合肥大學
HEFEI UNIVERSITY



Programming with Python

31. Ausnahmen auslösen

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: 2026-11-05

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. Inkorrekte Eingabedaten
3. Ausnahmen auslösen
4. Built-In Exceptions
5. Zusammenfassung



Einleitung

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat. Mit gründlichem Unit Testen können wir die Wahrscheinlichkeit für Fehler reduzieren, aber wir können sie nie ganz ausschließen.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat. Mit gründlichem Unit Testen können wir die Wahrscheinlichkeit für Fehler reduzieren, aber wir können sie nie ganz ausschließen.
  2. Auf der anderen Seite existiert ja unser Programm nicht für sich alleine, losgelöst.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat. Mit gründlichem Unit Testen können wir die Wahrscheinlichkeit für Fehler reduzieren, aber wir können sie nie ganz ausschließen.
  2. Auf der anderen Seite existiert ja unser Programm nicht für sich alleine, losgelöst. Es bekommt Eingabedaten, vielleicht vom Benutzer, vielleicht von anderen Programmen.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat. Mit gründlichem Unit Testen können wir die Wahrscheinlichkeit für Fehler reduzieren, aber wir können sie nie ganz ausschließen.
  2. Auf der anderen Seite existiert ja unser Programm nicht für sich alleine, losgelöst. Es bekommt Eingabedaten, vielleicht vom Benutzer, vielleicht von anderen Programmen. Vielleicht sind die ja falsch.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat. Mit gründlichem Unit Testen können wir die Wahrscheinlichkeit für Fehler reduzieren, aber wir können sie nie ganz ausschließen.
  2. Auf der anderen Seite existiert ja unser Programm nicht für sich alleine, losgelöst. Es bekommt Eingabedaten, vielleicht vom Benutzer, vielleicht von anderen Programmen. Vielleicht sind die ja falsch.
- Diese beiden Probleme zusammen sind Situationen, die wir nicht erwartet haben.

Einleitung

- Bisher haben wir uns hauptsächlich darauf konzentriert, korrekten Kode zu schreiben.
- Wir versuchen Kode ohne Fehler zu schreiben.
- Wenn wir unsere Programme ausführen, dann gibt es zwei Dinge, die schief gehen können:
  1. Auf der einen Seite können wir nie sicher sein, dass unser Kode wirklich keine Fehler hat. Je größer das Programm, desto wahrscheinlicher, dass es Fehler hat. Mit gründlichem Unit Testen können wir die Wahrscheinlichkeit für Fehler reduzieren, aber wir können sie nie ganz ausschließen.
  2. Auf der anderen Seite existiert ja unser Programm nicht für sich alleine, losgelöst. Es bekommt Eingabedaten, vielleicht vom Benutzer, vielleicht von anderen Programmen. Vielleicht sind die ja falsch.
- Diese beiden Probleme zusammen sind Situationen, die wir nicht erwartet haben.
- Es sind Ausnahmen (EN: *Exceptions*).

Einleitende Beispiele



- Wir haben schon viele solche Ausnahmesituationen kennengelernt.

Einleitende Beispiele



- Wir haben schon viele solche Ausnahmesituationen kennengelernt:
  - Ein auf ein Zeichen an einem Index hinter dem Ende eines Strings zuzugreifen führt zu einem `IndexError`.

Einleitende Beispiele



- Wir haben schon viele solche Ausnahmesituationen kennengelernt:
  - Ein auf ein Zeichen an einem Index hinter dem Ende eines Strings zuzugreifen führt zu einem `IndexError`.
  - Der Versuch, ein Tupel zu verändern, scheitert mit einem `TypeError`.

Einleitende Beispiele



- Wir haben schon viele solche Ausnahmesituationen kennengelernt:
  - Ein auf ein Zeichen an einem Index hinter dem Ende eines Strings zuzugreifen führt zu einem `IndexError`.
  - Der Versuch, ein Tupel zu verändern, scheitert mit einem `TypeError`.
  - Wenn wir versuchen, `(10 ** 400)* 1.0` zu berechnen, so bekommen wir einen `OverflowError`, weil 10^{400} zu groß für den Datentyp `float` ist.

Einleitende Beispiele



- Wir haben schon viele solche Ausnahmesituationen kennengelernt:
  - Ein auf ein Zeichen an einem Index hinter dem Ende eines Strings zuzugreifen führt zu einem `IndexError`.
  - Der Versuch, ein Tupel zu verändern, scheitert mit einem `TypeError`.
  - Wenn wir versuchen, `(10 ** 400)* 1.0` zu berechnen, so bekommen wir einen `OverflowError`, weil 10^{400} zu groß für den Datentyp `float` ist.
- Solche Ausnahmen können durch Programmierfehler ausgelöst werden.

Einleitende Beispiele



- Wir haben schon viele solche Ausnahmesituationen kennengelernt:
  - Ein auf ein Zeichen an einem Index hinter dem Ende eines Strings zuzugreifen führt zu einem `IndexError`.
  - Der Versuch, ein Tupel zu verändern, scheitert mit einem `TypeError`.
  - Wenn wir versuchen, `(10 ** 400)* 1.0` zu berechnen, so bekommen wir einen `OverflowError`, weil 10^{400} zu groß für den Datentyp `float` ist.
- Solche Ausnahmen können durch Programmierfehler ausgelöst werden.
- Sie könnten aber auch durch falsche Eingabedaten an unser Programm entstehen.

Inkorrekte Eingabedaten



Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41].

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen. z. B. unsere `factorial`-Funktion von früher erwartet eine Ganzzahl als Input.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen. z. B. unsere `factorial`-Funktion von früher erwartet eine Ganzzahl als Input. Wenn jemand `2.4` anstelle von `2` eingeben würde, könnten wir einfach mit `2` rechnen und das entsprechende Ergebnis zurückliefern.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen. z. B. unsere `factorial`-Funktion von früher erwartet eine Ganzzahl als Input. Wenn jemand `2.4` anstelle von `2` eingeben würde, könnten wir einfach mit `2` rechnen und das entsprechende Ergebnis zurückliefern. Unsere `sqrt`-Funktion liefert z. B. `0.0` als Ergebnis, wenn jemand eine negative Zahl eingibt.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen. z. B. unsere `factorial`-Funktion von früher erwartet eine Ganzzahl als Input. Wenn jemand `2.4` anstelle von `2` eingeben würde, könnten wir einfach mit `2` rechnen und das entsprechende Ergebnis zurückliefern. Unsere `sqrt`-Funktion liefert z. B. `0.0` als Ergebnis, wenn jemand eine negative Zahl eingibt.
  3. Wir können die Funktion beschützen, in dem wir eine Ausnahme auslösen.[11,32,45] (EN: *raise an* `Exception`)

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen. z. B. unsere `factorial`-Funktion von früher erwartet eine Ganzzahl als Input. Wenn jemand `2.4` anstelle von `2` eingeben würde, könnten wir einfach mit `2` rechnen und das entsprechende Ergebnis zurückliefern. Unsere `sqrt`-Funktion liefert z. B. `0.0` als Ergebnis, wenn jemand eine negative Zahl eingibt.
  3. Wir können die Funktion beschützen, in dem wir eine Ausnahme auslösen.[11,32,45] (EN: *raise an `Exception`*)
- Letzteres ist, was Python oft macht.

Inkorrekte Eingabedaten



- Es stellt sich die Frage: Was soll man überhaupt machen, wenn eine Funktion fehlerhafte Eingabedaten bekommt?
- Dafür gibt es drei grundlegende Taktiken
  1. Wir ignorieren einfach, dass die Daten falsch sind, und rechnen normal mit ihnen. Wenn die Eingabedaten von unserem Programm falsch sind, dann produziert es eben auch falsche Ausgabedaten. Diese Taktik nenn man Garbage In–Garbage Out (GIGO)[41]. Das Konvertieren vom Integer `10 ** 400` zu einem `float` könnte z. B. einfach `inf` liefern.
  2. Wir versuchen, den Input zu bereinigen. z. B. unsere `factorial`-Funktion von früher erwartet eine Ganzzahl als Input. Wenn jemand `2.4` anstelle von `2` eingeben würde, könnten wir einfach mit `2` rechnen und das entsprechende Ergebnis zurückliefern. Unsere `sqrt`-Funktion liefert z. B. `0.0` als Ergebnis, wenn jemand eine negative Zahl eingibt.
  3. Wir können die Funktion beschützen, in dem wir eine Ausnahme auslösen.[11,32,45] (EN: *raise an `Exception`*)
- Letzteres ist, was Python oft macht. z. B. *könnte* es einfach ignorieren, wenn jemand ein Tupel verändern will. Stattdessen löst es einen `TypeError` aus.

Inkorrekte Eingabedaten: Ausnahme auslösen



- Python löst oftmals Ausnahmen, so genannte `Exceptions` aus, wenn eine Python-Funktion mit fehlerhaften Eingabedaten aufgerufen wird, oder wenn verbotene Aktionen durchgeführt werden.

Inkorrekte Eingabedaten: Ausnahme auslösen



- Python löst oftmals Ausnahmen, so genannte `Exceptions` aus, wenn eine Python-Funktion mit fehlerhaften Eingabedaten aufgerufen wird, oder wenn verbotene Aktionen durchgeführt werden.
- Ich bin ebenfalls ein Fan dieses Ansatzes.

Inkorrekte Eingabedaten: Ausnahme auslösen



- Python löst oftmals Ausnahmen, so genannte `Exceptions` aus, wenn eine Python-Funktion mit fehlerhaften Eingabedaten aufgerufen wird, oder wenn verbotene Aktionen durchgeführt werden.
- Ich bin ebenfalls ein Fan dieses Ansatzes.
- Und die Python-Dokumentation ist es auch.

Inkorrekte Eingabedaten: Ausnahme auslösen

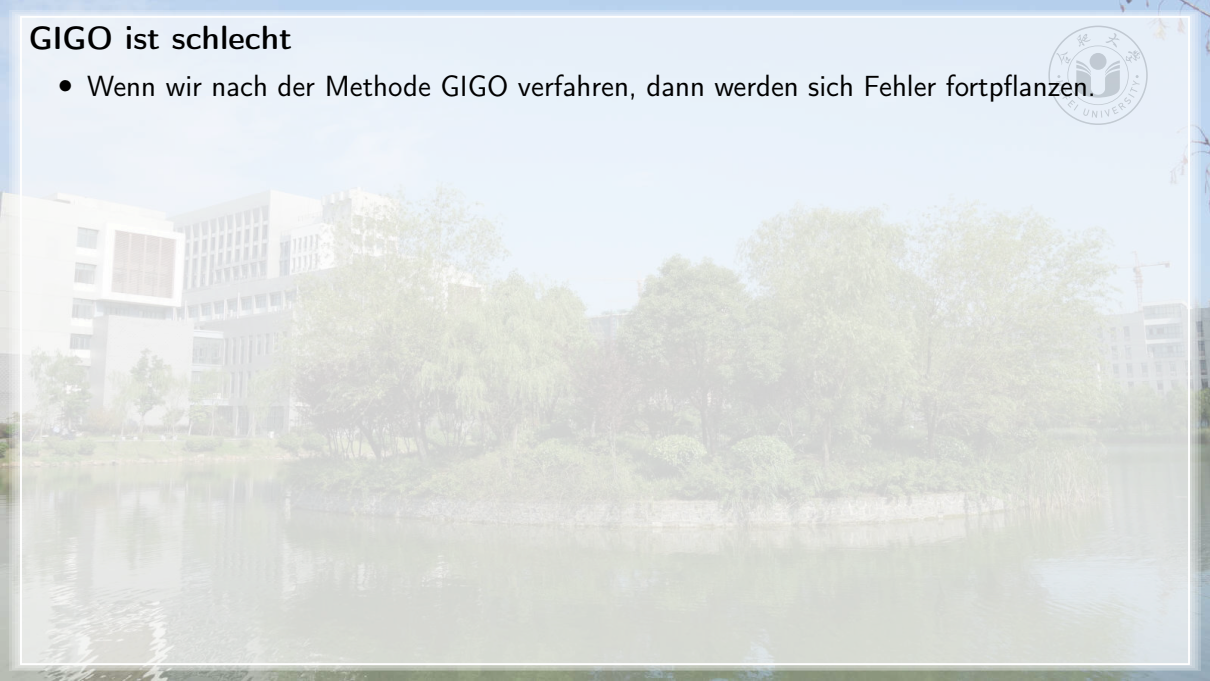


- Python löst oftmals Ausnahmen, so genannte `Exceptions` aus, wenn eine Python-Funktion mit fehlerhaften Eingabedaten aufgerufen wird, oder wenn verbotene Aktionen durchgeführt werden.
- Ich bin ebenfalls ein Fan dieses Ansatzes.
- Und die Python-Dokumentation ist es auch:

> *Errors should never pass silently. Unless explicitly silenced.*
>
> — *Tim Peters [40], 2004*

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.



GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.
- Ein fehlerhaftes Ergebnis kann Teil von Entscheidungen und Designs im echten Leben werden.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.
- Ein fehlerhaftes Ergebnis kann Teil von Entscheidungen und Designs im echten Leben werden.
- Und selbst wenn wir das merken, dann wird es extrem schwer, herauszufinden wo denn der Fehler in der langen Kette von Berechnungen und Funktionsaufrufen passiert ist.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.
- Ein fehlerhaftes Ergebnis kann Teil von Entscheidungen und Designs im echten Leben werden.
- Und selbst wenn wir das merken, dann wird es extrem schwer, herauszufinden wo denn der Fehler in der langen Kette von Berechnungen und Funktionsaufrufen passiert ist.
- Vielleicht wurden falsche Ergebnisse in Dateien gespeichert, wodurch andere Programme dann später abstürzen.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.
- Ein fehlerhaftes Ergebnis kann Teil von Entscheidungen und Designs im echten Leben werden.
- Und selbst wenn wir das merken, dann wird es extrem schwer, herauszufinden wo denn der Fehler in der langen Kette von Berechnungen und Funktionsaufrufen passiert ist.
- Vielleicht wurden falsche Ergebnisse in Dateien gespeichert, wodurch andere Programme dann später abstürzen.
- Und *später* könnte eine Woche später sein.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.
- Ein fehlerhaftes Ergebnis kann Teil von Entscheidungen und Designs im echten Leben werden.
- Und selbst wenn wir das merken, dann wird es extrem schwer, herauszufinden wo denn der Fehler in der langen Kette von Berechnungen und Funktionsaufrufen passiert ist.
- Vielleicht wurden falsche Ergebnisse in Dateien gespeichert, wodurch andere Programme dann später abstürzen.
- Und *später* könnte eine Woche später sein.
- Und *andere Programme* könnten Programme sein, die von jemand anders in einer anderen Abteilung ausgeführt werden.

GIGO ist schlecht

- Wenn wir nach der Methode GIGO verfahren, dann werden sich Fehler fortpflanzen.
- Vielleicht werden die Ergebnisse unserer Funktion als Parameter in eine andere Funktion gefüttert, deren Ergebnisse dann wieder in eine andere Funktion eingehen, und so weiter.
- Ein Fehler könnte dann später irgendwo anders zu einem Crash führen.
- Wenn Funktionen, die nach der GIGO-Idee programmiert sind mit solchen kombiniert werden, die ihre Eingabedaten bereinigen, dann könnte ein Fehler ganz unbemerkt bleiben.
- Ein fehlerhaftes Ergebnis kann Teil von Entscheidungen und Designs im echten Leben werden.
- Und selbst wenn wir das merken, dann wird es extrem schwer, herauszufinden wo denn der Fehler in der langen Kette von Berechnungen und Funktionsaufrufen passiert ist.
- Vielleicht wurden falsche Ergebnisse in Dateien gespeichert, wodurch andere Programme dann später abstürzen.
- Und *später* könnte eine Woche später sein.
- Und *andere Programme* könnten Programme sein, die von jemand anders in einer anderen Abteilung ausgeführt werden.
- Viel Glück dann, das Stück Kode zu finden, wo der Fehler entstanden ist.

