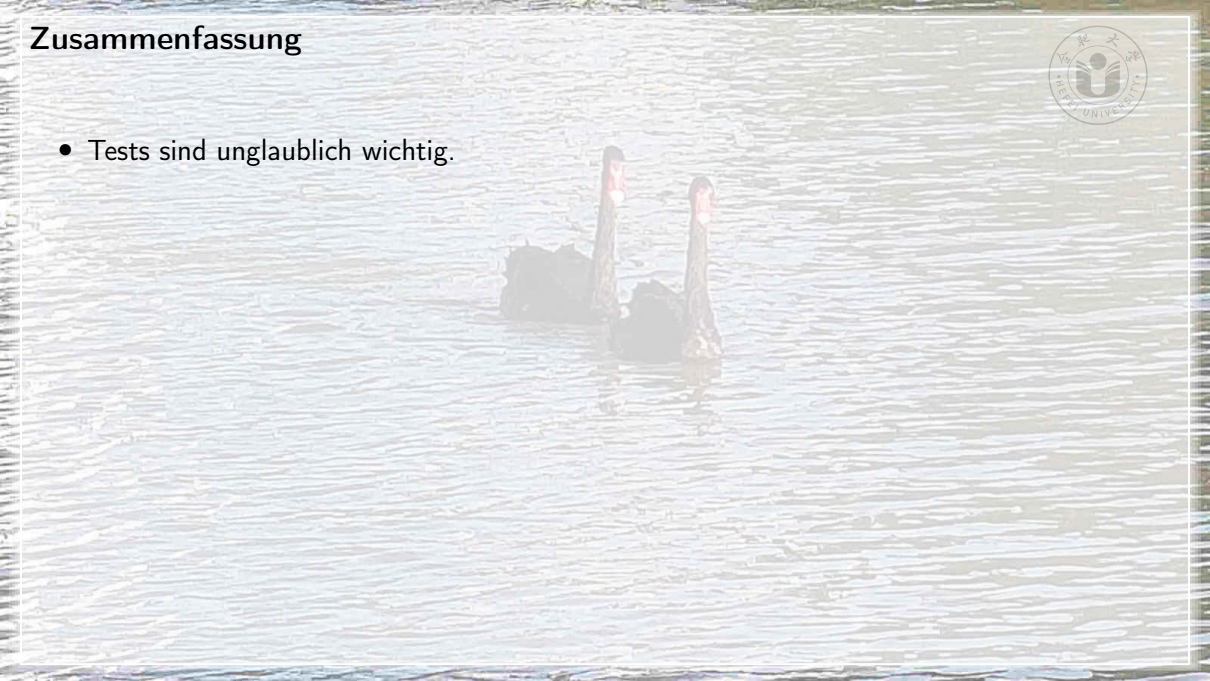
- SQLite ist die weitverbreitetste SQL Datenbank der Welt.
- Sie ist auf fast jedem Smartphone, Computer, Web Browser, Fernseher, und Auto installiert[15,27,88].

Wichtigkeit

> *No single factor is likely responsible for SQLite's popularity. Instead, in addition to its fundamentally embeddable design, several characteristics combine to make SQLite useful in a broad range of scenarios. In particular, SQLite strives to be:*
>
> *[. . .]*
>
> ***Reliable**. There are over 600 lines of test code for every line of code in SQLite[34]. Tests cover 100% of branches in the library. The test suite is extremely diverse, including fuzz tests, boundary value tests, regression tests, and tests that simulate operating system crashes, power losses, I/O errors, and out-of-memory errors. Due to its reliability, SQLite is often used in mission-critical applications such as flight software[33]*
>
> — *Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy und Jignesh M. Patel [27], 2022*

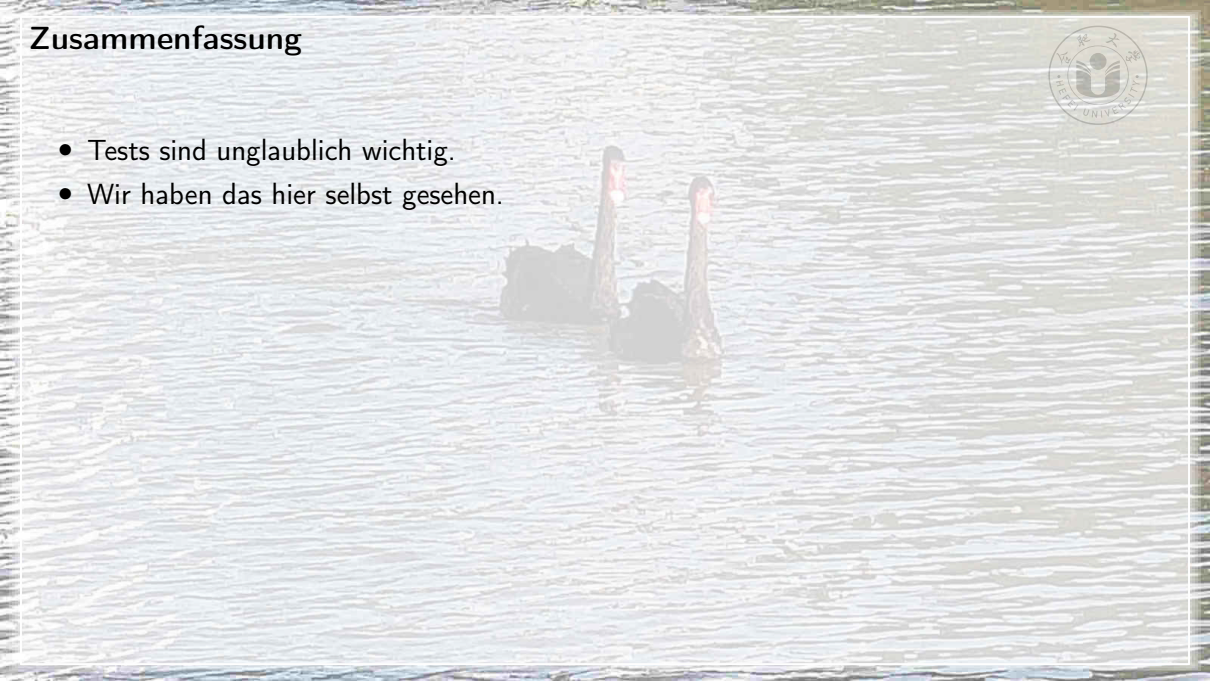
- Sie ist auf fast jedem Smartphone, Computer, Web Browser, Fernseher, und Auto installiert[15,27,88].
- Seine Entwickler stellen die Verlässlichkeit, gezeigt durch Tests, als eine der vier Gründen davon vor.

Zusammenfassung



- Tests sind unglaublich wichtig.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.
- Ich denke, für die meisten von uns sah unsere Implementierung der `sqrt`-Funktion eigentlich ganz gut aus.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.
- Ich denke, für die meisten von uns sah unsere Implementierung der `sqrt`-Funktion eigentlich ganz gut aus.
- Ja, OK, es war klar, das negative Zahlen ein Problem wären ... aber das heben wir uns auf für wenn wir über `Exceptions` lernen.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.
- Ich denke, für die meisten von uns sah unsere Implementierung der `sqrt`-Funktion eigentlich ganz gut aus.
- Ja, OK, es war klar, das negative Zahlen ein Problem wären ... aber das heben wir uns auf für wenn wir über `Exceptions` lernen.
- Ich glaube aber nicht, dass viele von uns die Sache mit `0.0` gesehen haben.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.
- Ich denke, für die meisten von uns sah unsere Implementierung der `sqrt`-Funktion eigentlich ganz gut aus.
- Ja, OK, es war klar, das negative Zahlen ein Problem wären ... aber das heben wir uns auf für wenn wir über `Exceptions` lernen.
- Ich glaube aber nicht, dass viele von uns die Sache mit `0.0` gesehen haben.
- Und ich bezweifle, dass viele das Problem mit `inf` haben kommen sehen.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.
- Ich denke, für die meisten von uns sah unsere Implementierung der `sqrt`-Funktion eigentlich ganz gut aus.
- Ja, OK, es war klar, das negative Zahlen ein Problem wären ... aber das heben wir uns auf für wenn wir über `Exceptions` lernen.
- Ich glaube aber nicht, dass viele von uns die Sache mit `0.0` gesehen haben.
- Und ich bezweifle, dass viele das Problem mit `inf` haben kommen sehen.
- Ich glaube sogar, dass einigen von uns `inf` und `nan` gar nicht auf dem Radar hatten, bis wir mit dem Nachdenken über mögliche Eingabewerte für unsere Tests angefangen haben.