Eingabedaten bereinigen ist schlecht



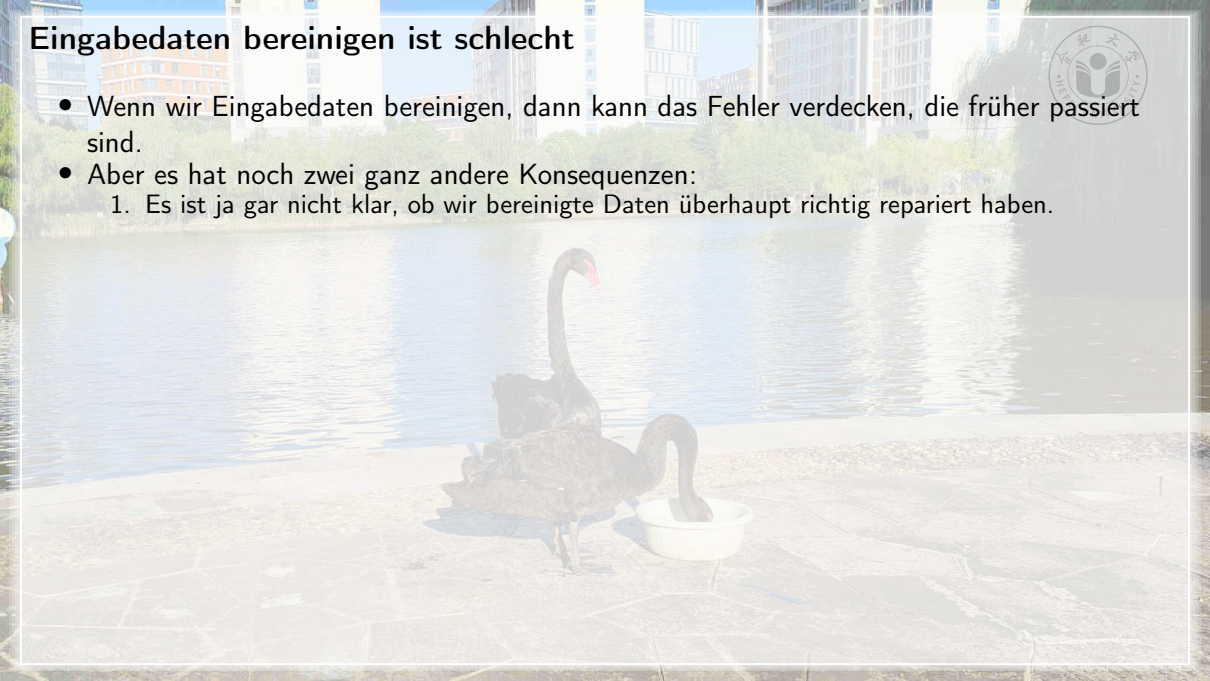
- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen.

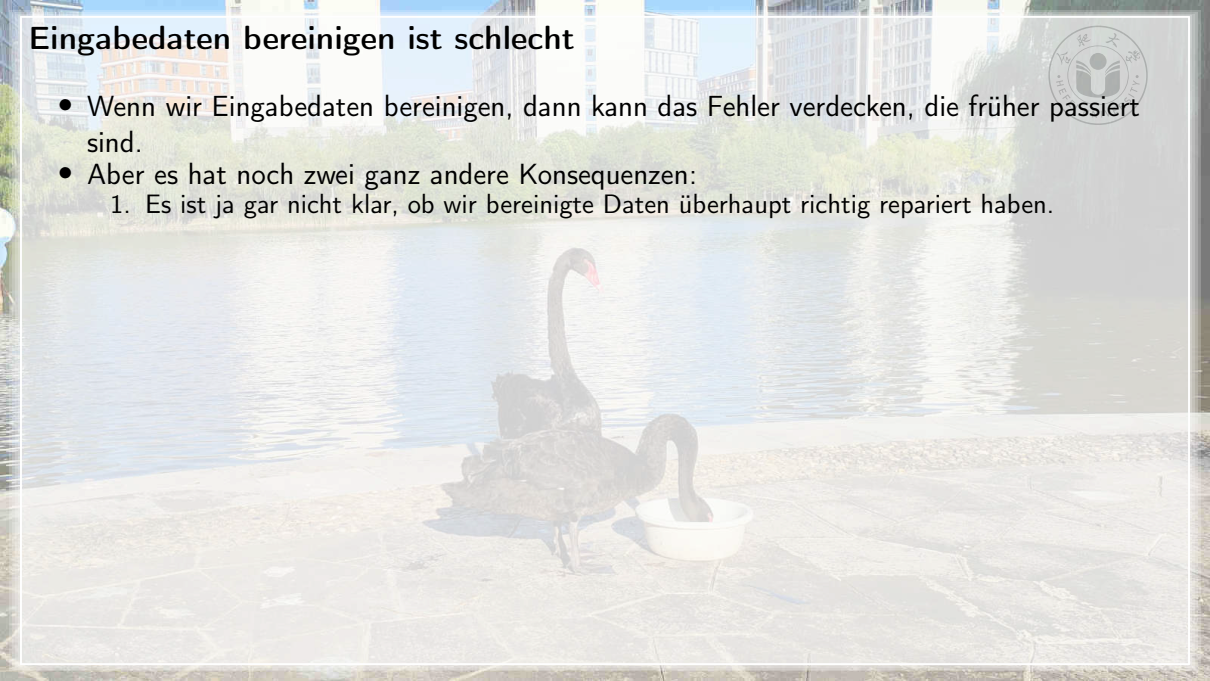


Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.
     - Jetzt bekommt unsere Funktion den fehlerhaften Wert `5.5` für `x`.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.
     - Jetzt bekommt unsere Funktion den fehlerhaften Wert `5.5` für `x`.
     - Sollen wir das zu `5` oder `6` reparieren?

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.
     - Jetzt bekommt unsere Funktion den fehlerhaften Wert `5.5` für `x`.
     - Sollen wir das zu `5` oder `6` reparieren?
     - Jede der beiden Möglichkeiten könnte richtig sein.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.
     - Jetzt bekommt unsere Funktion den fehlerhaften Wert `5.5` für `x`.
     - Sollen wir das zu `5` oder `6` reparieren?
     - Jede der beiden Möglichkeiten könnte richtig sein.
     - Beide könnten auch falsch sein.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.
     - Jetzt bekommt unsere Funktion den fehlerhaften Wert `5.5` für `x`.
     - Sollen wir das zu `5` oder `6` reparieren?
     - Jede der beiden Möglichkeiten könnte richtig sein.
     - Beide könnten auch falsch sein.
     - Sollen wir vielleicht einfach immer mit `int(x)` rechnen?

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
     - Nehmen wir an, wir haben eine Funktion `compute` mit einem Parameter `x`, der eine Ganzzahl seien soll.
     - Jetzt bekommt unsere Funktion den fehlerhaften Wert `5.5` für `x`.
     - Sollen wir das zu `5` oder `6` reparieren?
     - Jede der beiden Möglichkeiten könnte richtig sein.
     - Beide könnten auch falsch sein.
     - Sollen wir vielleicht einfach immer mit `int(x)` rechnen?
     - Es ist schwer, eine feste Regel zu definieren, die immer den richtigen Wert nimmt.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.
     - Natürlich funktioniert `int(x)` auch mit `"12"` als Input und liefert dann `12`.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.
     - Natürlich funktioniert `int(x)` auch mit `"12"` als Input und liefert dann `12`.
     - Jetzt gibt es also plötzlich Leute, die unsere Funktion mit Strings benutzen.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.
     - Natürlich funktioniert `int(x)` auch mit `"12"` als Input und liefert dann `12`.
     - Jetzt gibt es also plötzlich Leute, die unsere Funktion mit Strings benutzen.
     - Und *wir werden diesen Use-Case jetzt immer weiter unterstützen müssen.*

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.
     - Natürlich funktioniert `int(x)` auch mit `"12"` als Input und liefert dann `12`.
     - Jetzt gibt es also plötzlich Leute, die unsere Funktion mit Strings benutzen.
     - Und *wir werden diesen Use-Case jetzt immer weiter unterstützen müssen.*
     - Denn jetzt gibt es Kode in anderen Programmen, der sich darauf verlässt, dass das geht.

Eingabedaten bereinigen ist schlecht

- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Bleiben wir bei dem Beispiel der Funktion `compute` mit dem Ganzzahl-Parameter `x`.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.
     - Natürlich funktioniert `int(x)` auch mit `"12"` als Input und liefert dann `12`.
     - Jetzt gibt es also plötzlich Leute, die unsere Funktion mit Strings benutzen.
     - Und *wir werden diesen Use-Case jetzt immer weiter unterstützen müssen.*
     - Denn jetzt gibt es Kode in anderen Programmen, der sich darauf verlässt, dass das geht.
     - Wenn wir es uns jetzt anders überlegen und keine Strings mehr erlauben... ...dann wird dieser Kode abstürzen.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
     - Nehmen wir an, wir haben uns entschieden, einfach immer mit `int(x)` zu rechnen.
     - Damit haben konvertieren wir alle endlichen `floats` zu `ints` und bereinigen so die Eingabewerte zu unserem gewünschten Datentyp.
     - Wir hielten das für eine gute Idee.
     - Jetzt bemerken wir, dass es plötzlich Leute gibt, die unsere Funktion so aufrufen: `compute("12")`.
     - Natürlich funktioniert `int(x)` auch mit `"12"` als Input und liefert dann `12`.
     - Jetzt gibt es also plötzlich Leute, die unsere Funktion mit Strings benutzen.
     - Und *wir werden diesen Use-Case jetzt immer weiter unterstützen müssen.*
     - Denn jetzt gibt es Kode in anderen Programmen, der sich darauf verlässt, dass das geht.
     - Wenn wir es uns jetzt anders überlegen und keine Strings mehr erlauben... ...dann wird dieser Kode abstürzen.
     - Das Bereinigen von Eingabedaten lädt also geradezu zu schlampigem Programmieren ein.

Eingabedaten bereinigen ist schlecht



- Wenn wir Eingabedaten bereinigen, dann kann das Fehler verdecken, die früher passiert sind.
- Aber es hat noch zwei ganz andere Konsequenzen:
  1. Es ist ja gar nicht klar, ob wir bereinigte Daten überhaupt richtig repariert haben.
  2. Eingabedaten bereinigen kann dazu führen, dass andere Programmierer unseren Kode falsch verwenden.
- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.

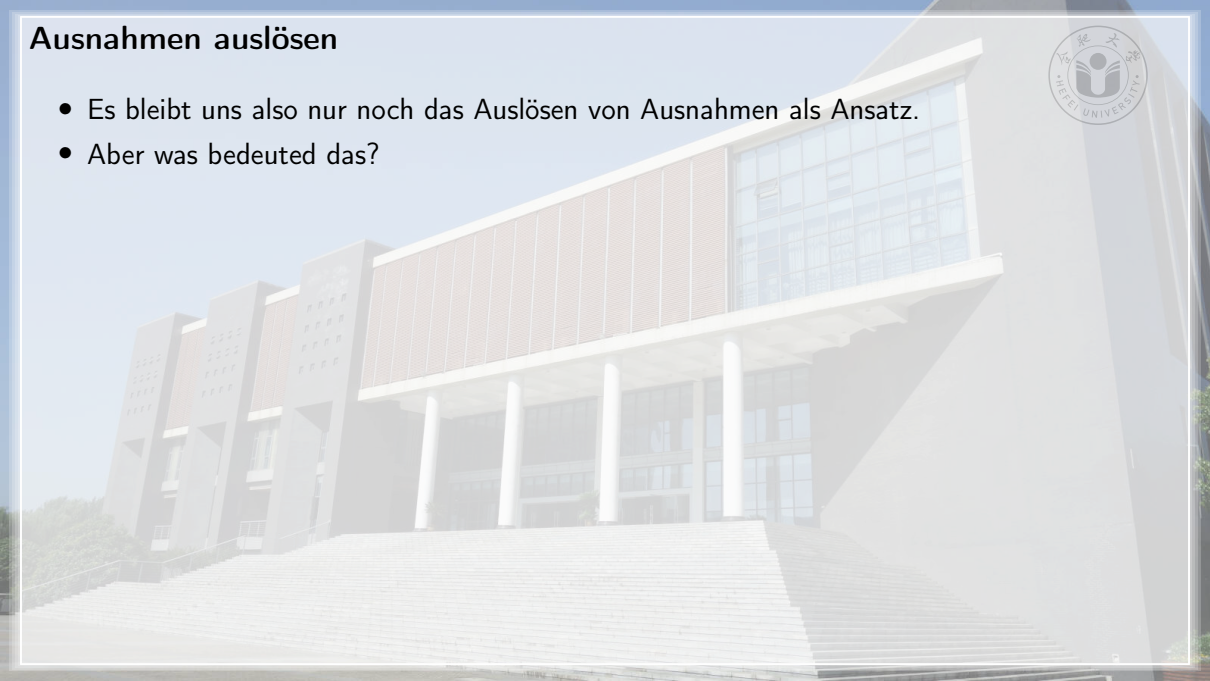
Ausnahmen auslösen

- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.



Ausnahmen auslösen

- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?



Ausnahmen auslösen

- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge.

Ausnahmen auslösen



- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge
  1. Informationen über den Fehler und Informationen über den aktuellen Ausführungszustand des Programs (z. B. die aktuelle Kode-Zeile, die Aufruf-Hierarchie) werden einem Objekt gespeichert (der `Exception`).

Ausnahmen auslösen

- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge
  1. Informationen über den Fehler und Informationen über den aktuellen Ausführungszustand des Programs (z. B. die aktuelle Kode-Zeile, die Aufruf-Hierarchie) werden einem Objekt gespeichert (der `Exception`).
  2. Der Kontrollfluss verlässt sofort den aktuellen Kodeblock sowie alle aufrufenden Kodeblöcke und Funktionen.

Ausnahmen auslösen



- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge
  1. Informationen über den Fehler und Informationen über den aktuellen Ausführungszustand des Programs (z. B. die aktuelle Kode-Zeile, die Aufruf-Hierarchie) werden einem Objekt gespeichert (der `Exception`).
  2. Der Kontrollfluss verlässt sofort den aktuellen Kodeblock sowie alle aufrufenden Kodeblöcke und Funktionen. Er springt in der Aufrufhierarchie so lange nach oben, bis er Kode erreicht, der die ausgelöste Ausnahme behandeln kann.

Ausnahmen auslösen



- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge
  1. Informationen über den Fehler und Informationen über den aktuellen Ausführungszustand des Programs (z. B. die aktuelle Kode-Zeile, die Aufruf-Hierarchie) werden einem Objekt gespeichert (der `Exception`).
  2. Der Kontrollfluss verlässt sofort den aktuellen Kodeblock sowie alle aufrufenden Kodeblöcke und Funktionen. Er springt in der Aufrufhierarchie so lange nach oben, bis er Kode erreicht, der die ausgelöste Ausnahme behandeln kann. Wenn kein solcher Kode existiert, dann wird der Prozess terminiert und ein Exit-Kode anders als `0` zurückgeliefert.