Zusammenfassung



- Tests sind unglaublich wichtig.
- Wir haben das hier selbst gesehen.
- Ich denke, für die meisten von uns sah unsere Implementierung der `sqrt`-Funktion eigentlich ganz gut aus.
- Ja, OK, es war klar, das negative Zahlen ein Problem wären ... aber das heben wir uns auf für wenn wir über `Exceptions` lernen.
- Ich glaube aber nicht, dass viele von uns die Sache mit `0.0` gesehen haben.
- Und ich bezweifle, dass viele das Problem mit `inf` haben kommen sehen.
- Ich glaube sogar, dass einigen von uns `inf` und `nan` gar nicht auf dem Radar hatten, bis wir mit dem Nachdenken über mögliche Eingabewerte für unsere Tests angefangen haben.
- So oder so: Wir können extrem viel von Tests profitieren.

谢谢您们！

Thank you!

Vielen Dank!



References I

[1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[2] (siehe S. 233, 244).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[1] (siehe S. 233, 244).

[3] 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. 244, 245).

[4] 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. 244).

[5] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 15–23, 246).

[6] Kent L. Beck und Erich Gamma. "Test-Infected: Programmers Love Writing Tests". In: *More Java Gems*. Hrsg. von Dwight Deugo. New York, NY, USA: Cambridge University Press (CUP), Jan. 2000, S. 357–376. ISBN: 978-0-521-77477-2. doi:10.1017/CBO9780511550881.029. URL: http://members.pingnet.ch/gamma/junit.htm (besucht am 2025-09-05) (siehe S. 24–33).

[7] 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. 246).

[8] 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. 243).

[9] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: 978-0-321-35668-0 (siehe S. 244).

[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. 244).

[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. 244).

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. 243).

[13] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 244, 245).

[14] Noureddine Chabini und Rachid Beguenane. "FPGA-Based Designs of the Factorial Function". In: *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'2022)*. 18.–20. Sep. 2022, Halifax, NS, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2022, S. 16–20. ISBN: 978-1-6654-8432-9. doi:10.1109/CCECE49351.2022.9918302 (siehe S. 246).

[15] 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. 218–223, 245).

[16] 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. 245).

[17] 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. 245).

[18] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 245).

[19] 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. 245).

[20] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 245).

[21] 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. 246).

References III

[22] 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. 59–66, 244).

[23] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 243).

[24] Jacques Dutka. "The Early History of the Factorial Function". *Archive for History of Exact Sciences* 43(3):225–249, Sep. 1991. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0003-9519. doi:10.1007/BF00389433. Communicated by Umberto Bottazzini (siehe S. 246).

[25] 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. 244).

[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. 244).

[27] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy und Jignesh M. Patel. "SQLite: Past, Present, and Future". *Proceedings of the VLDB Endowment (PVLDB)* 15(12):3535–3547, Aug. 2022. Irvine, CA, USA: Very Large Data Bases Endowment Inc. ISSN: 2150-8097. doi:10.14778/3554821.3554842. URL: https://www.vldb.org/pvldb/vol15/p3535-gaffney.pdf (besucht am 2025-01-12). All papers in this issue were presented at the *48th International Conference on Very Large Data Bases (VLDB 2022)*, 9 5-9, 2022, hybrid/Sydney, NSW, Australia (siehe S. 218–223, 245).