Ausnahmen auslösen



- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge
  1. Informationen über den Fehler und Informationen über den aktuellen Ausführungszustand des Programs (z. B. die aktuelle Kode-Zeile, die Aufruf-Hierarchie) werden einem Objekt gespeichert (der `Exception`).
  2. Der Kontrollfluss verlässt sofort den aktuellen Kodeblock sowie alle aufrufenden Kodeblöcke und Funktionen. Er springt in der Aufrufhierarchie so lange nach oben, bis er Kode erreicht, der die ausgelöste Ausnahme behandeln kann. Wenn kein solcher Kode existiert, dann wird der Prozess terminiert und ein Exit-Kode anders als `0` zurückgeliefert.
- In anderen Worten, eine Ausnahme auszulösen führt zum Verlassen des aktuellen Pfads des Kontrollflusses und zum Signalisieren eines Fehlers der entweder explizit behandelt werden muss, oder der andernfalls zum Beenden des Programmes führt.

Ausnahmen auslösen



- Es bleibt uns also nur noch das Auslösen von Ausnahmen als Ansatz.
- Aber was bedeuted das?
- Das Auslösen von Ausnamen, in (EN: *raising of* `Exceptions`) bedeutet zwei Dinge
  1. Informationen über den Fehler und Informationen über den aktuellen Ausführungszustand des Programs (z. B. die aktuelle Kode-Zeile, die Aufruf-Hierarchie) werden einem Objekt gespeichert (der `Exception`).
  2. Der Kontrollfluss verlässt sofort den aktuellen Kodeblock sowie alle aufrufenden Kodeblöcke und Funktionen. Er springt in der Aufrufhierarchie so lange nach oben, bis er Kode erreicht, der die ausgelöste Ausnahme behandeln kann. Wenn kein solcher Kode existiert, dann wird der Prozess terminiert und ein Exit-Kode anders als `0` zurückgeliefert.
- In anderen Worten, eine Ausnahme auszulösen führt zum Verlassen des aktuellen Pfads des Kontrollflusses und zum Signalisieren eines Fehlers der entweder explizit behandelt werden muss, oder der andernfalls zum Beenden des Programmes führt.
- Das ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist. Dadurch wird es viel einfacher, herauszufinden, was den Fehler ausgelöst hat, z. B. war es ein Programmierfehler oder fehlerhafte Daten?

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert. In einer fehlerhaften Situation oder wenn fehlerhafte Daten auftauchen, dann kann das Problem sich nicht aus dem aktuellen Kontext heraus fortpflanzen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert. In einer fehlerhaften Situation oder wenn fehlerhafte Daten auftauchen, dann kann das Problem sich nicht aus dem aktuellen Kontext heraus fortpflanzen. Eine Ausnahme wird ausgelöst, die den aktuellen Ausführungspfad verlässt und somit den verunreinigten Kontrollfluss beendet.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen. Sie können den Fehler nicht einfach implizit ignorieren.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen. Sie können den Fehler nicht einfach implizit ignorieren. Natürlich können sie das Ausnahme-Objekt abfangen und wegwerfen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen. Sie können den Fehler nicht einfach implizit ignorieren. Natürlich können sie das Ausnahme-Objekt abfangen und wegwerfen. Aber sie müssen es explizit und mit Absicht machen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen. Sie können den Fehler nicht einfach implizit ignorieren. Natürlich können sie das Ausnahme-Objekt abfangen und wegwerfen. Aber sie müssen es explizit und mit Absicht machen. Fehler können nicht unbeabsichtigt übersehen werden, denn dann würde die Ausnahme das Programm beenden.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“*

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“* Aber was ist schlimmer?

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“* Aber was ist schlimmer? Das ein Fehler das aktuelle Programm unerwartet zum Absturz bringt ... oder dass alle zukünftigen Ausgaben des Programms falsch sind, und zwar *unerkannt* falsch?

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“* Aber unentdeckte Fehler wären noch viel schlimmer...

Gute Praxis

Fehler sollten nicht ignoriert werden und Eingabedaten sollten nicht repariert werden.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“* Aber unentdeckte Fehler wären noch viel schlimmer...

Gute Praxis

Fehler sollten nicht ignoriert werden und Eingabedaten sollten nicht repariert werden. Stattdessen sollten Eingabedaten von Funktion, wo immer sinnvol, auf Gültigkeit geprüft werden.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“* Aber unentdeckte Fehler wären noch viel schlimmer...

Gute Praxis

Fehler sollten nicht ignoriert werden und Eingabedaten sollten nicht repariert werden. Stattdessen sollten Eingabedaten von Funktion, wo immer sinnvol, auf Gültigkeit geprüft werden. Fehlerhafte Eingabedaten sollten durch Fehlermeldungen signalisiert werden, die denn Kontrollfluss unterbrechen.

Ausnahmen auslösen ist gut



- Ausnahmen auszulösen ist meiner Meinung nach die beste Methode, um mit problematischen Situationen umzugehen weil:
  1. Es zeigt klar und explizit, dass ein Fehler passiert ist, wo er passiert ist, wann er passiert, und zu einem gewissen Grad, weshalb er passiert ist.
  2. Es verhindert, dass GIGO überhaupt passiert.
  3. Programmierer werden gezwungen, explizit mit der fehlerhaften Situation umzugehen.
  4. Man kann vielleicht sagen: *„Aber wenn wir die Ausnahme nicht explizit behandeln, dann stürzt das Programm doch ab! Das ist doch schlecht?“* Aber unentdeckte Fehler wären noch viel schlimmer...

Gute Praxis

Fehler sollten nicht ignoriert werden und Eingabedaten sollten nicht repariert werden. Stattdessen sollten Eingabedaten von Funktion, wo immer sinnvol, auf Gültigkeit geprüft werden. Fehlerhafte Eingabedaten sollten durch Fehlermeldungen signalisiert werden, die denn Kontrollfluss unterbrechen. `Exceptions` sollten so früh wie möglich ausgelöst werden und zwar immer, wenn eine unerwartete Situation eintritt.

Ausnahmen auslösen



Ausnahmen auslösen

- Ausnahmen auslösen, auf (EN: *raising exceptions*), ist sehr einfach.

```
"""
The syntax of raising an `Exception` in Python.

There are many different types of `Exception`s for different situations
in Python. For example, `ArithmeticError`, `OverflowError`,
`IndexError`, `TypeError`, and `ValueError`, to name a few.

They can be raised by writing `raise ExceptionType` without additional
information or by providing an error message string, like
`raise ExceptionType("This is the error message.")`.
"""

# Raise an `Exception` of type `TypeError` without an error message.
raise TypeError
```

Ausnahmen auslösen

- Ausnahmen auslösen, auf (EN: *raising exceptions*), ist sehr einfach.
- Python bietet uns viele verschiedene Typen von Exceptions an, die wir aber später diskutieren.

```
"""
The syntax of raising an `Exception` in Python.

There are many different types of `Exception`s for different situations
in Python. For example, `ArithmeticError`, `OverflowError`,
`IndexError`, `TypeError`, and `ValueError`, to name a few.

They can be raised by writing `raise ExceptionType` without additional
information or by providing an error message string, like
`raise ExceptionType("This is the error message.")`.
"""

# Raise an `Exception` of type `TypeError` without an error message.
raise TypeError
```

Ausnahmen auslösen

- Ausnahmen auslösen, auf (EN: *raising exceptions*), ist sehr einfach.
- Python bietet uns viele verschiedene Typen von Exceptions an, die wir aber später diskutieren.
- Wichtige Beispiele sind `ValueError` (wenn ein Parameter einen inkorrekten Wert hat) und `TypeError` (wenn ein Parameter einen falschen Typ hat).

```
"""
The syntax of raising an `Exception` in Python.

There are many different types of `Exception`s for different situations
in Python. For example, `ArithmeticError`, `OverflowError`,
`IndexError`, `TypeError`, and `ValueError`, to name a few.

They can be raised by writing `raise ExceptionType` without additional
information or by providing an error message string, like
`raise ExceptionType("This is the error message.")`.
"""

# Raise an `Exception` of type `TypeError` without an error message.
raise TypeError
```

Ausnahmen auslösen



- Python bietet uns viele verschiedene Typen von Exceptions an, die wir aber später diskutieren.
- Wichtige Beispiele sind `ValueError` (wenn ein Parameter einen inkorrekten Wert hat) und `TypeError` (wenn ein Parameter einen falschen Typ hat).
- Wenn Sie einen Fehler eines dieser Typen auslösen wollen, dann brauchen Sie nur `raise` gefolgt vom Exception-Typ zu schreiben.

```
"""
The syntax of raising an `Exception` in Python.

There are many different types of `Exception`s for different situations
in Python. For example, `ArithmeticError`, `OverflowError`,
`IndexError`, `TypeError`, and `ValueError`, to name a few.

They can be raised by writing `raise ExceptionType` without additional
information or by providing an error message string, like
`raise ExceptionType("This is the error message.")`.
"""

# Raise an `Exception` of type `TypeError` without an error message.
raise TypeError
```

Ausnahmen auslösen



- Wenn Sie einen Fehler eines dieser Typen auslösen wollen, dann brauchen Sie nur `raise` gefolgt vom Exception-Typ zu schreiben.
- Wenn Sie zusätzlich eine Fehlermeldung bereitstellen wollen, dann schreiben Sie diese als String in Klammern nach dem Exception-Typ.

```
"""
The syntax of raising an `Exception` in Python.

There are many different types of `Exception`s for different situations
in Python. For example, `ArithmeticError`, `OverflowError`,
`IndexError`, `TypeError`, and `ValueError`, to name a few.

They can be raised by writing `raise ExceptionType` without additional
information or by providing an error message string, like
`raise ExceptionType("This is the error message.")`.
"""

# Raise an `Exception` of type `TypeError` without an error message.
raise TypeError

# Raise an `Exception` of type `ValueError` with an error message.
raise ValueError("Cannot compute ln(-1)!")
```

Ausnahmen in Funktionen auslösen

- Oftmals wollen wir Ausnahmen in Funktionen auslösen.

```
"""Raising an `Exception` in a function, as explained in docstring."""

def my_func(x: int) -> None:
    """
    This is a function.

    :param x: a parameter
    :raises ValueError: if `x < 1`
    """
    if x < 1:
        raise ValueError("Invalid x!")
```

Ausnahmen in Funktionen auslösen



- Oftmals wollen wir Ausnahmen in Funktionen auslösen.
- Das geht genauso wie oben, aber wir sollten den Fehlertyp im Docstring der Funktion vermerken.

```
"""Raising an `Exception` in a function, as explained in docstring."""

def my_func(x: int) -> None:
    """
    This is a function.

    :param x: a parameter
    :raises ValueError: if `x < 1`
    """
    if x < 1:
        raise ValueError("Invalid x!")
```

Ausnahmen in Funktionen auslösen



- Oftmals wollen wir Ausnahmen in Funktionen auslösen.
- Das geht genauso wie oben, aber wir sollten den Fehlertyp im Docstring der Funktion vermerken.
- Das geht, in dem wir einen Eintrag `:raises ExceptionType: Erklärung` nach der Parameterlist anhängen.

```
"""Raising an `Exception` in a function, as explained in docstring."""

def my_func(x: int) -> None:
    """
    This is a function.

    :param x: a parameter
    :raises ValueError: if `x < 1`
    """
    if x < 1:
        raise ValueError("Invalid x!")
```

Ausnahmen in Funktionen auslösen



- Oftmals wollen wir Ausnahmen in Funktionen auslösen.
- Das geht genauso wie oben, aber wir sollten den Fehlertyp im Docstring der Funktion vermerken.
- Das geht, in dem wir einen Eintrag `:raises ExceptionType: Erklärung` nach der Parameterlist anhängen.

Gute Praxis

Jede Funktion die selbst explizit eine Ausnahme auslösen kann, muss das auch explizit in ihrem Docstring anmerken.

Ausnahmen in Funktionen auslösen



- Oftmals wollen wir Ausnahmen in Funktionen auslösen.
- Das geht genauso wie oben, aber wir sollten den Fehlertyp im Docstring der Funktion vermerken.
- Das geht, in dem wir einen Eintrag `:raises ExceptionType: Erklärung` nach der Parameterlist anhängen.

Gute Praxis

Jede Funktion die selbst explizit eine Ausnahme auslösen kann, muss das auch explizit in ihrem Docstring anmerken. Dafür schreibt man eine Notiz der Form `:raises ExceptionType: why` wobei `ExceptionType` mit dem Typ der Ausnahme und `why` mit einer kurzen Erklärung ersetzt werden.

Beispiel: Quadratwurzel (Original)

- Schauen wir uns ein Beispiel an, wie wir eine Ausnahme in unserem Kode auslösen können.

```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
```

Beispiel: Quadratwurzel (Original)

- Schauen wir uns ein Beispiel an, wie wir eine Ausnahme in unserem Kode auslösen können.
- Wir gehen dafür zurück zu unserer Funktion `sqrt` in Datei `my_math_3.py` aus Einheit 28.

```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
```

Beispiel: Quadratwurzel (Original)

- Schauen wir uns ein Beispiel an, wie wir eine Ausnahme in unserem Kode auslösen können.
- Wir gehen dafür zurück zu unserer Funktion `sqrt` in Datei `my_math_3.py` aus Einheit 28.
- Damals war uns aufgefallen, dass es einige Eingabewerte wie z. B. `inf`, `-inf`, `nan`, und `0.0` gibt, die eine besondere Behandlung erfordern.

```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
```

Beispiel: Quadratwurzel (Original)

- Schauen wir uns ein Beispiel an, wie wir eine Ausnahme in unserem Kode auslösen können.
- Wir gehen dafür zurück zu unserer Funktion `sqrt` in Datei `my_math_3.py` aus Einheit 28.
- Damals war uns aufgefallen, dass es einige Eingabewerte wie z. B. `inf`, `-inf`, `nan`, und `0.0` gibt, die eine besondere Behandlung erfordern.
- Wir hatten auch bemerkt, dass jemand kommen könnte und negative Zahlen als Argument an unsere `sqrt`-Funktion übergeben könnte.

```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
```

Beispiel: Quadratwurzel (Original)

- Wir gehen dafür zurück zu unserer Funktion `sqrt` in Datei `my_math_3.py` aus Einheit 28.
- Damals war uns aufgefallen, dass es einige Eingabewerte wie z. B. `inf`, `-inf`, `nan`, und `0.0` gibt, die eine besondere Behandlung erfordern.
- Wir hatten auch bemerkt, dass jemand kommen könnte und negative Zahlen als Argument an unsere `sqrt`-Funktion übergeben könnte.
- Wir hatten damals nicht die Möglichkeit, gegen so etwas vorzugehen.

```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
```

Beispiel: Quadratwurzel (Original)

- Wir gehen dafür zurück zu unserer Funktion `sqrt` in Datei `my_math_3.py` aus Einheit 28.
- Damals war uns aufgefallen, dass es einige Eingabewerte wie z. B. `inf`, `-inf`, `nan`, und `0.0` gibt, die eine besondere Behandlung erfordern.
- Wir hatten auch bemerkt, dass jemand kommen könnte und negative Zahlen als Argument an unsere `sqrt`-Funktion übergeben könnte.
- Wir hatten damals nicht die Möglichkeit, gegen so etwas vorzugehen.
- Deshalb haben wir damals einfach `0.0` zurückgeliefert.

```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
```

Beispiel: Quadratwurzel (Original)

- Damals war uns aufgefallen, dass es einige Eingabewerte wie z. B. `inf`, `-inf`, `nan`, und `0.0` gibt, die eine besondere Behandlung erfordern.
- Wir hatten auch bemerkt, dass jemand kommen könnte und negative Zahlen als Argument an unsere `sqrt`-Funktion übergeben könnte.
- Wir hatten damals nicht die Möglichkeit, gegen so etwas vorzugehen.
- Deshalb haben wir damals einfach `0.0` zurückgeliefert.
- Das ist offensichtlich keine gute Idee.

```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
```

Beispiel: Quadratwurzel (Original)

- Wir hatten auch bemerkt, dass jemand kommen könnte und negative Zahlen als Argument an unsere `sqrt`-Funktion übergeben könnte.
- Wir hatten damals nicht die Möglichkeit, gegen so etwas vorzugehen.
- Deshalb haben wir damals einfach `0.0` zurückgeliefert.
- Das ist offensichtlich keine gute Idee.
- Es kann ja nur zwei Gründe geben, warum jemand negative Zahlen in `sqrt` eingibt.

```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
```

Beispiel: Quadratwurzel (Original)

- Wir hatten damals nicht die Möglichkeit, gegen so etwas vorzugehen.
- Deshalb haben wir damals einfach `0.0` zurückgeliefert.
- Das ist offensichtlich keine gute Idee.
- Es kann ja nur zwei Gründe geben, warum jemand negative Zahlen in `sqrt` eingibt.
- Entweder er weiß nicht, was eine Quadratwurzel 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
```

Beispiel: Quadratwurzel (Original)

- Deshalb haben wir damals einfach `0.0` zurückgeliefert.
- Das ist offensichtlich keine gute Idee.
- Es kann ja nur zwei Gründe geben, warum jemand negative Zahlen in `sqrt` eingibt.
- Entweder er weiß nicht, was eine Quadratwurzel ist.
- Oder die negative Zahl ist das Ergebnis einer anderen Berechnung und diese andere Berechnung war irgendwie falsch.