[28] 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. 180–194).

[29] 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. 245).

[30] 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. 245).

References IV

[31] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 244).

[32] 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. 244, 245).

[33] D. Richard Hipp u. a. "How SQLite Is Tested". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 13. März 2024. URL: https://sqlite.org/testing.html (besucht am 2025-01-12) (siehe S. 218–223).

[34] D. Richard Hipp u. a. "Well-Known Users of SQLite". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 2. Jan. 2023. URL: https://www.sqlite.org/famous.html (besucht am 2025-01-12) (siehe S. 218–223, 245).

[35] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 244).

[36] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 245).

[37] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (besucht am 2024-08-23) (siehe S. 244).

[38] 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. 68–78, 244).

[39] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 245).

References V

[40] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 244).

[41] Moritz Lenz. *Python Continuous Integration and Delivery: A Concise Guide with Examples*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4281-0 (siehe S. 243).

[42] 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. 243).

[43] Marc Loy, Patrick Niemeyer und Daniel Leuck. *Learning Java*. 5. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2020. ISBN: 978-1-4920-5627-0 (siehe S. 244).

[44] Peter Luschny. *A New Kind of Factorial Function*. Highland Park, NJ, USA: The OEIS Foundation Inc., 4. Okt. 2015. URL: https://oeis.org/A000142/a000142.pdf (besucht am 2024-09-29) (siehe S. 246).

[45] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 244).

[46] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 244).

[47] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 245).

[48] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: https://docs.astral.sh/ruff (besucht am 2024-08-23) (siehe S. 245).

[49] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISSN: 1046-1698. ISBN: 978-1-55860-456-8 (siehe S. 245).

[50] Carl Meyer. *Python Virtual Environments*. Python Enhancement Proposal (PEP) 405. Beaverton, OR, USA: Python Software Foundation (PSF), 13. Juni 2011–24. Mai 2012. URL: https://peps.python.org/pep-0405 (besucht am 2024-12-25) (siehe S. 246).

References VI

[51] 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. 243).

[52] 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. 244).

[53] 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. 15–23, 246).

[54] 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. 59–66, 244).

[55] 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. 15–23, 246).

[56] 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. 243).

[57] 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. 15–23, 59–66, 244, 246).

[58] 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. 243).

[59] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 244).

References VII

[60] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (besucht am 2024-06-29) (siehe S. 243).

[61] 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. 243).

[62] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 244).

[63] 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. 244).

[64] 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. 243).

[65] 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. 15–23, 59–66, 246).

[66] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: 978-1-4503-9475-8. doi:10.1145/3551349.3559497 (siehe S. 244).

[67] 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. 244).

[68] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 243).

References VIII

[69] 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. 245).

[70] *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 2025. URL: https://sqlite.org (besucht am 2025-04-24) (siehe S. 245).

[71] "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. 245).

[72] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Teach Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 240, 245).

[73] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of [72] (siehe S. 245).

[74] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 245).

[75] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 244).

[76] 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. 15–23, 59–66, 246).

[77] 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. 244).

References IX

[78] 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. 243, 246).

[79] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 244).

[80] "Virtual Environments and Packages". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 12. URL: https://docs.python.org/3/tutorial/venv.html (besucht am 2024-12-24) (siehe S. 246).

[81] 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. 243, 245).

[82] Thomas Weise (汤卫思). *Programming with C*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2026. URL: https://thomasweise.github.io/programmingWithC (besucht am 2026-05-10) (siehe S. 243).

[83] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 244, 245).

[84] *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. 245).

[85] James A. Whittaker. "What Is Software Testing? And Why Is It So Hard?" *IEEE Software* 17(1):70–79, Jan.–Feb. 2000. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/52.819971. Practice Tutorial (siehe S. 59–66).