```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
```

Beispiel: Quadratwurzel (Original)

- Das ist offensichtlich keine gute Idee.
- Es kann ja nur zwei Gründe geben, warum jemand negative Zahlen in `sqrt` eingibt.
- Entweder er weiß nicht, was eine Quadratwurzel ist.
- Oder die negative Zahl ist das Ergebnis einer anderen Berechnung und diese andere Berechnung war irgendwie falsch.
- Im ersten Fall sollten wir explizit signalisieren, dass die Quadratwurzel einer negativen Zahl nicht geht (oder zumindest keine reelle Zahl im Sinne von `float` ergibt).

```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
```

Beispiel: Quadratwurzel (Original)

- Entweder er weiß nicht, was eine Quadratwurzel ist.
- Oder die negative Zahl ist das Ergebnis einer anderen Berechnung und diese andere Berechnung war irgendwie falsch.
- Im ersten Fall sollten wir explizit signalisieren, dass die Quadratwurzel einer negativen Zahl nicht geht (oder zumindest keine reelle Zahl im Sinne von `float` ergibt).
- Im zweiten Fall sollten wir lieber hier die Berechnung abbrechen, bevor sich der Schaden weiter ausbreitet.

```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
```

Beispiel: Quadratwurzel (Original)

- Oder die negative Zahl ist das Ergebnis einer anderen Berechnung und diese andere Berechnung war irgendwie falsch.
- Im ersten Fall sollten wir explizit signalisieren, dass die Quadratwurzel einer negativen Zahl nicht geht (oder zumindest keine reelle Zahl im Sinne von `float` ergibt).
- Im zweiten Fall sollten wir lieber hier die Berechnung abbrechen, bevor sich der Schaden weiter ausbreitet.
- Dadurch kann man dann vielleicht den Fehler in der vorangegangenen Berechnung finden.

```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
```

Beispiel: Quadratwurzel (Original)

- Im ersten Fall sollten wir explizit signalisieren, dass die Quadratwurzel einer negativen Zahl nicht geht (oder zumindest keine reelle Zahl im Sinne von `float` ergibt).
- Im zweiten Fall sollten wir lieber hier die Berechnung abbrechen, bevor sich der Schaden weiter ausbreitet.
- Dadurch kann man dann vielleicht den Fehler in der vorangegangenen Berechnung finden.
- In beiden Fällen ist das auslösen einer Ausnahme besser als `0.0` zurückzuliefern.

```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
```

Beispiel: Quadratwurzel (Original)

- Im zweiten Fall sollten wir lieber hier die Berechnung abbrechen, bevor sich der Schaden weiter ausbreitet.
- Dadurch kann man dann vielleicht den Fehler in der vorangegangenen Berechnung finden.
- In beiden Fällen ist das auslösen einer Ausnahme besser als `0.0` zurückzuliefern.
- Genaugenommen sollte unsere Funktion vielleicht *immer* eine Ausnahme auslösen when ihr Ergebnis keine endliche reelle Zahl wäre.

```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
```

Beispiel: Quadratwurzel (Original)

- Dadurch kann man dann vielleicht den Fehler in der vorangegangenen Berechnung finden.
- In beiden Fällen ist das auslösen einer Ausnahme besser als `0.0` zurückzuliefern.
- Genaugenommen sollte unsere Funktion vielleicht *immer* eine Ausnahme auslösen when ihr Ergebnis keine endliche reelle Zahl wäre.
- Dann würden wir sicherstellen, dass unser `sqrt` entweder eine endliche Zahl zurückliefert, die in späteren Berechnungen verwendet werden kann...

```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
```

Beispiel: Quadratwurzel (Original)

- In beiden Fällen ist das auslösen einer Ausnahme besser als `0.0` zurückzuliefern.
- Genaugenommen sollte unsere Funktion vielleicht *immer* eine Ausnahme auslösen when ihr Ergebnis keine endliche reelle Zahl wäre.
- Dann würden wir sicherstellen, dass unser `sqrt` entweder eine endliche Zahl zurückliefert, die in späteren Berechnungen verwendet werden kann...
- ...oder, andernfalls, eine Ausnahme auslöst und fehlschlägt.

```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
```

Beispiel: Quadratwurzel (Original)

- In beiden Fällen ist das auslösen einer Ausnahme besser als `0.0` zurückzuliefern.
- Genaugenommen sollte unsere Funktion vielleicht *immer* eine Ausnahme auslösen when ihr Ergebnis keine endliche reelle Zahl wäre.
- Dann würden wir sicherstellen, dass unser `sqrt` entweder eine endliche Zahl zurückliefert, die in späteren Berechnungen verwendet werden kann...
- ...oder, andernfalls, eine Ausnahme auslöst und fehlschlägt.
- Unsere Funktion würde niemals eine falsche Zahl, `inf`, oder `nan` liefern.