[86] 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. 244).

[87] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 244).

References X

[88] Marianne Winslett und Vanessa Braganholo. "Richard Hipp Speaks Out on SQLite". *ACM SIGMOD Record* 48(2):39–46, Juni 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5808. doi:10.1145/3377330.3377338 (siehe S. 218–223, 245).

[89] 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. 243).

[90] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 243).

Glossary (in English) I

Bash is a shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[12,51,90]. Learn more at https://www.gnu.org/software/bash.

C is a programming language, which is very successful in system programming situations[23,60,82].

CI *Continuous Integration* is a software development process where developers integrate new code into a codebase hosted in a Version Control System (VCS), after which automated tools run an automated build process including code analysis (such as linters) and unit test execution[41]. If the build succeeds and no errors or problems with the code are, the code may automatically be deployed (if the CI system is configured to do so).

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

client-server architecture is a system design where a central server receives requests from one or multiple clients[8,42,56,61,64]. These requests and responses are usually sent over network connections. A typical example of such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[81].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[89].

Git is a distributed Version Control System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[68,78]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[58,78]. Learn more at https://github.com.

Glossary (in English) II

IT	information technology
Java	is another very successful programming language, with roots in the C family of languages[9,43].
LAMP Stack	A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[13,32].
linter	A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles[37,66]. Ruff is an example of a linter used in the Python world.
Linux	is the leading open source operating system, i.e., a free alternative to Microsoft Windows[3,31,67,77,79]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.
MariaDB	An open source relational database management system that has forked off from MySQL[1,2,4,25,46,62]. See https://mariadb.org for more information.
Microsoft Windows	is a commercial proprietary operating system[11]. 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.
MySQL	An open source relational database management system[10,25,63,75,86]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.
PostgreSQL	An open source object-relational DBMS[26,52,59,75]. 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[22,38,54,57,87]. Learn more at https://pytest.org.
Python	The Python programming language[35,40,45,83], i.e., what you will learn about in our book[83]. Learn more at https://python.org.

Glossary (in English) III



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[17,29,30,69,74,81,84].

Ruff is a linter and code formatting tool for Python[47,48]. Learn more at `https://docs.astral.sh/ruff` or in [83].

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples of servers are web servers[13] 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“[39].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[15,18–20,36,49,71–74]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[71].

SQLite is an relational DBMS which runs as in-process library that works directly on files as opposed to the client-server architecture used by other common DBMSes. It is the most wide-spread SQL-based DB in use today, installed in nearly every smartphone, computer, web browser, television, and automobile[15,27,34,88]. Learn more at `https://sqlite.org`[70].

terminal A terminal is a text-based window where you can enter commands and execute them[3,16]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

Ubuntu is a variant of the open source operating system Linux[16,32]. 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`.

Glossary (in English) IV

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[5,53,55,57,65,76]. 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 code[78]. 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.

virtual environment
: A virtual environment is a directory that contains a local Python installation[50,80]. It comes with its own package installation directory. Multiple different virtual environments can be installed on a system. This allows different applications to use different versions of the same packages without conflict, because we can simply install these applications into different virtual environments.

WWW
: World Wide Web[7,21]

$i!$
: The factorial $a!$ of a natural number $a \in \mathbb{N}_1$ is the product of all positive natural numbers less than or equal to a, i.e., $a! = 1 * 2 * 3 * 4 * \cdots * (a-1) * a$[14,24,44].

$\mathbb{N}_0$
: the set of the natural numbers *including* 0, i.e., 0, 1, 2, 3, and so on. It holds that $\mathbb{N}_0 \subset \mathbb{Z}$ and that $\mathbb{N}_0 = \mathbb{N}_1 \cup \{0\}$.

$\mathbb{N}_1$
: the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N}_1 \subset \mathbb{Z}$ and that $\mathbb{N}_1 = \mathbb{N}_0 \setminus \{0\}$.

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

Glossary (in English) V

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