```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
```

Beispiel: Quadratwurzel (neu)

- Wir erstellen also eine neue Implementierung der Funktion `sqrt` in der Datei `sqrt_raise.py`.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir erstellen also eine neue Implementierung der Funktion `sqrt` in der Datei `sqrt_raise.py`.
- Nun müssen wir uns entscheiden, welchen Typ Ausnahme wir verwenden wollen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir erstellen also eine neue Implementierung der Funktion `sqrt` in der Datei `sqrt_raise.py`.
- Nun müssen wir uns entscheiden, welchen Typ Ausnahme wir verwenden wollen.
- `ValueError` wäre OK, denn wir signalisieren einen ungültigen Parameterwert.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir erstellen also eine neue Implementierung der Funktion `sqrt` in der Datei `sqrt_raise.py`.
- Nun müssen wir uns entscheiden, welchen Typ Ausnahme wir verwenden wollen.
- `ValueError` wäre OK, denn wir signalisieren einen ungültigen Parameterwert.
- `ArithmeticError` ist auch OK, denn es schlägt eine Berechnung fehl.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir erstellen also eine neue Implementierung der Funktion `sqrt` in der Datei `sqrt_raise.py`.
- Nun müssen wir uns entscheiden, welchen Typ Ausnahme wir verwenden wollen.
- `ValueError` wäre OK, denn wir signalisieren einen ungültigen Parameterwert.
- `ArithmeticError` ist auch OK, denn es schlägt eine Berechnung fehl.
- Beide Typen wären hier völlig richtig.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Nun müssen wir uns entscheiden, welchen Typ Ausnahme wir verwenden wollen.
- `ValueError` wäre OK, denn wir signalisieren einen ungültigen Parameterwert.
- `ArithmeticError` ist auch OK, denn es schlägt eine Berechnung fehl.
- Beide Typen wären hier völlig richtig.
- Ich wähle `ArithmeticError`, weil ich explizit den mathematischen Hintergrund betonen will.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- `ValueError` wäre OK, denn wir signalisieren einen ungültigen Parameterwert.
- `ArithmeticError` ist auch OK, denn es schlägt eine Berechnung fehl.
- Beide Typen wären hier völlig richtig.
- Ich wähle `ArithmeticError`, weil ich explizit den mathematischen Hintergrund betonen will.
- Wir drücken das neue Verhalten der Funktion im Docstring aus, in dem wir einen entsprechenden `:raises:`-Eintrag anfügen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- `ArithmeticError` ist auch OK, denn es schlägt eine Berechnung fehl.
- Beide Typen wären hier völlig richtig.
- Ich wähle `ArithmeticError`, weil ich explizit den mathematischen Hintergrund betonen will.
- Wir drücken das neue Verhalten der Funktion im Docstring aus, in dem wir einen entsprechenden `:raises:`-Eintrag anfügen.
- In unserem Fall, jede Eingabe x für die $\sqrt{x}$ entweder undefiniert oder nicht endlich wäre, führt zu einem Fehler.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Beide Typen wären hier völlig richtig.
- Ich wähle `ArithmeticError`, weil ich explizit den mathematischen Hintergrund betonen will.
- Wir drücken das neue Verhalten der Funktion im Docstring aus, in dem wir einen entsprechenden `:raises:`-Eintrag anfügen.
- In unserem Fall, jede Eingabe x für die $\sqrt{x}$ entweder undefiniert oder nicht endlich wäre, führt zu einem Fehler.
- Wir schreiben daher `:raises ArithmeticError: if` `` `number` `` `is not finite or less than 0.0`.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir drücken das neue Verhalten der Funktion im Docstring aus, in dem wir einen entsprechenden `:raises:`-Eintrag anfügen.
- In unserem Fall, jede Eingabe x für die $\sqrt{x}$ entweder undefiniert oder nicht endlich wäre, führt zu einem Fehler.
- Wir schreiben daher `:raises ArithmeticError: if` `number` `is not finite or less than 0.0`.
- Jeder andere Programmierer, der unseren Kode verwendet, kann leicht sehen welche `Exceptions` unser Kode auslösen kann.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- In unserem Fall, jede Eingabe x für die $\sqrt{x}$ entweder undefiniert oder nicht endlich wäre, führt zu einem Fehler.
- Wir schreiben daher `:raises ArithmeticError: if `number` is not finite or less than 0.0`.
- Jeder andere Programmierer, der unseren Kode verwendet, kann leicht sehen welche `Exceptions` unser Kode auslösen kann.
- Im eigentlichen Kode prüfen wir nun zuerst ob entweder `not isfinite(number)` oder `number < 0.0` zutrifft.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir schreiben daher `:raises ArithmeticError: if` `number` `is not finite or less than 0.0`.
- Jeder andere Programmierer, der unseren Kode verwendet, kann leicht sehen welche `Exceptions` unser Kode auslösen kann.
- Im eigentlichen Kode prüfen wir nun zuerst ob entweder `not isfinite(number)` oder `number < 0.0` zutrifft.
- Die `isfinite`-Funktion kommt aus dem Modul `math` und liefert `True`, wenn ihr Argument eine finite Zahl ist.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Jeder andere Programmierer, der unseren Kode verwendet, kann leicht sehen welche `Exceptions` unser Kode auslösen kann.
- Im eigentlichen Kode prüfen wir nun zuerst ob entweder `not isfinite(number)` oder `number < 0.0` zutrifft.
- Die `isfinite`-Funktion kommt aus dem Modul `math` und liefert `True`, wenn ihr Argument eine finite Zahl ist.
- Die Funktion liefert `False` für `inf`, `-inf`, und `nan`.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Jeder andere Programmierer, der unseren Kode verwendet, kann leicht sehen welche `Exceptions` unser Kode auslösen kann.
- Im eigentlichen Kode prüfen wir nun zuerst ob entweder `not isfinite(number)` oder `number < 0.0` zutrifft.
- Die `isfinite`-Funktion kommt aus dem Modul `math` und liefert `True`, wenn ihr Argument eine finite Zahl ist.
- Die Funktion liefert `False` für `inf`, `-inf`, und `nan`.
- Der ganze Ausdruck is `True`, wenn das Argument unserer Funktion keine finite Zahl größer/gleich 0 ist.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Im eigentlichen Kode prüfen wir nun zuerst ob entweder `not isfinite(number)` oder `number < 0.0` zutrifft.
- Die `isfinite`-Funktion kommt aus dem Modul `math` und liefert `True`, wenn ihr Argument eine finite Zahl ist.
- Die Funktion liefert `False` für `inf`, `-inf`, und `nan`.
- Der ganze Ausdruck is `True`, wenn das Argument unserer Funktion keine finite Zahl größer/gleich 0 ist.
- In diesem Fall wäre das Ergebnis von `sqrt` keine finite Zahl.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Die `isfinite`-Funktion kommt aus dem Modul `math` und liefert `True`, wenn ihr Argument eine finite Zahl ist.
- Die Funktion liefert `False` für `inf`, `-inf`, und `nan`.
- Der ganze Ausdruck is `True`, wenn das Argument unserer Funktion keine finite Zahl größer/gleich 0 ist.
- In diesem Fall wäre das Ergebnis von `sqrt` keine finite Zahl.
- In dieser Situation machen wir `raise ArithmeticError` mit dem f-String `f"sqrt({number})is not permitted."` als Fehlermeldung.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Die Funktion liefert `False` für `inf`, `-inf`, und `nan`.
- Der ganze Ausdruck is `True`, wenn das Argument unserer Funktion keine finite Zahl größer/gleich 0 ist.
- In diesem Fall wäre das Ergebnis von `sqrt` keine finite Zahl.
- In dieser Situation machen wir `raise ArithmeticError` mit dem f-String `f"sqrt({number})is not permitted."` als Fehlermeldung.
- `raise` ist das Python-Schlüsselwort um Ausnahmen auszulösen, also um einen Fehler zu signalisieren.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Der ganze Ausdruck is `True`, wenn das Argument unserer Funktion keine finite Zahl größer/gleich 0 ist.
- In diesem Fall wäre das Ergebnis von `sqrt` keine finite Zahl.
- In dieser Situation machen wir `raise ArithmeticError` mit dem f-String `f"sqrt({number})is not permitted."` als Fehlermeldung.
- `raise` ist das Python-Schlüsselwort um Ausnahmen auszulösen, also um einen Fehler zu signalisieren.
- `ArithmeticError` erstellt dann ein Objekt mit der Information über diesen Fehler.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- In diesem Fall wäre das Ergebnis von `sqrt` keine finite Zahl.
- In dieser Situation machen wir `raise ArithmeticError` mit dem f-String `f"sqrt({number})is not permitted."` als Fehlermeldung.
- `raise` ist das Python-Schlüsselwort um Ausnahmen auszulösen, also um einen Fehler zu signalisieren.
- `ArithmeticError` erstellt dann ein Objekt mit der Information über diesen Fehler.
- Wir können dieser Funktion einen String als Parameter übergeben.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- In dieser Situation machen wir `raise ArithmeticError` mit dem f-String `f"sqrt({number})is not permitted."` als Fehlermeldung.
- `raise` ist das Python-Schlüsselwort um Ausnahmen auszulösen, also um einen Fehler zu signalisieren.
- `ArithmeticError` erstellt dann ein Objekt mit der Information über diesen Fehler.
- Wir können dieser Funktion einen String als Parameter übergeben.
- Wir benutzen hier einen f-String der den Wert der Zahl mit angibt, mit dem `sqrt` aufgerufen wurde.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- `raise` ist das Python-Schlüsselwort um Ausnahmen auszulösen, also um einen Fehler zu signalisieren.
- `ArithmeticError` erstellt dann ein Objekt mit der Information über diesen Fehler.
- Wir können dieser Funktion einen String als Parameter übergeben.
- Wir benutzen hier einen f-String der den Wert der Zahl mit angibt, mit dem `sqrt` aufgerufen wurde.
- (Man sollte zwar eigentlich keine f-String beim Konstruieren von Ausnahmen direkt verwenden, weil das eine weitere Fehlerquelle ist. . .
  . . . aber ich mache es hier trotzdem.)

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- `ArithmeticError` erstellt dann ein Objekt mit der Information über diesen Fehler.
- Wir können dieser Funktion einen String als Parameter übergeben.
- Wir benutzen hier einen f-String der den Wert der Zahl mit angibt, mit dem `sqrt` aufgerufen wurde.
- (Man sollte zwar eigentlich keine f-String beim Konstruieren von Ausnahmen direkt verwenden, weil das eine weitere Fehlerquelle ist...
  ...aber ich mache es hier trotzdem.)
- Diese Zeile Kode zwingt den Kontrollfluss dazu, unsere Funktion sofort zu verlassen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir können dieser Funktion einen String als Parameter übergeben.
- Wir benutzen hier einen f-String der den Wert der Zahl mit angibt, mit dem `sqrt` aufgerufen wurde.
- (Man sollte zwar eigentlich keine f-String beim Konstruieren von Ausnahmen direkt verwenden, weil das eine weitere Fehlerquelle ist. . . . . . aber ich mache es hier trotzdem.)
- Diese Zeile Kode zwingt den Kontrollfluss dazu, unsere Funktion sofort zu verlassen.
- Das `Exception`-Objekt „steigt auf“ bis es irgendwann „eingefangen“ wird (das lernen wir später).

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (neu)

- Wir benutzen hier einen f-String der den Wert der Zahl mit angibt, mit dem `sqrt` aufgerufen wurde.
- (Man sollte zwar eigentlich keine f-String beim Konstruieren von Ausnahmen direkt verwenden, weil das eine weitere Fehlerquelle ist. . . . . . aber ich mache es hier trotzdem.)
- Diese Zeile Kode zwingt den Kontrollfluss dazu, unsere Funktion sofort zu verlassen.
- Das `Exception`-Objekt „steigt auf“ bis es irgendwann „eingefangen“ wird (das lernen wir später).
- Wird es nicht eingefangen, wird der ganze Prozess abgebrochen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (Aufruf)

- Das Terminieren des Prozesses kann hier im Program `use_sqrt_raise.py` beobachtet werden.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                  ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Das Terminieren des Prozesses kann hier im Program `use_sqrt_raise.py` beobachtet werden.
- In dem Program wenden wir unsere neue `sqrt`-Funktion iterativ auf die Werte in einem Tupel mit Hilfe einer `for`-Schleife an.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Das Terminieren des Prozesses kann hier im Program `use_sqrt_raise.py` beobachtet werden.
- In dem Program wenden wir unsere neue `sqrt`-Funktion iterativ auf die Werte in einem Tupel mit Hilfe einer `for`-Schleife an.
- Wir schreiben die Ergebnisse der Berechnungen auf den standard output stream (`stdout`) mit Hilfe von f-Strings und `print`.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Das Terminieren des Prozesses kann hier im Program `use_sqrt_raise.py` beobachtet werden.
- In dem Program wenden wir unsere neue `sqrt`-Funktion iterativ auf die Werte in einem Tupel mit Hilfe einer `for`-Schleife an.
- Wir schreiben die Ergebnisse der Berechnungen auf den standard output stream (`stdout`) mit Hilfe von f-Strings und `print`.
- Beachten Sie, dass wir diesmal den optionalen Parameter `flush` von `print` verwenden, der sonst den Default Value `False` hat.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Wir schreiben die Ergebnisse der Berechnungen auf den standard output stream (`stdout`) mit Hilfe von f-Strings und `print`.
- Beachten Sie, dass wir diesmal den optionalen Parameter `flush` von `print` verwenden, der sonst den Default Value `False` hat.
- Hier setzen wir ihn auf `flush=True`, was erzwingt dass alle Ausgaben von `print` direkt und sofort auf den `stdout` geschrieben und nicht irgendwo zwischengespeichert wird.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Beachten Sie, dass wir diesmal den optionalen Parameter `flush` von `print` verwenden, der sonst den Default Value `False` hat.
- Hier setzen wir ihn auf `flush=True`, was erzwingt dass alle Ausgaben von `print` direkt und sofort auf den `stdout` geschrieben und nicht irgendwo zwischengespeichert wird.
- Das ist nützlich in diesem speziellen Beispiel um Vermischungen vom `stdout`-Stream, der die normale Ausgabe des Programms erhält, und dem standard error stream (`stderr`), auf dem Fehlermeldungen auftauchen, zu verhindern.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Hier setzen wir ihn auf `flush=True`, was erzwingt dass alle Ausgaben von `print` direkt und sofort auf den `stdout` geschrieben und nicht irgendwo zwischengespeichert wird.
- Das ist nützlich in diesem speziellen Beispiel um Vermischungen vom `stdout`-Stream, der die normale Ausgabe des Programms erhält, und dem standard error stream (`stderr`), auf dem Fehlermeldungen auftauchen, zu verhindern.
- Beide Streams sind nämlich zusammen in der Ausgabe rechts dargestellt.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Das ist nützlich in diesem speziellen Beispiel um Vermischungen vom `stdout`-Stream, der die normale Ausgabe des Programms erhält, und dem standard error stream (`stderr`), auf dem Fehlermeldungen auftauchen, zu verhindern.
- Beide Streams sind nämlich zusammen in der Ausgabe rechts dargestellt.
- Egal. Die ersten fünf Zahlen sind OK und `sqrt` liefert vernünftige Ergebnisse.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Beide Streams sind nämlich zusammen in der Ausgabe rechts dargestellt.
- Egal. Die ersten fünf Zahlen sind OK und `sqrt` liefert vernünftige Ergebnisse.
- Die drei Zahlen danach, `inf`, `nan`, und `-1.0`, würden aber alle einen Fehler auslösen.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Beide Streams sind nämlich zusammen in der Ausgabe rechts dargestellt.
- Egal. Die ersten fünf Zahlen sind OK und `sqrt` liefert vernünftige Ergebnisse.
- Die drei Zahlen danach, `inf`, `nan`, und `-1.0`, würden aber alle einen Fehler auslösen.
- Für `0.0`, `1.0`, `2.0`, `4.0`, und `10.0` werden die erwarteten Ergebnisse ausgegeben.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Beide Streams sind nämlich zusammen in der Ausgabe rechts dargestellt.
- Egal. Die ersten fünf Zahlen sind OK und `sqrt` liefert vernünftige Ergebnisse.
- Die drei Zahlen danach, `inf`, `nan`, und `-1.0`, würden aber alle einen Fehler auslösen.
- Für `0.0`, `1.0`, `2.0`, `4.0`, und `10.0` werden die erwarteten Ergebnisse ausgegeben.
- Wenn die `for`-Schleife aber `inf` erreicht, dann bricht das Programm ab und der so genannte Stack-Trace wird ausgegeben.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Die drei Zahlen danach, `inf`, `nan`, und `-1.0`, würden aber alle einen Fehler auslösen.
- Für `0.0`, `1.0`, `2.0`, `4.0`, und `10.0` werden die erwarteten Ergebnisse ausgegeben.
- Wenn die `for`-Schleife aber `inf` erreicht, dann bricht das Programm ab und der so genannte Stack-Trace wird ausgegeben.
- Wir hatten diese Informationen schonmal in Einheit 14 erwähnt, als es darum ging, Fehler mit Hilfe der IDE zu finden.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Für `0.0`, `1.0`, `2.0`, `4.0`, und `10.0` werden die erwarteten Ergebnisse ausgegeben.
- Wenn die `for`-Schleife aber `inf` erreicht, dann bricht das Programm ab und der so genannte Stack-Trace wird ausgegeben.
- Wir hatten diese Informationen schonmal in Einheit 14 erwähnt, als es darum ging, Fehler mit Hilfe der IDE zu finden.
- Der Stack-Trace beginnt mit der Zeile *„Traceback (most recent call last):“*.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Wenn die `for`-Schleife aber `inf` erreicht, dann bricht das Programm ab und der so genannte Stack-Trace wird ausgegeben.
- Wir hatten diese Informationen schonmal in Einheit 14 erwähnt, als es darum ging, Fehler mit Hilfe der IDE zu finden.
- Der Stack-Trace beginnt mit der Zeile *„Traceback (most recent call last):“*.
- Dann wird der Pfad der Programmdatei `use_sqrt_raise.py`, wo der Fehler aufgetreten ist, sowie die entsprechende Zeile ausgegeben.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Wir hatten diese Informationen schonmal in Einheit 14 erwähnt, als es darum ging, Fehler mit Hilfe der IDE zu finden.
- Der Stack-Trace beginnt mit der Zeile *„Traceback (most recent call last):“*.
- Dann wird der Pfad der Programmdatei `use_sqrt_raise.py`, wo der Fehler aufgetreten ist, sowie die entsprechende Zeile ausgegeben.
- (Die Pfade sehen bei Ihnen anders aus.)

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Der Stack-Trace beginnt mit der Zeile *„Traceback (most recent call last):“*.
- Dann wird der Pfad der Programmdatei `use_sqrt_raise.py`, wo der Fehler aufgetreten ist, sowie die entsprechende Zeile ausgegeben.
- (Die Pfade sehen bei Ihnen anders aus.)
- The last part of the path, `exceptions/use_sqrt_raise.py`, stellt unser aufrufendes Programm klar als Fehlerursache dar.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Der Stack-Trace beginnt mit der Zeile *„Traceback (most recent call last):“*.
- Dann wird der Pfad der Programmdatei `use_sqrt_raise.py`, wo der Fehler aufgetreten ist, sowie die entsprechende Zeile ausgegeben.
- (Die Pfade sehen bei Ihnen anders aus.)
- The last part of the path, `exceptions/use_sqrt_raise.py`, stellt unser aufrufendes Programm klar als Fehlerursache dar.
- Die folgende Zeile Text identifiziert die fehlerhafte Instruktion und „unterstreicht“ diese sogar.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Dann wird der Pfad der Programmdatei `use_sqrt_raise.py`, wo der Fehler aufgetreten ist, sowie die entsprechende Zeile ausgegeben.
- (Die Pfade sehen bei Ihnen anders aus.)
- The last part of the path, `exceptions/use_sqrt_raise.py`, stellt unser aufrufendes Programm klar als Fehlerursache dar.
- Die folgende Zeile Text identifiziert die fehlerhafte Instruktion und „unterstreicht“ diese sogar.
- Dann sehen wir den Kontext unserer `sqrt`-Funktion.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- (Die Pfade sehen bei Ihnen anders aus.)
- The last part of the path, `exceptions/use_sqrt_raise.py`, stellt unser aufrufendes Programm klar als Fehlerursache dar.
- Die folgende Zeile Text identifiziert die fehlerhafte Instruktion und „unterstreicht“ diese sogar.
- Dann sehen wir den Kontext unserer `sqrt`-Funktion.
- Der Pfad zu ihrem Modul wird angegeben (er endet mit `exceptions/sqrt_raise.py`) und es wird angezeigt, dass der Fehler in Zeile 16 auftrat.

```
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- The last part of the path, `exceptions/use_sqrt_raise.py`, stellt unser aufrufendes Programm klar als Fehlerursache dar.
- Die folgende Zeile Text identifiziert die fehlerhafte Instruktion und „unterstreicht“ diese sogar.
- Dann sehen wir den Kontext unserer `sqrt`-Funktion.
- Der Pfad zu ihrem Modul wird angegeben (er endet mit `exceptions/sqrt_raise.py`) und es wird angezeigt, dass der Fehler in Zeile 16 auftrat.
- Diese Zeile Kode wird auch angezeigt.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Die folgende Zeile Text identifiziert die fehlerhafte Instruktion und „unterstreicht“ diese sogar.
- Dann sehen wir den Kontext unserer `sqrt`-Funktion.
- Der Pfad zu ihrem Modul wird angegeben (er endet mit `exceptions/sqrt_raise.py`) und es wird angezeigt, dass der Fehler in Zeile 16 auftrat.
- Diese Zeile Kode wird auch angezeigt.
- Es ist genau die Zeile, wo wir `raise ArithmeticError` machen.

```
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Dann sehen wir den Kontext unserer `sqrt`-Funktion.
- Der Pfad zu ihrem Modul wird angegeben (er endet mit `exceptions/sqrt_raise.py`) und es wird angezeigt, dass der Fehler in Zeile 16 auftrat.
- Diese Zeile Kode wird auch angezeigt.
- Es ist genau die Zeile, wo wir `raise ArithmeticError` machen.
- Der Stack-Trace hat uns also genau gezeigt, wo der Fehler auftrat.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Der Pfad zu ihrem Modul wird angegeben (er endet mit `exceptions/sqrt_raise.py`) und es wird angezeigt, dass der Fehler in Zeile 16 auftrat.
- Diese Zeile Kode wird auch angezeigt.
- Es ist genau die Zeile, wo wir `raise ArithmeticError` machen.
- Der Stack-Trace hat uns also genau gezeigt, wo der Fehler auftrat.
- Danach sehen wir die weiteren Informationen des Fehlers, es wir nämlich ausgedruckt dass *„ArithmeticError: sqrt(inf) is not permitted.“*

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Diese Zeile Kode wird auch angezeigt.
- Es ist genau die Zeile, wo wir `raise ArithmeticError` machen.
- Der Stack-Trace hat uns also genau gezeigt, wo der Fehler auftrat.
- Danach sehen wir die weiteren Informationen des Fehlers, es wir nämlich ausgedruckt dass *„ArithmeticError: sqrt(inf) is not permitted.“*
- Wir hatten diese Nachricht selber zusammengebaut, als wir die Ausnahme ausgelöst haben.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Es ist genau die Zeile, wo wir `raise ArithmeticError` machen.
- Der Stack-Trace hat uns also genau gezeigt, wo der Fehler auftrat.
- Danach sehen wir die weiteren Informationen des Fehlers, es wir nämlich ausgedruckt dass *„ArithmeticError: sqrt(inf) is not permitted.“*
- Wir hatten diese Nachricht selber zusammengebaut, als wir die Ausnahme ausgelöst haben.
- Wir machten dass, damit der Benutzer versteht, dass `sqrt` mit dem Argument `inf` aufgerufen wurde, was wir nicht erlauben.

```
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Danach sehen wir die weiteren Informationen des Fehlers, es wir nämlich ausgedruckt dass *„ArithmeticError: sqrt(inf) is not permitted.“*
- Wir hatten diese Nachricht selber zusammengebaut, als wir die Ausnahme ausgelöst haben.
- Wir machten dass, damit der Benutzer versteht, dass `sqrt` mit dem Argument `inf` aufgerufen wurde, was wir nicht erlauben.
- Mit dieser Information können wir ziemlich genau die Quelle des Problems finden.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                  ~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Wir hatten diese Nachricht selber zusammengebaut, als wir die Ausnahme ausgelöst haben.
- Wir machten dass, damit der Benutzer versteht, dass `sqrt` mit dem Argument `inf` aufgerufen wurde, was wir nicht erlauben.
- Mit dieser Information können wir ziemlich genau die Quelle des Problems finden.
- Programmierer ignorieren diese Meldungen leider oft.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Wir hatten diese Nachricht selber zusammengebaut, als wir die Ausnahme ausgelöst haben.
- Wir machten dass, damit der Benutzer versteht, dass `sqrt` mit dem Argument `inf` aufgerufen wurde, was wir nicht erlauben.
- Mit dieser Information können wir ziemlich genau die Quelle des Problems finden.
- Programmierer ignorieren diese Meldungen leider oft.
- Oft sehen sie, dass das Programm einen Fehler ausgelöst hat und suchen dann im Kode nach Fehlern.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Wir machten dass, damit der Benutzer versteht, dass `sqrt` mit dem Argument `inf` aufgerufen wurde, was wir nicht erlauben.
- Mit dieser Information können wir ziemlich genau die Quelle des Problems finden.
- Programmierer ignorieren diese Meldungen leider oft.
- Oft sehen sie, dass das Programm einen Fehler ausgelöst hat und suchen dann im Kode nach Fehlern.
- Dabei lesen sie den Stack-Trace oftmals nicht.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Mit dieser Information können wir ziemlich genau die Quelle des Problems finden.
- Programmierer ignorieren diese Meldungen leider oft.
- Oft sehen sie, dass das Programm einen Fehler ausgelöst hat und suchen dann im Kode nach Fehlern.
- Dabei lesen sie den Stack-Trace oftmals nicht.

Gute Praxis

Der Stack-Trace und die Fehlerinformation, welche in der Python-Konsole ausgegeben werden, wenn eine Ausnahme nicht verarbeitet wurde, sind *essenziell* um das Problem zu finden. Sie sollten *immer* gelesen und verstanden werden, bevor am Kode weitergearbeitet wird.

Beispiel: Quadratwurzel (Aufruf)

- Programmierer ignorieren diese Meldungen leider oft.
- Oft sehen sie, dass das Programm einen Fehler ausgelöst hat und suchen dann im Kode nach Fehlern.
- Dabei lesen sie den Stack-Trace oftmals nicht.
- In dem Tupel mit Eingabedaten für `sqrt` würden die letzten drei Zahlen `inf`, `nan`, und `-1.0` zu Fehlern führen.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Oft sehen sie, dass das Programm einen Fehler ausgelöst hat und suchen dann im Kode nach Fehlern.
- Dabei lesen sie den Stack-Trace oftmals nicht.
- In dem Tupel mit Eingabedaten für `sqrt` würden die letzten drei Zahlen `inf`, `nan`, und `-1.0` zu Fehlern führen.
- Der Aufruf von `sqrt` mit `inf` als Argument wurde durchgeführt – und ist fehlgeschlagen.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Dabei lesen sie den Stack-Trace oftmals nicht.
- In dem Tupel mit Eingabedaten für `sqrt` würden die letzten drei Zahlen `inf`, `nan`, und `-1.0` zu Fehlern führen.
- Der Aufruf von `sqrt` mit `inf` als Argument wurde durchgeführt – und ist fehlgeschlagen.
- Danach gibt es keine weitere Ausgabe.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- In dem Tupel mit Eingabedaten für `sqrt` würden die letzten drei Zahlen `inf`, `nan`, und `-1.0` zu Fehlern führen.
- Der Aufruf von `sqrt` mit `inf` als Argument wurde durchgeführt – und ist fehlgeschlagen.
- Danach gibt es keine weitere Ausgabe.
- Der Kontrollfluss hat den Kode verlassen, die `for`-Schleife wurde abgebrochen, und das Programm mit exit code 1 beendet.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- In dem Tupel mit Eingabedaten für `sqrt` würden die letzten drei Zahlen `inf`, `nan`, und `-1.0` zu Fehlern führen.
- Der Aufruf von `sqrt` mit `inf` als Argument wurde durchgeführt – und ist fehlgeschlagen.
- Danach gibt es keine weitere Ausgabe.
- Der Kontrollfluss hat den Kode verlassen, die `for`-Schleife wurde abgebrochen, und das Programm mit exit code 1 beendet.
- Ein ganzes Program wegen diesem einem Fehler abzubrechen sieht ziemlich harsch aus.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Der Aufruf von `sqrt` mit `inf` als Argument wurde durchgeführt – und ist fehlgeschlagen.
- Danach gibt es keine weitere Ausgabe.
- Der Kontrollfluss hat den Kode verlassen, die `for`-Schleife wurde abgebrochen, und das Programm mit exit code 1 beendet.
- Ein ganzes Program wegen diesem einem Fehler abzubrechen sieht ziemlich harsch aus.
- Aber es ist gut.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Danach gibt es keine weitere Ausgabe.
- Der Kontrollfluss hat den Kode verlassen, die `for`-Schleife wurde abgebrochen, und das Programm mit exit code 1 beendet.
- Ein ganzes Program wegen diesem einem Fehler abzubrechen sieht ziemlich harsch aus.
- Aber es ist gut.
- Wenn ein Programmierer unsere `sqrt`-Funktion falsch verwendet hat, dann wird er gezwungen, seinen Fehler zu korrigieren.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Der Kontrollfluss hat den Kode verlassen, die `for`-Schleife wurde abgebrochen, und das Programm mit exit code 1 beendet.
- Ein ganzes Program wegen diesem einem Fehler abzubrechen sieht ziemlich harsch aus.
- Aber es ist gut.
- Wenn ein Programmierer unsere `sqrt`-Funktion falsch verwendet hat, dann wird er gezwungen, seinen Fehler zu korrigieren.
- Wenn die Eingabedaten fehlerhaft waren, dann verhindert das Terminieren des Programms, das der Fehler sich fortpflanzt.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (Aufruf)

- Ein ganzes Program wegen diesem einem Fehler abzubrechen sieht ziemlich harsch aus.
- Aber es ist gut.
- Wenn ein Programmierer unsere `sqrt`-Funktion falsch verwendet hat, dann wird er gezwungen, seinen Fehler zu korrigieren.
- Wenn die Eingabedaten fehlerhaft waren, dann verhindert das Terminieren des Programms, das der Fehler sich fortpflanzt.
- In beiden Fällen hat uns der Stack-Trace klar gezeigt, wo der Fehler liegt.

```python
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ python3 use_sqrt_raise.py ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (mit `TypeError`)

- Früher in dieser Einheit haben wir gesagt, dass es eine schlechte Idee ist, zu lässig mit den Datentypen von Parametern umzugehen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Früher in dieser Einheit haben wir gesagt, dass es eine schlechte Idee ist, zu lässig mit den Datentypen von Parametern umzugehen.
- Wenn wir die Eingabedaten einfach gemütlich in die Datentypen umwandeln, die wir brauchen, dann verlassen sich Benutzer auf dieses Verhalten und wir müssen das in allen zukünftigen Versionen unseres Kodes weiter unterstützen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Früher in dieser Einheit haben wir gesagt, dass es eine schlechte Idee ist, zu lässig mit den Datentypen von Parametern umzugehen.
- Wenn wir die Eingabedaten einfach gemütlich in die Datentypen umwandeln, die wir brauchen, dann verlassen sich Benutzer auf dieses Verhalten und wir müssen das in allen zukünftigen Versionen unseres Kodes weiter unterstützen.
- Wir sollten also niemals zulassen, dass unsere Funktionen mit Datentypen aufgerufen werden, für die sie nicht gedacht sind.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Wenn wir die Eingabedaten einfach gemütlich in die Datentypen umwandeln, die wir brauchen, dann verlassen sich Benutzer auf dieses Verhalten und wir müssen das in allen zukünftigen Versionen unseres Kodes weiter unterstützen.
- Wir sollten also niemals zulassen, dass unsere Funktionen mit Datentypen aufgerufen werden, für die sie nicht gedacht sind.
- Natürlich, wenn jetzt jemand daher käme und unsere `sqrt`-Funktion mit einem String als Argument aufruft, dann wird das mit einem `TypeError` fehlschlagen.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Wir sollten also niemals zulassen, dass unsere Funktionen mit Datentypen aufgerufen werden, für die sie nicht gedacht sind.
- Natürlich, wenn jetzt jemand daher käme und unsere `sqrt`-Funktion mit einem String als Argument aufruft, dann wird das mit einem `TypeError` fehlschlagen.
- Dieser Fehler würde irgendwo im Kode auftauchen, weil Fließkommaarithmetik nunmal nicht mit Strings funktioniert.

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)

- Natürlich, wenn jetzt jemand daher käme und unsere `sqrt`-Funktion mit einem String als Argument aufruft, dann wird das mit einem `TypeError` fehlschlagen.
- Dieser Fehler würde irgendwo im Kode auftauchen, weil Fließkommaarithmetik nunmal nicht mit Strings funktioniert.
- Wir fragen uns: Können wir selber auch gezielt einen `TypeError` auslösen?

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Dieser Fehler würde irgendwo im Kode auftauchen, weil Fließkommaarithmetik nunmal nicht mit Strings funktioniert.
- Wir fragen uns: Können wir selber auch gezielt einen `TypeError` auslösen?
- Und wie können wir überhaupt wissen, ob eine Variable einen bestimmten Typ hat?

```python
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Wir fragen uns: Können wir selber auch gezielt einen `TypeError` auslösen?
- Und wie können wir überhaupt wissen, ob eine Variable einen bestimmten Typ hat?
- Die Antwort zu beiden Fragen findet sich in unserer neuen Variante der `sqrt`-Funktion in Datei `sqrt_raise_2.py`.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Wir fragen uns: Können wir selber auch gezielt einen `TypeError` auslösen?
- Und wie können wir überhaupt wissen, ob eine Variable einen bestimmten Typ hat?
- Die Antwort zu beiden Fragen findet sich in unserer neuen Variante der `sqrt`-Funktion in Datei `sqrt_raise_2.py`.
- Hier prüfen wir anfänglich ob `isinstance(number, float)`.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Wir fragen uns: Können wir selber auch gezielt einen `TypeError` auslösen?
- Und wie können wir überhaupt wissen, ob eine Variable einen bestimmten Typ hat?
- Die Antwort zu beiden Fragen findet sich in unserer neuen Variante der `sqrt`-Funktion in Datei `sqrt_raise_2.py`.
- Hier prüfen wir anfänglich ob `isinstance(number, float)`.
- Die Funktion `isinstance` verlangt ein Objekt als ersten Parameter und einen Typ als zweiten Parameter.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Und wie können wir überhaupt wissen, ob eine Variable einen bestimmten Typ hat?
- Die Antwort zu beiden Fragen findet sich in unserer neuen Variante der `sqrt`-Funktion in Datei `sqrt_raise_2.py`.
- Hier prüfen wir anfänglich ob `isinstance(number, float)`.
- Die Funktion `isinstance` verlangt ein Objekt als ersten Parameter und einen Typ als zweiten Parameter.
- Wir wollen das Objekt `number` überprüfen.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Die Antwort zu beiden Fragen findet sich in unserer neuen Variante der `sqrt`-Funktion in Datei `sqrt_raise_2.py`.
- Hier prüfen wir anfänglich ob `isinstance(number, float)`.
- Die Funktion `isinstance` verlangt ein Objekt als ersten Parameter und einen Typ als zweiten Parameter.
- Wir wollen das Objekt `number` überprüfen.
- Der Typ, auf den wir prüfen wollen, ist `float`.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)

- Hier prüfen wir anfänglich ob `isinstance(number, float)`.
- Die Funktion `isinstance` verlangt ein Objekt als ersten Parameter und einen Typ als zweiten Parameter.
- Wir wollen das Objekt `number` überprüfen.
- Der Typ, auf den wir prüfen wollen, ist `float`.
- `isinstance(number, float)` liefert `True` wenn `number` zum Typ `float` gehört.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)

- Die Funktion `isinstance` verlangt ein Objekt als ersten Parameter und einen Typ als zweiten Parameter.
- Wir wollen das Objekt `number` überprüfen.
- Der Typ, auf den wir prüfen wollen, ist `float`.
- `isinstance(number, float)` liefert `True` wenn `number` zum Typ `float` gehört.
- Andernfalls liefert es `False`.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Wir wollen das Objekt `number` überprüfen.
- Der Typ, auf den wir prüfen wollen, ist `float`.
- `isinstance(number, float)` liefert `True` wenn `number` zum Typ `float` gehört.
- Andernfalls liefert es `False`.
- Wir lösen also einen `TypeError` aus, wenn `isinstance(number, float)` *nicht* gilt.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- Der Typ, auf den wir prüfen wollen, ist `float`.
- `isinstance(number, float)` liefert `True` wenn `number` zum Typ `float` gehört.
- Andernfalls liefert es `False`.
- Wir lösen also einen `TypeError` aus, wenn `isinstance(number, float)` *nicht* gilt.
- Dann übergeben wir auch eine vernünftige Fehlermeldung.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError`)



- `isinstance(number, float)` liefert `True` wenn `number` zum Typ `float` gehört.
- Andernfalls liefert es `False`.
- Wir lösen also einen `TypeError` aus, wenn `isinstance(number, float)` *nicht* gilt.
- Dann übergeben wir auch eine vernünftige Fehlermeldung.
- Natürlich schreiben wir das auch wieder in den Docstring.

```python
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


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 == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    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
```

Beispiel: Quadratwurzel (mit `TypeError` aufrufen)

- Jetzt schreiben wir ein Programm `use_sqrt_raise_2.py`, dass unsere neue Funktion `sqrt` verwendet.

```python
"""Using the sqrt function which raises errors also for wrong types."""

from sqrt_raise_2 import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in [0.0, 1.0, 2.0, 4.0, "0.3"]:
    # We get an error when reaching `"0.3"`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise_2.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise_2.py", line 8, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~^^^^^^^^^
  File "{...}/exceptions/sqrt_raise_2.py", line 17, in sqrt
    raise TypeError("number must be float!")
TypeError: number must be float!
# 'python3 use_sqrt_raise_2.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (mit `TypeError` aufrufen)

- Jetzt schreiben wir ein Programm `use_sqrt_raise_2.py`, dass unsere neue Funktion `sqrt` verwendet.
- Anders als beim letzten Mal übergeben wir diesmal einen String `"0.3"` als „falschen“ Parameter.

```python
"""Using the sqrt function which raises errors also for wrong types."""

from sqrt_raise_2 import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in [0.0, 1.0, 2.0, 4.0, "0.3"]:
    # We get an error when reaching `"0.3"`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise_2.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise_2.py", line 8, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~^^^^^^^^
  File "{...}/exceptions/sqrt_raise_2.py", line 17, in sqrt
    raise TypeError("number must be float!")
TypeError: number must be float!
# 'python3 use_sqrt_raise_2.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (mit `TypeError` aufrufen)

- Jetzt schreiben wir ein Programm `use_sqrt_raise_2.py`, dass unsere neue Funktion `sqrt` verwendet.
- Anders als beim letzten Mal übergeben wir diesmal einen String `"0.3"` als „falschen“ Parameter.
- Das führt dann dazu, dass ein `TypeError` ausgelöst wird.

```python
"""Using the sqrt function which raises errors also for wrong types."""

from sqrt_raise_2 import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in [0.0, 1.0, 2.0, 4.0, "0.3"]:
    # We get an error when reaching `"0.3"`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise_2.py` ↓

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise_2.py", line 8, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                 ~~~~^^^^^^^^
  File "{...}/exceptions/sqrt_raise_2.py", line 17, in sqrt
    raise TypeError("number must be float!")
TypeError: number must be float!
# 'python3 use_sqrt_raise_2.py' failed with exit code 1.
```

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Wie ist es überhaupt möglich ist, dass wir `sqrt` mit einem String als Argument aufrufen, obwohl wir doch seinen Parameter explizit mit dem Type-Hint `float` versehen haben?

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Wie ist es überhaupt möglich ist, dass wir `sqrt` mit einem String als Argument aufrufen, obwohl wir doch seinen Parameter explizit mit dem Type-Hint `float` versehen haben?
- Das geht, weil Python keine Typ-Restriktionen zur Laufzeit erzwingt.

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Wie ist es überhaupt möglich ist, dass wir `sqrt` mit einem String als Argument aufrufen, obwohl wir doch seinen Parameter explizit mit dem Type-Hint `float` versehen haben?
- Das geht, weil Python keine Typ-Restriktionen zur Laufzeit erzwingt.
- Wir können zwar *schreiben*, dass unsere Funktion nur `floats` als Eingabedaten bekommen soll.

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Wie ist es überhaupt möglich ist, dass wir `sqrt` mit einem String als Argument aufrufen, obwohl wir doch seinen Parameter explizit mit dem Type-Hint `float` versehen haben?
- Das geht, weil Python keine Typ-Restriktionen zur Laufzeit erzwingt.
- Wir können zwar *schreiben*, dass unsere Funktion nur `floats` als Eingabedaten bekommen soll.
- Der Benutzer kann das aber einfach ignorieren.

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Wie ist es überhaupt möglich ist, dass wir `sqrt` mit einem String als Argument aufrufen, obwohl wir doch seinen Parameter explizit mit dem Type-Hint `float` versehen haben?
- Das geht, weil Python keine Typ-Restriktionen zur Laufzeit erzwingt.
- Wir können zwar *schreiben*, dass unsere Funktion nur `floats` als Eingabedaten bekommen soll.
- Der Benutzer kann das aber einfach ignorieren.
- Wir lassen ihn nur nicht damit davonkommen...

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Wie ist es überhaupt möglich ist, dass wir `sqrt` mit einem String als Argument aufrufen, obwohl wir doch seinen Parameter explizit mit dem Type-Hint `float` versehen haben?
- Das geht, weil Python keine Typ-Restriktionen zur Laufzeit erzwingt.
- Wir können zwar *schreiben*, dass unsere Funktion nur `floats` als Eingabedaten bekommen soll.
- Der Benutzer kann das aber einfach ignorieren.
- Wir lassen ihn nur nicht damit davonkommen...
- Mypy hätte übrigens gesehen, dass `use_sqrt_raise_2.py` falsch ist.

```
1 $ mypy use_sqrt_raise_2.py --no-strict-optional --check-untyped-defs
2 use_sqrt_raise_2.py:8: error: Argument 1 to "sqrt" has incompatible type
    ↪ "object"; expected "float"  [arg-type]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 2.4.1 failed with exit code 1.
```

Beispiel: Quadratwurzel (mit `TypeError` und Mypy)



- Das geht, weil Python keine Typ-Restriktionen zur Laufzeit erzwingt.
- Wir können zwar *schreiben*, dass unsere Funktion nur `floats` als Eingabedaten bekommen soll.
- Der Benutzer kann das aber einfach ignorieren.
- Wir lassen ihn nur nicht damit davonkommen...
- Mypy hätte übrigens gesehen, dass `use_sqrt_raise_2.py` falsch ist.
- Hätte der Programmierer das Programm damit überprüft, hätte er den Fehler beheben können.

```
1 $ mypy use_sqrt_raise_2.py --no-strict-optional --check-untyped-defs
2 use_sqrt_raise_2.py:8: error: Argument 1 to "sqrt" has incompatible type
    ↪ "object"; expected "float"  [arg-type]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 2.4.1 failed with exit code 1.
```

Built-In Exceptions



Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?
- Welche Typen von `Exceptions` sind „built-in“?

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?
- Welche Typen von `Exceptions` sind „built-in“?
- Python bietet eine *Hierarchie* von Ausnahmen.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?
- Welche Typen von `Exceptions` sind „built-in“?
- Python bietet eine *Hierarchie* von Ausnahmen.
- Manche Ausnahmeklassen sind Sonderfälle anderer Ausnahmen.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?
- Welche Typen von `Exceptions` sind „built-in“?
- Python bietet eine *Hierarchie* von Ausnahmen.
- Manche Ausnahmeklassen sind Sonderfälle anderer Ausnahmen.
- Ein `ArithmeticError` markiert einen Rechenfehler.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?
- Welche Typen von `Exceptions` sind „built-in“?
- Python bietet eine *Hierarchie* von Ausnahmen.
- Manche Ausnahmeklassen sind Sonderfälle anderer Ausnahmen.
- Ein `ArithmeticError` markiert einen Rechenfehler.
- Ein `OverflowError` ist ein besonderer Rechenfehler, bei dem das Ergebnis zu große für den Datentyp wird.

Built-In Exceptions



- Eine große Vielzahl von Dingen kann bei der Ausführung eines Programm fehlschlagen.
- Wir haben schon ein paar mögliche Fehler genannt.
- Treten Fehler auf, dann werden Ausnahmen (EN: *Exceptions*) ausgelöst.
- Diese brechen den aktuellen Kontrollfluss ab und werden immer weiter nach oben propagiert, bis sie entweder behandelt werden (lernen wir in der nächsten Einheit) oder der Prozess abbricht.
- Was für Typen von Ausnahmen stellt uns Python eigentlich zur Verfügung?
- Welche Typen von `Exceptions` sind „built-in“?
- Python bietet eine *Hierarchie* von Ausnahmen.
- Manche Ausnahmeklassen sind Sonderfälle anderer Ausnahmen.
- Ein `ArithmeticError` markiert einen Rechenfehler.
- Ein `OverflowError` ist ein besonderer Rechenfehler, bei dem das Ergebnis zu große für den Datentyp wird.
- Schauen wir uns die Typ-Hierarchy von Fehlern mal an[6].

Built-In Exceptions (1)

- `BaseException` — die Basisklasse aller Ausnahmen
  - `Exception` — Situationen, wo vernüftige Fehlerbehandlung möglich seien sollte
    - `ArithmeticError` — eine Rechenoperation ist fehlgeschlagen
      - `FloatingPointError` — nicht Python benutzt, aber z. B. für NumPy bei ungültigen Fließkommaoperationen
      - `OverflowError` — das Ergebnis einer Rechenoperation ist zu groß
      - `ZeroDivisionError` — eine Division durch 0
    - `AssertionError` — ein `assert`-Statement schlug fehl
    - `BufferError` — eine Puffer-bezogene Operation ist fehlgeschlagen
    - `EOFError` — das Ende von standard input stream (`stdin`) wurde von `input` erreicht, ohne Daten zu lesen
    - `AttributeError` — wenn eine Attribut-Referenz oder Zuweisung fehlschlägt
    - `ImportError` — wenn `import` fehlschlägt
      - `ModuleNotFoundError` — ein Modul konnte nicht geladen werden
    - `LookupError` — ein Schlüssel oder Index in eine Dictionary oder einer Sequenz war ungültig
      - `IndexError` — ein Sequenz-Index war außerhalb der Reichweite
      - `KeyError` — ein Schlüssel für ein Dictionary war falsch
    - `MemoryError` — nicht mehr genug freier Speicher
    - `NameError` — z. B. Zugriff auf eine nicht-existierende Variable
      - `UnboundLocalError` — Referenz auf eine Methode oder Funktion, die nicht gebunden ist
    - `OSError` — ein Betriebssystem-Funktionsaufruf schlug fehl

Built-In Exceptions (2)

- `BaseException` — die Basisklasse aller Ausnahmen
  - `Exception` — Situationen, wo vernüftige Fehlerbehandlung möglich seien sollte
    - `OSError` — ein Betriebssystem-Funktionsaufruf schlug fehl
      - `BlockingIOError` — eine blockierende Operation wurde auf ein Objekt im nicht-blockierenden Modus angewandt
      - `ChildProcessError` — eine Operation in einem Subprozess schlug fehl
      - `ConnectionError` — ein Verbindungs- oder Pipe-bezogener Fehler
        - `BrokenPipeError` — Schreiben in eine Pipe deren Ende geschlossen it
        - `ConnectionAbortedError` — Verbindungsversuch von anderer Seite abgebrochen
        - `ConnectionRefusedError` — andere Seite verweigert Verbindung
        - `ConnectionResetError` — Verbindung von anderer Seite abgebrochen
      - `FileExistsError` — Versuch, eine Datei zu erstellen, die schon existiert
      - `FileNotFoundError` — Datei nicht gefunden
      - `IsADirectoryError` — Versuch einer Dateioperation auf ein Verzeichnis
      - `NotADirectoryError` — Versuch einer Verzeichnisoperation auf eine Datei
      - `PermissionError` — unzureichende Zugriffsrechte
      - `ProcessLookupError` — Prozess existiert nicht
      - `TimeoutError` — eine Operation wurde wegen Time-out abgebrochen
    - `ReferenceError` — auf ein schwach-referenziertes Objekt wurde nach seiner Freigabe zugegriffen

Built-In Exceptions (3)

- `BaseException` — die Basisklasse aller Ausnahmen
  - `Exception` — Situationen, wo vernüftige Fehlerbehandlung möglich seien sollte
    - `RuntimeError` — ein Fehler der nicht in die anderen Kategorien passen
      - `NotImplementedError` — eine Methode wurde noch nicht implementiert, aber vielleicht später
      - `PythonFinalizationError` — einer Operation wurde beim Herunterfahren des Interpreters blockiert
      - `RecursionError` — die maximale Rekursionstiefe wurde überschritten
    - `StopAsyncIteration` — kein Fehler, sondern das Ende einer asynchronen Iteration
    - `StopIteration` — kein Fehler, sondern das Ende einer Iteration
    - `SyntaxError` — fehlerhafte Python-Datei
      - `IndentationError` — falsch eingrückter Kode
        - `TabError` — inkonsistente Benutzung von Tabs und Leerzeichen
    - `SystemError` — ein interner Fehler des Interpreter
    - `TypeError` — in Parameter hatte einen falschen Typ oder war `None`
    - `ValueError` — ein Parameter hatte den richtigen Typ, aber einen falschen Wert

Built-In Exceptions (4)



- `BaseException` – die Basisklasse aller Ausnahmen
  - `Exception` – Situationen, wo vernüftige Fehlerbehandlung möglich seien sollte
    - `ValueError` – ein Parameter hatte den richtigen Typ, aber einen falschen Wert
      - `UnicodeError` – Fehler bein Verarbeiten von Unicode Text
        - `UnicodeDecodeError` – Fehler bein Verarbeiten von Unicode Text
        - `UnicodeEncodeError` – Fehler bein Verarbeiten von Unicode Text
        - `UnicodeTranslateError` – Fehler bein Verarbeiten von Unicode Text
  - `GeneratorExit` – kein Fehler, sondern ein `Generator` oder eine Coroutine sind fertig
  - `KeyboardInterrupt` – der Nutzer drückte Ctrl + C
  - `SystemExit` – kein Fehler, wird von `exit` ausgelöst

Zusammenfassung



Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.

Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.
- Wir haben nun die Mittel, um unseren Kode gegen einige Benutzerfehler zu schützen.

Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.
- Wir haben nun die Mittel, um unseren Kode gegen einige Benutzerfehler zu schützen.
- Natürlich können wir nicht verhindern, dass jemand unseren Kode mit Argumenten falschen Typs aufruft.

Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.
- Wir haben nun die Mittel, um unseren Kode gegen einige Benutzerfehler zu schützen.
- Natürlich können wir nicht verhindern, dass jemand unseren Kode mit Argumenten falschen Typs aufruft.
- Wir können auch nicht verhindern, dass wir fehlerhafte Parameterwerte als Eingabe bekommen.

Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.
- Wir haben nun die Mittel, um unseren Kode gegen einige Benutzerfehler zu schützen.
- Natürlich können wir nicht verhindern, dass jemand unseren Kode mit Argumenten falschen Typs aufruft.
- Wir können auch nicht verhindern, dass wir fehlerhafte Parameterwerte als Eingabe bekommen.
- Wenn wir aber in der Lage sind, bestimmte falsche Werte zu erkennen, dann können wir eine Ausnahme auslösen.

Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.
- Wir haben nun die Mittel, um unseren Kode gegen einige Benutzerfehler zu schützen.
- Natürlich können wir nicht verhindern, dass jemand unseren Kode mit Argumenten falschen Typs aufruft.
- Wir können auch nicht verhindern, dass wir fehlerhafte Parameterwerte als Eingabe bekommen.
- Wenn wir aber in der Lage sind, bestimmte falsche Werte zu erkennen, dann können wir eine Ausnahme auslösen.
- Wir können uns weigern, GIGO zu machen.

Zusammenfassung



- Wieder haben wir einen wichtigen Schritt hin zum professionellen Programmieren gemacht.
- Wir haben nun die Mittel, um unseren Kode gegen einige Benutzerfehler zu schützen.
- Natürlich können wir nicht verhindern, dass jemand unseren Kode mit Argumenten falschen Typs aufruft.
- Wir können auch nicht verhindern, dass wir fehlerhafte Parameterwerte als Eingabe bekommen.
- Wenn wir aber in der Lage sind, bestimmte falsche Werte zu erkennen, dann können wir eine Ausnahme auslösen.
- Wir können uns weigern, GIGO zu machen.
- Wir können uns weigern, Ergebnisse, die auf falschen Annahmen beruhen, zurückzuliefern.

Zwei Dinge

- Wir müssen aber zwei Dinge bedenken.

Zwei Dinge



- Wir müssen aber zwei Dinge bedenken:
  1. Es ist nicht möglich, alle fehlerhaften Eingabedaten zu endecken. Wenn der Benutzer `sqrt(3.1)` aufrufen wollte, aber aus Versehen `sqrt(1.3)` aufruft, dann können wir das unmöglich wissen.

Zwei Dinge



- Wir müssen aber zwei Dinge bedenken:
  1. Es ist nicht möglich, alle fehlerhaften Eingabedaten zu endecken. Wenn der Benutzer `sqrt(3.1)` aufrufen wollte, aber aus Versehen `sqrt(1.3)` aufruft, dann können wir das unmöglich wissen.
  2. Eingabeparameter zu überprüfen kommt mit Performanzkosten. Wenn unsere `sqrt`-Funktionen millionenfach in einer Schleife aufgerufen wird um einen Strom von Eingabedaten zu verarbeiten ... wollen wir dann wirklich jedesmal die Datentypen „manuell“ prüfen?

Zwei Dinge



- Wir müssen aber zwei Dinge bedenken:
  1. Es ist nicht möglich, alle fehlerhaften Eingabedaten zu endecken. Wenn der Benutzer `sqrt(3.1)` aufrufen wollte, aber aus Versehen `sqrt(1.3)` aufruft, dann können wir das unmöglich wissen.
  2. Eingabeparameter zu überprüfen kommt mit Performanzkosten. Wenn unsere `sqrt`-Funktionen millionenfach in einer Schleife aufgerufen wird um einen Strom von Eingabedaten zu verarbeiten ... wollen wir dann wirklich jedesmal die Datentypen „manuell“ prüfen?
- Es ergibt Sinn, alle Eingabedaten extrem genau zu prüfen, wenn unsere Funktion entweder sowieso lange braucht oder nicht oft aufgerufen wird.

Zwei Dinge



- Wir müssen aber zwei Dinge bedenken:
  1. Es ist nicht möglich, alle fehlerhaften Eingabedaten zu endecken. Wenn der Benutzer `sqrt(3.1)` aufrufen wollte, aber aus Versehen `sqrt(1.3)` aufruft, dann können wir das unmöglich wissen.
  2. Eingabeparameter zu überprüfen kommt mit Performanzkosten. Wenn unsere `sqrt`-Funktionen millionenfach in einer Schleife aufgerufen wird um einen Strom von Eingabedaten zu verarbeiten ... wollen wir dann wirklich jedesmal die Datentypen „manuell“ prüfen?
- Es ergibt Sinn, alle Eingabedaten extrem genau zu prüfen, wenn unsere Funktion entweder sowieso lange braucht oder nicht oft aufgerufen wird.
- Wenn unser Kode schnell sein muss oder sehr sehr oft aufgerufen wird, dann ist es sinnvoll, weniger explizite Überprüfungen durchzuführen.

Zwei Dinge



- Wir müssen aber zwei Dinge bedenken:
  1. Es ist nicht möglich, alle fehlerhaften Eingabedaten zu endecken. Wenn der Benutzer `sqrt(3.1)` aufrufen wollte, aber aus Versehen `sqrt(1.3)` aufruft, dann können wir das unmöglich wissen.
  2. Eingabeparameter zu überprüfen kommt mit Performanzkosten. Wenn unsere `sqrt`-Funktionen millionenfach in einer Schleife aufgerufen wird um einen Strom von Eingabedaten zu verarbeiten ... wollen wir dann wirklich jedesmal die Datentypen „manuell“ prüfen?
- Es ergibt Sinn, alle Eingabedaten extrem genau zu prüfen, wenn unsere Funktion entweder sowieso lange braucht oder nicht oft aufgerufen wird.
- Wenn unser Kode schnell sein muss oder sehr sehr oft aufgerufen wird, dann ist es sinnvoll, weniger explizite Überprüfungen durchzuführen.
- Sicheres Programmieren ist immer ein Trade-off.

谢谢您们！

Thank you!

Vielen Dank!



References I

[1] 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. 258, 259).

[2] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 260).

[3] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 258).

[4] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 257).

[5] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 257).

[6] "Built-in `Exception`S". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/exceptions.html (besucht am 2024-10-29) (siehe S. 220–230).

[7] Josh Centers. *Take Control of iOS 18 and iPadOS 18*. San Diego, CA, USA: Take Control Books, Dez. 2024. ISBN: 978-1-990783-55-5 (siehe S. 257).

[8] 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. 259, 260).

[9] Justin Dennison, Cherokee Boose und Peter van Rysdam. *Intro to NumPy*. Centennial, CO, USA: ACI Learning. Birmingham, England, UK: Packt Publishing Ltd, Juni 2024. ISBN: 978-1-83620-863-1 (siehe S. 258).

[10] "Formatted String Literals". In: *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 7.1.1. URL: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals (besucht am 2024-07-25) (siehe S. 257).

[11] Alessandro F. Garcia, andCecília M. F. Rubira, Alexander B. Romanovsky und Jie Xu. "A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software". *Journal of Systems and Software* 59(2):197–222, 15. Nov. 2001. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0164-1212. doi:10.1016/S0164-1212(01)00062-0 (siehe S. 23–35).

References II

[12] Bhavesh Gawade. "Mastering F-Strings in Python: Efficient String Handling in Python Using Smart F-Strings". In: *C O D E B*. Mumbai, Maharashtra, India: Code B Solutions Pvt Ltd, 25. Apr.–3. Juni 2025. URL: https://code-b.dev/blog/f-strings-in-python (besucht am 2025-08-04) (siehe S. 257).

[13] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: https://peps.python.org/pep-0257 (besucht am 2024-07-27) (siehe S. 257).

[14] Olaf Górski. "Why f-strings are awesome: Performance of different string concatenation methods in Python". In: *DEV Community*. Sacramento, CA, USA: DEV Community Inc., 8. Nov. 2022. URL: https://dev.to/grski/performance-of-different-string-concatenation-methods-in-python-why-f-strings-are-awesome-2e97 (besucht am 2025-08-04) (siehe S. 257).

[15] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli „pv" Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke und Travis E. Oliphant. "Array programming with NumPy". *Nature* 585:357–362, 2020. London, England, UK: Springer Nature Limited. ISSN: 0028-0836. doi:10.1038/S41586-020-2649-2 (siehe S. 258).

[16] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 258).

[17] 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. 260).

[18] 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. 258).

[19] John D. Hunter. "Matplotlib: A 2D Graphics Environment". *Computing in Science & Engineering* 9(3):90–95, Mai–Juni 2007. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2007.55 (siehe S. 258).

[20] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom und The Matplotlib Development Team. *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: https://matplotlib.org (besucht am 2025-02-02) (siehe S. 258).

References III

[21] *Information Technology – Universal Coded Character Set (UCS)*. International Standard ISO/IEC 10646:2020. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Dez. 2020 (siehe S. 260).

[22] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4246-9 (siehe S. 258).

[23] .“exit – Terminate a Process”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Hrsg. von Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/exit.html (besucht am 2024-10-30) (siehe S. 257).

[24] Andrew Josey, Hrsg. *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799 (besucht am 2024-10-30).

[25] .“stderr, stdin, stdout – Standard I/O Streams”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Hrsg. von Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/stdin.html (besucht am 2024-10-30) (siehe S. 259).

[26] Donald Ervin Knuth. *Fundamental Algorithms*. 3. Aufl. Bd. 1 der Reihe The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1997. ISBN: 978-0-201-89683-1 (siehe S. 258).

[27] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 259).

[28] 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. 258).

[29] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 258).

References IV

[30] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 258).

[31] Aaron Maxwell. *What are f-strings in Python and how can I use them?* Oakville, ON, Canada: Infinite Skills Inc, Juni 2017. ISBN: 978-1-4919-9486-3 (siehe S. 257).

[32] Pedro Mejia Alvarez, Raul E. Gonzalez Torres und Susana Ortega Cisneros. *Exception Handling – Fundamentals and Programming*. SpringerBriefs in Computer Science. Cham, Switzerland: Springer, Feb. 2024. ISSN: 2191-5768. ISBN: 978-3-031-50680-2. doi:10.1007/978-3-031-50681-9 (siehe S. 23–35).

[33] 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. 257).

[34] NumPy Team. *NumPy*. San Francisco, CA, USA: GitHub Inc und Austin, TX, USA: NumFOCUS, Inc. URL: https://numpy.org (besucht am 2025-02-02) (siehe S. 258).

[35] 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. 260).

[36] 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. 260).

[37] Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python 3*. New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1 (siehe S. 258).

[38] 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. 260).

References V

[39] 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. 257).

[40] Tim Peters. *The Zen of Python*. Python Enhancement Proposal (PEP) 20. Beaverton, OR, USA: Python Software Foundation (PSF), 19.–22. Aug. 2004. URL: https://peps.python.org/pep-0020 (besucht am 2025-08-03) (siehe S. 36–39).

[41] Amit Phalgun, Cory Kissinger, Margaret M. Burnett, Curtis R. Cook, Laura Beckwith und Joseph R. Ruthruff. "Garbage in, Garbage out? An Empirical Look at Oracle Mistakes by End-User Programmers". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'2005)*. 21.–24. Sep. 2005, Dallas, TX, USA. Hrsg. von Martin Erwig und Andy Schürr. Los Alamitos, CA, USA: IEEE Computer Society, 2005, S. 45–52. ISSN: 1943-6092. ISBN: 978-0-7695-2443-6. doi:10.1109/VLHCC.2005.40 (siehe S. 23–35, 257).

[42] Ernest E. Rothman, Rich Rosen und Brian Jepson. *Mac OS X for Unix Geeks*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2008. ISBN: 978-0-596-52062-5 (siehe S. 258).

[43] 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. 260).

[44] Ahmad Sahar. *iOS 26 Programming for Beginners*. 10. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2025. ISBN: 978-1-80602-393-6 (siehe S. 260).

[45] Ali Shahrokni und Robert Feldt. "A Systematic Review of Software Robustness". *Information and Software Technology* 55(1):1–17, Jan. 2013. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0950-5849. doi:10.1016/J.INFSOF.2012.06.002. URL: http://robertfeldt.net/publications/shahrokni_2013_sysrev_robustness.pdf (besucht am 2024-10-29) (siehe S. 23–35).

[46] 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. 258).

[47] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 257).

References VI

[48] Drew Smith. *Modern Apple Platform Administration – macOS and iOS Essentials (2025)*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2025. ISBN: 978-1-80580-309-6 (siehe S. 257, 258).

[49] Eric V. „ericvsmith“ Smith. *Literal String Interpolation*. Python Enhancement Proposal (PEP) 498. Beaverton, OR, USA: Python Software Foundation (PSF), 6. Nov. 2016–9. Sep. 2023. URL: https://peps.python.org/pep-0498 (besucht am 2024-07-25) (siehe S. 257).

[50] *The Unicode Standard, Version 15.1: Archived Code Charts*. South San Francisco, CA, USA: The Unicode Consortium, 25. Aug. 2023. URL: https://www.unicode.org/Public/15.1.0/charts/CodeCharts.pdf (besucht am 2024-07-26) (siehe S. 260).

[51] 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. 260).

[52] 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. 258).

[53] 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. 257, 260).

[54] *Unicode® 15.1.0*. South San Francisco, CA, USA: The Unicode Consortium, 12. Sep. 2023. ISBN: 978-1-936213-33-7. URL: https://www.unicode.org/versions/Unicode15.1.0 (besucht am 2024-07-26) (siehe S. 260).

[55] Bruce M. Van Horn II und Quan Nguyen. *Hands-On Application Development with PyCharm*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-235-0 (siehe S. 258).

[56] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 259).

[57] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: https://peps.python.org/pep-0008 (besucht am 2024-07-27) (siehe S. 257).

References VII

[58] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 258).

[59] Pauli „pv“ Virtanen, Ralf Gommers, Travis E. Oliphant, Matt „mdhaber“ Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan „ilayn“ Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregos, Paul van Mulbregt und SciPy 1.0 Contributors. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. *Nature Methods* 17:261–272, 2. März 2020. London, England, UK: Springer Nature Limited. ISSN: 1548-7091. doi:10.1038/s41592-019-0686-2. URL: http://arxiv.org/abs/1907.10121 (besucht am 2024-06-26). See also arXiv:1907.10121v1 [cs.MS] 23 Jul 2019. (Siehe S. 258).

[60] 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. 258).

[61] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 258).

[62] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2022. ISBN: 978-1-83763-244-2 (siehe S. 258).

[63] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 257).

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[4,33,63]. Learn more at https://www.gnu.org/software/bash.

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[13]. They must be delimited by `"""..."""`[13,57].

exit code When a process terminates, it can return a single integer value (the exit status code) to indicate success or failure[23]. Per convention, an exit code of 0 means success. Any non-zero exit code indicates an error. Under Python, you can terminate the current process at any time by calling `exit` and optionally passing in the exit code that should be returned. If `exit` is not explicitly called, then the interpreter will return an exit code of 0 once the process normally terminates.If the process was terminated by an uncaught `Exception`, a non-zero exit code, usually 1, is returned.

f-string let you include the results of expressions in strings[5,10,12,14,31,49]. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.

GIGO Garbage In–Garbage Out, see, e.g., [41]

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[47,53]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[39,53]. Learn more at https://github.com.

IDE An *Integrated Developer Environment* is a program that allows the user do multiple different activities required for software development in one single system. It often offers functionality such as editing source code, debugging, testing, or interaction with a distributed version control system. For Python, we recommend using PyCharm. On Apple systems, Xcode is often used.

iOS is the operating system that powers Apple iPhones[7,48]. Learn more at https://www.apple.com/ios.

iPadOS is the operating system that powers Apple iPads[7]. Learn more at https://www.apple.com/ipados.

Glossary (in English) II

IT	information technology
Linux	is the leading open source operating system, i.e., a free alternative to Microsoft Windows[1,16,46,52,58]. 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.
macOS	or Mac OS is the operating system that powers Apple Mac(intosh) computers[42,48]. Learn more at https://www.apple.com/macos.
Matplotlib	is a Python package for plotting diagrams and charts[19,20,22,37]. Learn more at at https://matplotlib.org[20].
Microsoft Windows	is a commercial proprietary operating system[3]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.
Mypy	is a static type checking tool for Python[29] that makes use of type hints. Learn more at https://github.com/python/mypy and in [60].
NumPy	is a fundamental package for scientific computing with Python, which offers efficient array data structures[9,15,22]. Learn more at https://numpy.org[34].
PyCharm	is the convenient Python Integrated Development Environment (IDE) that we recommend for this course[55,61,62]. It comes in a free edition, so it can be downloaded and used at no cost. Learn more at https://www.jetbrains.com/pycharm.
Python	The Python programming language[18,28,30,60], i.e., what you will learn about in our book[60]. Learn more at https://python.org.
SciPy	is a Python library for scientific computing[32,59]. Learn more at https://scipy.org.
stack trace	A stack trace gives information the way in which one function invoked another. The term comes from the fact that the data needed to implement function calls is stored in a stack data structure[26]. The data for the most recently invoked function is on top, the data of the function that called is right below, the data of the function that called that one comes next, and so on. Printing a stack trace can be very helpful when trying to find out where an `Exception` occurred.

Glossary (in English) III

`stderr` The *standard error stream* is one of the three pre-defined streams of a console process (together with the `stdin` and the `stdout`)[25]. It is the text stream to which the process writes information about errors and exceptions. If an uncaught `Exception` is raised in Python and the program terminates, then this information is written to `stderr`. If you run a program in a terminal, then the text that a process writes to its `stderr` appears in the console.

`stdin` The *standard input stream* is one of the three pre-defined streams of a console process (together with the `stdout` and the `stderr`)[25]. It is the text stream from which the process reads its input text, if any. The Python instruction `input` reads from this stream. If you run a program in a terminal, then the text that you type into the terminal while the process is running appears in this stream.

`stdout` The *standard output stream* is one of the three pre-defined streams of a console process (together with the `stdin` and the `stderr`)[25]. It is the text stream to which the process writes its normal output. The `print` instruction of Python writes text to this stream. If you run a program in a terminal, then the text that a process writes to its `stdout` appears in the console.

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

terminal A terminal is a text-based window where you can enter commands and execute them[1,8]. 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.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[27,56]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

Glossary (in English) IV



Ubuntu is a variant of the open source operating system Linux[8,17]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

Unicode A standard for assigning characters to numbers[21,50,54]. The Unicode standard supports basically all characters from all languages that are currently in use, as well as many special symbols. It is the predominantly used way to represent characters in computers and is regularly updated and improved.

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[2,35,36,38,43,51]. 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[53]. 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.

Xcode is offers the tools for developing, testing, and distributing applications as well as an IDE for Apple platforms such as macOS and iOS[44].